# Build Your Own React: Design Document

## Overview

This document outlines the architecture for a React-like UI library built from scratch. It tackles the core challenge of declarative, high-performance UI updates by implementing a Virtual DOM, a reconciliation diffing algorithm, an interruptible Fiber-based renderer, and a Hooks system for state and side effects. The key architectural challenge is managing the complexity of tree updates while maintaining performance and developer ergonomics.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

## 1. Context and Problem Statement

> **Milestone(s):** This section establishes the foundational problem all subsequent milestones (Virtual DOM, Reconciliation, Fiber, Hooks) are designed to solve.

Building modern web applications involves managing complex, dynamic user interfaces that update frequently in response to user interactions, data changes, and other events. The fundamental challenge is updating the Document Object Model (DOM) — the browser's representation of the webpage — efficiently and predictably as application state changes. This section explores why directly manipulating the DOM is problematic, surveys historical approaches, and introduces the core architectural pattern that React popularized: the Virtual DOM with reconciliation.

### The Problem: Direct DOM Manipulation is Painful

Imagine you're building a complex Lego structure. Each time you want to change it, you have two possible approaches:

1. **Imperative Approach (Direct DOM Manipulation):** You completely disassemble the entire structure and rebuild it from scratch according to your new design, even if 90% of the pieces remain in the same place. This is slow, labor-intensive, and error-prone. You might misplace pieces, forget connections, or introduce structural instability.
2. **Declarative Approach (Virtual DOM):** You maintain a blueprint (a detailed plan) of your desired final structure. When you want to make a change, you draw a new blueprint. Then, you compare the new blueprint to the old one, identify the specific differences (e.g., "swap these two red blocks," "add a blue block here"), and execute only those minimal changes on the physical Lego model.

**Direct DOM manipulation is the imperative approach.** The DOM API is inherently imperative: you command the browser to create elements ( `document.createElement` ), set attributes ( `element.setAttribute` ), and append children ( `parent.appendChild` ). As application logic grows, tracking which parts of the DOM need to change for a given state update becomes a cognitive nightmare. Developers must manually orchestrate sequences of DOM operations, leading to code that is:

| Problem | Description | Consequence |
|---|---|---|
| **Brittle and Hard to Maintain** | Business logic becomes intertwined with DOM update instructions. Changing the UI often requires rewriting complex, state-dependent update sequences. | Bugs proliferate. Adding new features becomes risky and time-consuming. |
| **Inefficient** | Without careful analysis, developers often update more of the DOM than necessary. Unnecessary reflows and repaints degrade performance, especially in complex UIs. | Janky, unresponsive user experiences. |
| **Non-Compositional** | It's difficult to break UI into independent, reusable components because each component's internal DOM manipulation might conflict with others. | Code duplication and tight coupling between components. |
| **Error-Prone** | Manually synchronizing application state with DOM state is a classic source of bugs (e.g., showing stale data, forgetting to remove event listeners). | UI gets out of sync with data, leading to confusing user experiences. |

The core issue is the **mismatch of paradigms**. Application logic is best expressed *declaratively* (what the UI should look like given the current state), but the browser's DOM API is *imperative* (how to change it). This forces developers to act as human "diffing engines," mentally calculating the minimal set of DOM operations needed—a process that is both tedious and unreliable at scale.

> **Key Insight:** The complexity doesn't come from drawing the UI once; it comes from *updating* it efficiently and correctly in response to countless state changes over the application's lifetime. We need a way to describe the UI as a function of state ( `UI = f(state)` ) and let the system figure out the imperative steps to get there.

### Existing Approaches and React's Solution

Before React, the web development community explored several strategies to manage UI complexity. Each represents a point on the spectrum between pure imperative control and full declarative abstraction.

| Approach | Description | Pros | Cons |
|---|---|---|---|
| **Naive Re-render** | On every state change, discard the entire DOM subtree and rebuild it from a template or string (e.g., `innerHTML = newHTML`). | Simple to implement. Always produces a UI perfectly in sync with state. | Extremely inefficient. Loses all UI state (focus, scroll position, video playback). Security risk with `innerHTML` if not sanitized. |
| **Manual Diffing (Observer Pattern)** | Manually listen for specific state changes and write targeted DOM update functions for each possible change. Common in early MVC frameworks. | Can be highly optimized for known patterns. | Exponential complexity ( `O(n²)` relationships between state and DOM). Becomes unmanageable as application grows. Tight coupling. |
| **Dirty Checking (AngularJS)** | Periodically compare current application state to previous state. For any changed data, update all DOM bindings that depend on it. | Declarative templates. No manual event listeners for updates. | Performance degrades with number of bindings. Change detection cycles can be confusing. Over-renders (updates more than necessary). |
| **Reactive Programming (MobX, Vue)** | Use observable data structures and automatic dependency tracking. When data changes, precisely re-run computations (like render functions) that depend on it. | Granular, efficient updates. Excellent developer experience. | Requires adopting a reactive data model. The "reactivity system" itself is complex to implement. |

React introduced a different architectural division of labor. Its solution is built on two core ideas:

1. **Virtual DOM:** A lightweight, in-memory JavaScript object representation of the real DOM tree. It's cheap to create and manipulate.
2. **Reconciliation (Diffing Algorithm):** A process that compares the new Virtual DOM tree with the previous one, computes the difference (*diff*), and calculates the **minimal, optimal set of imperative DOM operations** required to synchronize the real DOM.

> **Mental Model:** Think of the Virtual DOM as a **detailed blueprint** and the reconciliation algorithm as a **skilled foreman**. You, the developer, only draw new blueprints ( `React.createElement` ). The foreman compares the new blueprint to the old one, creates a precise "work order" of changes (e.g., "move wall A, replace window B"), and gives it to the construction crew (the renderer) to execute on the real building (the DOM). This splits the problem: you think declaratively, while the system handles the imperative optimization.

The data flow follows a predictable pattern:

1. **Render:** Application state changes, triggering a re-render. This generates a new Virtual DOM tree (a blueprint).
2. **Diff:** The new Virtual DOM tree is compared with the previous one.
3. **Patch:** The calculated differences are applied to the real DOM.

This architecture offers significant benefits:

| Benefit | Explanation |
|---|---|
| **Declarative API** | Developers describe the *desired* UI state. The library determines the *how*. This drastically reduces cognitive load and bug surface area. |
| **Performance Optimization** | The diffing algorithm minimizes expensive DOM operations. While not *free* (JavaScript computation is required), it's often faster than manual management or naive re-renders for complex updates. |
| **Abstraction Over Browser Quirks** | The library can handle cross-browser inconsistencies and provide a consistent API for features like event handling. |
| **Component Model** | The declarative nature enables composition of self-contained components, each managing its own Virtual DOM subtree. |

However, the initial React reconciliation algorithm had a critical flaw: it was **synchronous and recursive**. The diffing process, which can be CPU-intensive for large component trees, ran in a single, uninterrupted JavaScript task. If the tree was deep, this could block the main thread for long enough (>16ms for 60fps) to cause dropped frames, making the UI feel unresponsive. This is the problem that later led to the Fiber architecture (Milestone 3), which made rendering interruptible and schedulable.

> **Architecture Decision Record: Virtual DOM as a Primary Abstraction Context:** We need a bridge between the declarative UI descriptions (components) and the imperative DOM API. The bridge must be efficient and enable powerful features like server-side rendering and testing. **Options Considered:**
>
> 1. **Direct DOM Patching with Observables:** Emit precise DOM operations directly from state changes (like in Svelte). This can yield optimal bundle size and runtime performance for known patterns but requires a sophisticated compile-time analysis.
> 2. **Incremental DOM (Google Closure Library):** Walk a template and apply changes to the real DOM node-by-node, reusing existing nodes. Memory efficient but can be harder to optimize for complex diffing scenarios.
> 3. **Virtual DOM (React):** Maintain a complete JavaScript copy of the DOM tree to diff against. **Decision:** Implement a Virtual DOM system. **Rationale:** For an educational "build your own" project, the Virtual DOM provides the clearest separation of concerns. The diffing algorithm is a distinct, understandable module. It's also the most generalizable pattern, forming the foundation for concurrent features (Fiber). The mental model of "blueprint diffing" is highly intuitive. **Consequences:** We introduce a memory overhead for the JavaScript object trees and CPU overhead for the diffing computation. We must design an efficient diffing algorithm (Milestone 2) to mitigate this.

The following table summarizes the core concepts we will implement to realize this architecture:

| Concept | Role in Our Architecture | Corresponding Milestone |
|---|---|---|
| **Virtual DOM Node (VNode)** | The fundamental unit of the blueprint. A plain JavaScript object describing a DOM element or text node. | Milestone 1 |
| **Reconciliation (Diffing)** | The algorithm that compares two VNode trees and produces a list of DOM patches. | Milestone 2 |
| **Fiber Node** | An extended VNode with links to facilitate interruptible, incremental rendering. | Milestone 3 |
| **Renderer** | The subsystem that takes the final list of DOM patches (or a Fiber tree) and executes them on the real browser DOM. | Milestones 1 & 3 |
| **Hooks** | A mechanism for function components to "hook into" the Fiber node's lifecycle and state storage. | Milestone 4 |

This document guides you through building these concepts from the ground up, starting with the static Virtual DOM and culminating in a dynamic, interruptible renderer with stateful hooks. By understanding each layer, you will gain deep insight into how modern declarative UI libraries operate under the hood.

## Implementation Guidance

This section is conceptual; there is no code to implement yet. However, setting up your project structure correctly from the start is crucial. We recommend a simple, modular layout that mirrors the separation of concerns in our architecture.

**A. Technology Recommendations Table:**

| Component | Simple Option | Advanced Option |
|---|---|---|
| **Language** | Vanilla JavaScript (ES6+) | TypeScript for type safety |
| **Build Tool** | None (run directly in browser with ES modules) | Vite or Parcel for bundling and development server |
| **Testing** | Browser console and manual inspection | Jest + jsdom for unit tests |

**B. Recommended File/Module Structure:** Create the following directory and file structure. This keeps related code together and separates the public API from internal reconciler logic.

```
build-your-own-react/
├── index.html                  # HTML entry point to test your library
├── package.json                # Project configuration (if using Node/npm)
├── src/
│   ├── index.js                # Main public API (createElement, render)
│   ├── reconciler.js           # Diffing algorithm & fiber logic (Milestones 2 & 3)
│   ├── hooks.js                # useState, useEffect implementations (Milestone 4)
│   └── utils.js                # Shared utilities and constants
└── examples/                   # Example applications for testing
    ├── hello-world.js
    ├── counter.js
    └── todo-app.js
```

**C. Infrastructure Starter Code:** Create a basic `package.json` if you plan to use npm or a similar tool. This is optional if you are loading scripts directly in the browser.

```json
// package.json (optional)
{
  "name": "build-your-own-react",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "live-server ."  // Simple static server if you have live-server
  }
}
```

**D. Core Logic Skeleton Code:** For now, we will just create placeholder files with their intended exports. The next section (Milestone 1) will fill these in.

```javascript
// src/index.js
/**
 * Public API of our React-like library.
 */

// TODO Milestone 1: Implement createElement
export function createElement(type, props, ...children) {
  // This function should create and return a Virtual DOM node object.
}

// TODO Milestone 1: Implement render
export function render(vnode, container) {
  // This function should mount a Virtual DOM tree into a real DOM container.
}
```

```javascript
// src/reconciler.js
/**
 * Internal reconciliation and fiber architecture logic.
 */

// TODO Milestone 2 & 3: Implement diffing, fibers, and the work loop.
export function reconcile(/* ... */) {
  // Compare old and new trees, schedule updates.
}
```

```javascript
// src/hooks.js

/**
 * Hooks implementation.
 */

// TODO Milestone 4: Implement useState and useEffect.
export function useState(initialState) {
  // Return [state, setState].
}

export function useEffect(effect, deps) {
  // Schedule side effect.
}
```

**E. Language-Specific Hints:**

- Use ES6 modules ( `import` / `export` ) for clean separation. You can run this directly in modern browsers by setting `<script type="module">` in your `index.html` .
- Use `document.createElement` , `element.textContent` , `element.setAttribute` , and `element.appendChild` for basic DOM operations in the renderer.
- Use `requestIdleCallback` for the Fiber work loop scheduler (Milestone 3). Note: This is a browser API and may require a polyfill for wider support.

**F. Milestone Checkpoint:** After reading this section, you should have:

1. A clear understanding of the problems solved by the Virtual DOM and reconciliation pattern.
2. A project directory created with the skeleton files above.
3. An `index.html` file that includes your main script as a module ( `<script type="module" src="./src/index.js"></script>` ).

You are now ready to proceed to Milestone 1 and implement the `createElement` and `render` functions.

## 2. Goals and Non-Goals

> **Milestone(s):** This section establishes the foundational scope for all four milestones. It defines the precise capabilities we intend to build (Goals) and explicitly carves out a set of related, but non-essential, features we will not implement (Non-Goals). This clarity is critical for managing the complexity of an ambitious educational project and ensuring focus on the core architectural principles of React.

This section delineates the precise boundaries of our "Build Your Own React" project. Given the expansive nature of a full-featured UI library, it is essential to constrain our implementation to a well-defined set of core concepts. The **Goals** represent the **must-have** pillars of the React mental model: a declarative API, efficient updates, non-blocking rendering, and a hooks-based state management system. Achieving these four goals will result in a functional, educational library that demonstrates the foundational architecture of React. Conversely, the **Non-Goals** are a list of important but non-essential features commonly found in production-grade frameworks. We explicitly exclude them to maintain project focus and tractability, acknowledging that they represent natural extensions for future exploration.

## 2.1 Goals (Must-Have Features)

The following features constitute the minimal viable product for our educational React clone. Each goal is a direct reflection of a core architectural principle we aim to understand and implement.

| Goal | Description | Corresponding Milestone | Key Architectural Principle |
|---|---|---|---|
| **Declarative Virtual DOM API** | Provide a `createElement` function to construct a lightweight JavaScript object tree ( `VNode` ) describing the desired UI structure. A `render` function will initially translate this description into actual browser DOM elements. This separates the *what* (the declared UI state) from the *how* (the imperative DOM updates). | Milestone 1: Virtual DOM | **Declarative Rendering:** The UI is a function of application state. |
| **Efficient Updates via Reconciliation** | Implement a **diffing algorithm** that compares the previous `VNode` tree with the new one, calculates the minimal set of mutations required, and applies them to the real DOM. This avoids expensive full re-renders and is the performance heart of React-like libraries. | Milestone 2: Reconciliation | **Incremental Updates:** Only update what changed, not the entire subtree. |
| **Non-Blocking Render Scheduling with Fiber** | Implement the **Fiber architecture**, breaking rendering work into small, interruptible units of work ( `Fiber` nodes). Use a scheduler (e.g., `requestIdleCallback` ) to yield control back to the browser to ensure high-priority tasks (like animations or input) aren't blocked by rendering. | Milestone 3: Fiber Architecture | **Cooperative Scheduling:** The renderer yields to the browser to maintain responsiveness. |
| **Hooks API for State and Effects** | Implement `useState` and `useEffect` hooks, enabling function components to manage local state and side effects. This requires a mechanism to associate hook data with specific component instances across re-renders, enforcing the **Rules of Hooks**. | Milestone 4: Hooks | **Stateful Function Components:** Components can have persistent, encapsulated memory. |

The successful implementation of these four goals results in a library where a developer can write a component like the example below and have it behave predictably: rendering efficiently, updating based on state changes without blocking the UI, and cleaning up side effects.

> **Mental Model: The Four Pillars of a Modern UI Library** Imagine building a house. The **Declarative API** is your blueprint ( `VNode` ). The **Reconciliation** process is the foreman who, given a new blueprint, instructs the crew on the *exact* nails to hammer and boards to replace, rather than rebuilding the whole house. The **Fiber Scheduler** is the project manager who breaks the foreman's list into small tasks and ensures the crew takes breaks so the homeowners (the browser's main thread) can still live in the house. Finally, **Hooks** are the utilities (plumbing, electrical) installed within the walls of each room (component) that remember their state (water pressure, light switch position) between renovations (re-renders).

## 2.2 Non-Goals (Out of Scope)

To maintain a clear focus on educational value and avoid scope creep, we explicitly rule out the following features. While many are critical for a production-ready library, they are either complementary enhancements to our core architecture or represent significant additional complexity that would distract from understanding the fundamentals.

| Non-Goal | Rationale for Exclusion | Potential Implementation Complexity |
|---|---|---|
| **Server-Side Rendering (SSR) & Hydration** | SSR involves rendering components to a string on the server and then "hydrating" that static markup into an interactive client-side application. While a powerful pattern, it introduces a separate rendering environment and a complex reconciliation process between server-generated markup and client-side `VNode`s. | High. Requires a second render-to-string pipeline and a delicate hydration diffing algorithm. |
| **Synthetic Event System** | React's normalized, cross-browser event wrapper system. Our library will use native browser events ( `element.addEventListener` ). Implementing a synthetic system involves event delegation, pooling, and complex normalization logic that is not core to the rendering architecture. | Medium-High. Involves event delegation at the root, custom event objects, and lifecycle management. |
| **React DevTools Integration** | The browser extension for inspecting component trees, state, and performance. While invaluable for debugging, it requires a stable public API for the renderer to communicate with the extension, which is a separate project in itself. | High. Requires a defined bridge protocol and stable internal hooks. |
| **Portals, Fragments, and Special Component Types** | `ReactDOM.createPortal` allows rendering subtrees into different DOM containers. `React.Fragment` lets you group children without a wrapper node. These are ergonomic APIs built on top of the core architecture but are not required for its fundamental operation. | Medium. Portals require special handling in the fiber tree and commit phase. |
| **Context API ( `useContext` , `Provider` )** | A mechanism for passing data through the component tree without prop drilling. It is a specific form of state propagation that can be initially mimicked via prop drilling or implemented later as an extension of the fiber tree's internal propagation mechanisms. | Medium. Requires creating a publish-subscribe mechanism linked to the fiber tree reconciliation. |
| **Concurrent Features (e.g., `useTransition` , `Suspense` )** | Advanced React 18 features that build upon the Fiber architecture to enable more intelligent interruption and data fetching integration. These represent the *advanced* use of our Milestone 3 foundation but are a significant leap in scheduling complexity. | Very High. Involves complex priority-based scheduling lanes and promise integration. |
| **Full React API Surface ( `PureComponent` , `memo` , `ref` API, `ErrorBoundary` )** | We implement the core functional paradigm with hooks. Class components ( `Component` , `PureComponent` ), the `memo` higher-order component for optimization, the full `ref` object API, and `ErrorBoundary` lifecycle methods are omitted for simplicity, though the architecture could be extended to support them. | Varies (Medium-High). Each requires additional fields in the fiber node and specific logic in the reconciliation phase. |
| **Custom Renderers (React Native, React ART)** | Our renderer is tightly coupled to the browser's DOM. Building an abstraction layer to target different environments (e.g., canvas, native mobile) is a separate, major architectural undertaking. | Very High. Requires a complete abstraction of host environment operations. |

**Architecture Decision Record: Scope Definition for Educational Clarity**

- **Context:** We are building an educational clone of React to understand its core architectural pillars. The full React API is vast and includes many features that are either performance optimizations, developer ergonomics, or advanced patterns built upon a stable foundation.
- **Options Considered:**
  1. **Implement a Minimal Core:** Focus exclusively on the data flow and algorithms that enable declarative, efficient UI updates (Virtual DOM, Diffing, Fiber, Hooks).
  2. **Implement a Production Subset:** Include a broader set of features like Synthetic Events, Context, and `memo` to more closely mirror real-world React development.
  3. **Implement a Fully Compatible Subset:** Aim for near-complete API compatibility for a small subset of features, potentially sacrificing clarity in implementation for API familiarity.
- **Decision:** Choose Option 1 (Minimal Core).
- **Rationale:** The primary objective is *understanding*, not creating a drop-in replacement. Each additional feature dilutes focus and adds layers of complexity that can obscure the foundational algorithms. By implementing only the four pillars, a developer can fully grasp *how* React works at its core. The chosen goals are sequential and inter-dependent, creating a coherent learning journey. Features listed as Non-Goals are excellent follow-on projects once the core is understood.
- **Consequences:**
  - **Enables:** Deep focus on algorithm implementation (diffing, fiber tree traversal, hook linked list management). The resulting codebase is smaller and easier to reason about.
  - **Trade-offs:** The library is not suitable for production use without significant additional work. Developers must be aware of API differences (e.g., using `onClick` vs. `onclick` natively).

The table below summarizes the key distinctions between our educational scope and a full production framework.

| Aspect | Our Educational Library | Production React |
|---|---|---|
| **Primary Aim** | Understand core architecture | Build robust, performant applications |
| **Rendering Target** | Browser DOM only | DOM, Native, Canvas, etc. via renderers |
| **Event System** | Native DOM events | Synthetic, normalized event system |
| **Scheduling** | Basic `requestIdleCallback` cooperative yielding | Advanced, priority-based lane scheduling |
| **Component Types** | Function Components with Hooks only | Function & Class Components |
| **State/Context** | `useState`, `useEffect` | Full suite: `useReducer`, `Context`, Redux integration |
| **Optimizations** | Reconciliation via keys | Plus: `memo`, `useMemo`, `useCallback` |
| **API Surface** | ~4 core functions | Hundreds of APIs and helpers |

By adhering to these Goals and Non-Goals, we ensure the project remains a tractable yet deeply enlightening exploration of modern UI library architecture.

## Implementation Guidance

### A. Technology Recommendations Table

| Component | Simple Option (Recommended) | Advanced Option (For Exploration) |
|---|---|---|
| **Language & Environment** | Vanilla JavaScript (ES6+) in a browser environment. Use a simple HTML file with script tags or a bundler like Parcel/Vite for module support. | TypeScript for type safety and better developer experience, compiled to ES6. |
| **DOM Abstraction** | Direct use of `document.createElement`, `element.setAttribute`, `element.addEventListener`. | Use a minimal helper library for creating SVG elements or namespaced attributes, but generally stick to the DOM API. |
| **Scheduler** | `window.requestIdleCallback` (or a `setTimeout` polyfill for browsers without support). | Implement a custom priority-based scheduler with multiple lanes, simulating React's Scheduler package. |
| **Testing** | Browser-based manual testing with `console.log` and a test application. Use Jest + jsdom for unit testing core functions. | Use a full testing library (React Testing Library patterns) on top of your own renderer for integration tests. |
| **Build Tool** | None, or a zero-config bundler like Parcel to handle JSX transformation via Babel plugin. | Custom Babel plugin to transform JSX directly into your `createElement` calls. |

**B. Recommended File/Module Structure**

Given the focused scope, a flat or simply organized module structure is sufficient. This keeps imports straightforward.

```
build-your-own-react/
├── index.js (or index.html)          # Main entry point and public API export
├── core.js                           # Milestone 1 & 2: createElement, render (initial), diff/patch
├── reconciler.js                     # Milestone 2 & 3: Fiber node definition, work loop, reconciliation logic
├── scheduler.js                      # Milestone 3: requestIdleCallback wrapper and work loop trigger
├── hooks.js                          # Milestone 4: useState, useEffect, hook dispatcher
├── test-app/                         # Directory for manual testing applications
│   ├── app.js
│   └── index.html
└── tests/                            # Unit tests (if using a test runner)
    ├── core.test.js
    ├── reconciler.test.js
    └── hooks.test.js
```

**C. Infrastructure Starter Code**

For the absolute beginning, here is a complete HTML test harness you can use to run your library. It includes a polyfill for `requestIdleCallback` for broader browser support.

**File:** `test-app/index.html`

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Test Build Your Own React</title>
    <style>
        /* Optional: Add some basic styles for your test app */
    </style>
</head>
<body>
    <div id="root"></div>

    <script>
        // Polyfill for requestIdleCallback
        window.requestIdleCallback = window.requestIdleCallback ||
            function (cb) {
                return setTimeout(function () {
                    var start = Date.now();
                    cb({
                        didTimeout: false,
                        timeRemaining: function () {
                            return Math.max(0, 50 - (Date.now() - start));
                        }
                    });
                }, 1);
            };
        window.cancelIdleCallback = window.cancelIdleCallback ||
            function (id) {
                clearTimeout(id);
            };
    </script>

    <!-- Load your library. In a real setup, use a module bundler. -->
    <script src="../core.js"></script>
```

```
    <script src="../reconciler.js"></script>
    <script src="../scheduler.js"></script>
    <script src="../hooks.js"></script>

    <!-- Your test application script -->
    <script src="app.js"></script>
</body>
</html>
```

**D. Core Logic Skeleton Code**

While the core logic will be filled in later sections, the public API signatures are defined here to set expectations.

**File: `core.js` - Public API Stubs**

```javascript
/**
 * Creates a virtual DOM node (VNode).
 * @param {string|Function} type - The HTML tag name or a function component.
 * @param {Object|null} props - The properties/attributes of the node.
 * @param {...any} children - Child elements (can be VNodes, strings, numbers, arrays).
 * @returns {VNode} A virtual DOM node object.
 */
function createElement(type, props, ...children) {
    // TODO 1: Flatten children array and handle primitive values (string/number).
    // TODO 2: Return a plain object with structure { type, props, children }.
    // TODO 3: (Later) Handle function component types.
}

/**
 * Renders a VNode tree into a real DOM container (initial mount).
 * @param {VNode} vnode - The root virtual node.
 * @param {DOMElement} container - The real DOM container (e.g., a div).
 */
function render(vnode, container) {
    // TODO 1: Convert the initial VNode tree to DOM elements.
    // TODO 2: Append the created DOM tree to the container.
    // TODO 3: (Later) This will be replaced by the Fiber renderer's initial work scheduling.
}

// Export the public API
export { createElement, render };
```

**E. Language-Specific Hints**

- **Children Handling:** Remember that `children` in `createElement` is a rest parameter (`...children`). It will be an array, but children themselves can be arrays (e.g., from `map` calls). You will need a helper to flatten the children list.
- **Text Nodes:** When a child is a string or number, you must create a *virtual text node*. A common convention is to give it a special `type` (e.g., `'TEXT_ELEMENT'`) and store the raw value in a property like `nodeValue`.
- **Props Assignment:** When setting props on a DOM element, remember that `style` should be an object, event handlers start with 'on', and `className` is used instead of `class` (which is a reserved word in JS). Our initial `render` will handle these directly.

**F. Milestone Checkpoint** After implementing the **Goals** for Milestone 1 (Virtual DOM), you should be able to run the following test application and see a static UI rendered in the browser.

**File:** `test-app/app.js`

```javascript
// Assuming createElement and render are available globally from core.js
const { createElement, render } = window.myReactLib; // You'd attach them to window or use modules.

const staticApp = createElement('div', { className: 'container' },
    createElement('h1', null, 'Hello, World'),
    createElement('p', { style: { color: 'blue' } }, 'This is a static render.'),
    createElement('button', { onClick: () => alert('Clicked!') }, 'Click me')
);

render(staticApp, document.getElementById('root'));
```

**Expected Outcome:** The browser displays a page with a title, a blue paragraph, and a button. Clicking the button triggers a native browser alert. This confirms that `createElement` correctly builds the `VNode` tree and `render` successfully creates and attaches real DOM elements, including event handlers.

## 3. High-Level Architecture

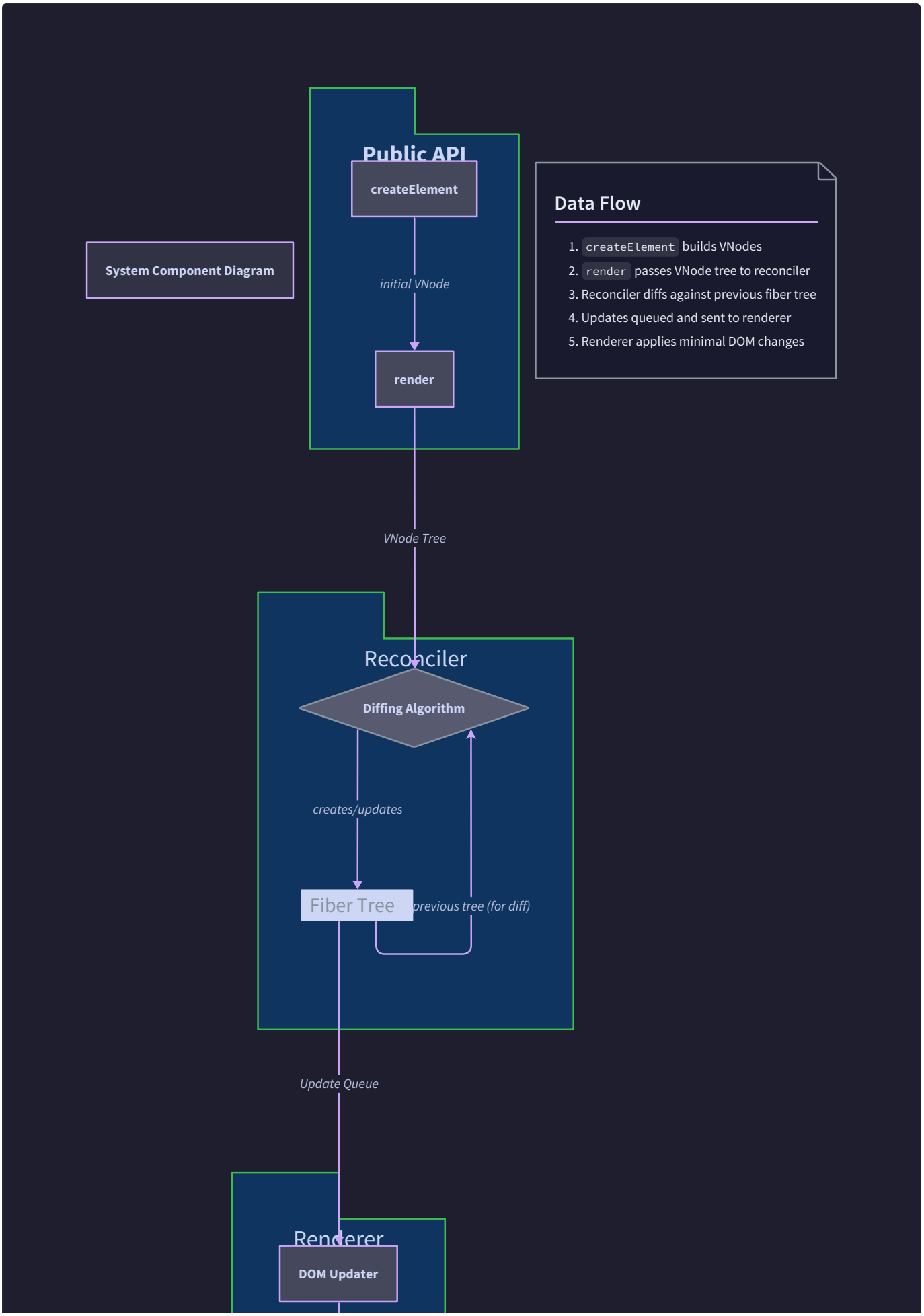
> **Milestone(s):** This section provides the architectural overview that connects all four subsequent milestones (Virtual DOM, Reconciliation, Fiber Architecture, and Hooks). It establishes the conceptual foundation and organizational structure for the entire system.

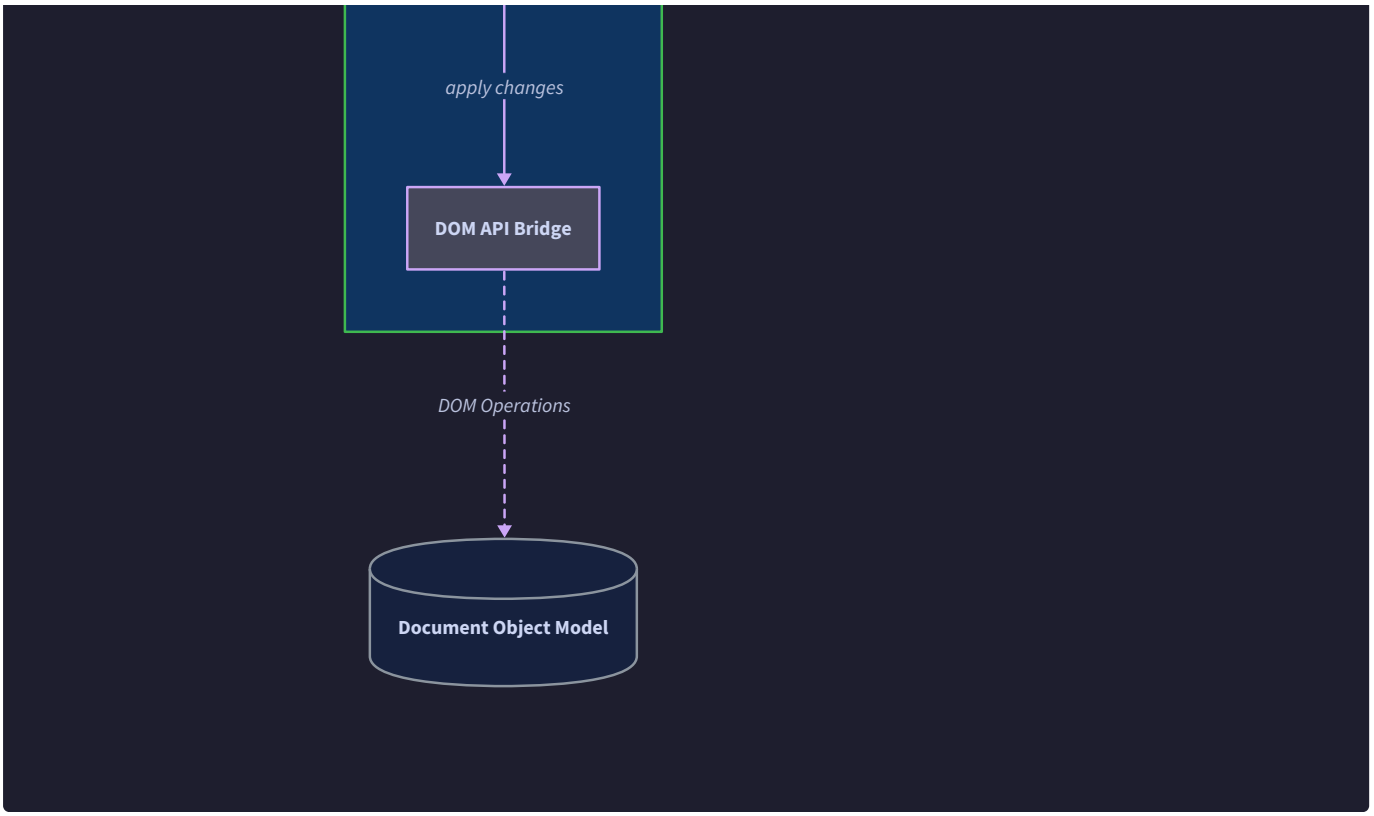
At its core, our React-like library transforms a declarative component description into an interactive user interface, efficiently updating the UI when state changes. This transformation occurs through a carefully orchestrated pipeline of four major subsystems working in concert. Think of building a complex model from architectural blueprints: first you create the detailed plans (Virtual DOM), then you figure out what's changed since the last build (Reconciliation), then you schedule your construction crew to work without blocking other tasks (Fiber Architecture), and finally you manage the ongoing maintenance and utilities of the finished structure (Hooks).

### Component Overview and Responsibilities

The library decomposes into four interdependent pillars, each handling a distinct phase of the render-and-update lifecycle. These components form a unidirectional data flow from component definitions to screen pixels.

| Component | Primary Responsibility | Key Inputs | Key Outputs | Lifecycle Phase |
|---|---|---|---|---|
| **Virtual DOM (Blueprint Layer)** | Creates and maintains lightweight JavaScript object representations of the desired UI structure | Component JSX/function calls, props, state | `VNode` trees (virtual nodes) | Creation & Description |
| **Reconciliation Engine (Diffing Layer)** | Compares consecutive `VNode` trees to compute minimal update instructions | Previous `VNode` tree, new `VNode` tree | Update "diff" payload (add, remove, update operations) | Comparison & Planning |
| **Fiber Scheduler (Execution Layer)** | Breaks rendering work into incremental units, schedules them, and applies final changes to the real DOM | `VNode` trees, update diffs, state changes | Mutated real `DOMElement` nodes | Scheduling & Application |
| **Hooks Runtime (State & Effects Layer)** | Provides state persistence and side-effect management for function components | Hook function calls, dependency arrays | State values, effect callbacks, cleanup functions | State Management & Side Effects |


System Component Diagram
Public API
createElement
initial VNode
render
Data Flow
1. createElement builds VNodes
2. render passes VNode tree to reconciler
3. Reconciler diffs against previous fiber tree
4. Updates queued and sent to renderer
5. Renderer applies minimal DOM changes
VNode Tree
Reconciler
Diffing Algorithm
creates/updates
Fiber Tree
previous tree (for diff)
Update Queue
Renderer
DOM Updater




The architecture follows a **"render-then-commit"** model with two distinct phases:

1. **Render/Reconciliation Phase**: The library traverses the component tree, calls component functions, builds the new `VNode` tree, and compares it with the previous one. During this phase, no changes are made to the real DOM—it's pure computation.
2. **Commit Phase**: After the entire tree has been processed and all necessary updates are calculated, the library applies all DOM mutations in a single, synchronous batch. This ensures the UI updates atomically, preventing partial updates that could break visual consistency.

The relationship between these components during a state update follows a precise sequence:

- A `setState` call (from a hook) marks a component as needing update
- The Fiber Scheduler adds this work to its queue
- When the browser is idle, the scheduler begins the render phase
- The Reconciliation Engine compares old and new `VNode` trees for that component
- Updates are recorded as "effects" on Fiber nodes
- After the entire tree is processed, the commit phase flushes all effects to the DOM

> **Architecture Insight**: This separation between render (computing what changed) and commit (applying changes) is fundamental. It enables features like concurrent rendering—the render phase can be interrupted and resumed, but the commit phase always runs to completion once started. This prevents the UI from displaying inconsistent intermediate states.

## Recommended File/Module Structure

A well-organized codebase mirrors the architectural separation of concerns, making the system easier to understand, test, and extend. The following structure groups related functionality while maintaining clear interfaces between subsystems.

**Project Root Layout:**

```
build-your-own-react/
├── package.json
├── tsconfig.json (if using TypeScript)
├── src/
│   ├── index.js (or index.ts)          # Public API entry point
│   ├── core/                           # Milestone 1: Virtual DOM fundamentals
│   │   ├── createElement.js
│   │   ├── render.js
│   │   └── vnode.js                    # VNode type definition & utilities
│   ├── reconciler/                     # Milestone 2 & 3: Diffing & Fiber
│   │   ├── diff.js                     # Reconciliation algorithm
│   │   ├── patch.js                    # DOM update application
│   │   ├── scheduler.js                # Work loop & requestIdleCallback
│   │   ├── fiber.js                    # Fiber node structure & traversal
│   │   └── commitWork.js               # Commit phase implementation
│   ├── hooks/                          # Milestone 4: Hooks system
│   │   ├── useState.js
│   │   ├── useEffect.js
│   │   ├── hook.js                     # Hook instance structure & dispatcher
│   │   └── utils.js                    # Rules of Hooks validation
│   └── utils/
│       ├── constants.js                # Effect tags, update types
│       ├── domProperties.js            # DOM property mapping
│       └── validation.js               # Development warnings
├── test/                               # Test suites
│   ├── core.test.js
│   ├── reconciler.test.js
│   ├── hooks.test.js
│   └── integration.test.js
└── examples/                           # Demo applications
    ├── counter/
    ├── todo-list/
    └── hooks-demo/
```

**Module Dependencies Flow:**

```
index.js (Public API)
  → imports from core/ (createElement, render)
  → imports from reconciler/ (scheduler for updates)
  → re-exports hooks (useState, useEffect)

core/
  → createElement → vnode utilities
  → render → reconciler/scheduler (for initial mount)

reconciler/
  → scheduler → fiber traversal logic
  → diff → patch → DOM manipulation
  → commitWork → applies fiber effects

hooks/
  → hook dispatcher → reconciler/fiber (access current fiber)
  → useState/useEffect → share hook instance management
```

**Decision: Layered Architecture vs. Monolithic**

- **Context**: We need to organize the codebase for clarity and maintainability while allowing learners to understand each milestone's components.
- **Options Considered**:
  1. **Single File Monolith**: All code in one file for simplicity
  2. **Feature-Based Grouping**: Group by milestone/feature with clear interfaces
  3. **Framework-Style Separation**: Split into packages (react, react-dom, scheduler)
- **Decision**: Feature-based grouping with clear module boundaries
- **Rationale**: A single file becomes unwieldy (~1000+ lines), obscuring architectural boundaries. Framework-style separation adds unnecessary complexity for an educational project. Feature-based grouping mirrors the conceptual milestones while maintaining clear separation of concerns—learners can focus on one folder at a time.
- **Consequences**: Imports/exports add some boilerplate but make testing and understanding dependencies explicit. The structure can evolve naturally as features are added.

**Key Module Interfaces:**

| Module | Exports | Description |
|---|---|---|
| `src/index.js` | `{ createElement, render, useState, useEffect }` | Public API - everything a component developer uses |
| `src/core/createElement.js` | `createElement(type, props, ...children)` | Transforms JSX into `VNode` trees |
| `src/core/render.js` | `render(vnode, container)` | Initial mount entry point, kicks off first render |
| `src/reconciler/scheduler.js` | `scheduleUpdate(fiber)`, `workLoop()` | Manages render scheduling via browser idle periods |
| `src/reconciler/diff.js` | `reconcileChildren(wipFiber, children)` | Core diffing algorithm for child lists |
| `src/hooks/useState.js` | `useState(initialValue)` | State hook implementation |
| `src/hooks/hook.js` | `Dispatcher`, `resolveCurrentlyRenderingFiber()` | Hook runtime coordination |

This structure supports incremental implementation: you can complete and test Milestone 1 (core/) before moving to Milestone 2 (reconciler/diff.js), then extend the reconciler with fiber architecture, and finally implement hooks that integrate with the fiber system.

## Implementation Guidance

**Technology Recommendations Table:**

| Component | Simple Option | Advanced Option |
|---|---|---|
| **Build Tool** | No build tool (vanilla JS with script tags) | Parcel/Vite for hot reload and ES modules |
| **Testing** | Manual browser testing with console logs | Jest + jsdom for unit tests |
| **Type Safety** | Plain JavaScript with JSDoc comments | TypeScript for compile-time checks |
| **Development Server** | Simple HTTP server (Python, `http-server`) | Webpack Dev Server with hot reload |

**Recommended File/Module Structure Implementation:**

Create the following directory and file structure to begin implementation. The entry point file below establishes the public API.

**File: `src/index.js` (Complete Starter Code)**

```javascript
/**
 * Build Your Own React - Main Entry Point
 * Re-exports the public API of the library
 */

import { createElement } from './core/createElement.js';
import { render } from './core/render.js';
import { useState } from './hooks/useState.js';
import { useEffect } from './hooks/useEffect.js';

// Public API matching React's core primitives
export { createElement, render, useState, useEffect };

// Optional: Support for JSX pragma (see ADR in Milestone 1)
// Developers can set /** @jsx createElement */ in their files
// or configure Babel to use createElement automatically
```

**File: `src/core/vnode.js` (Complete Starter Code)**

JAVASCRIPT

```javascript
/**
 * Virtual Node (VNode) type definition and utilities
 * This is the fundamental data structure representing UI elements
 */

/**
 * @typedef {Object} VNode
 * @property {string | Function} type - HTML tag name or function component
 * @property {Object} props - Properties/attributes including children
 * @property {Array<VNode | string | number>} props.children - Child nodes
 * @property {string | number | null} key - Optional unique identifier for lists
 * @property {DOMElement} _dom - Internal reference to corresponding DOM node (set during render)
 */

/**
 * Creates a virtual node object with consistent structure
 * @param {string | Function} type
 * @param {Object} props
 * @param {...(VNode | string | number)} children
 * @returns {VNode}
 */
export function createVNode(type, props = {}, ...children) {
  // Flatten children array and handle primitive values
  const flatChildren = children.reduce((acc, child) => {
    if (Array.isArray(child)) {
      return [...acc, ...child];
    }
    return [...acc, child];
  }, []);

  // Handle text nodes: convert strings/numbers to text VNodes
  const normalizedChildren = flatChildren.map(child => {
    if (typeof child === 'string' || typeof child === 'number') {
      return createTextVNode(child);
    }
```

```
    // Handle falsey values (null, undefined, false) as empty nodes
    if (child == null || child === false) {
      return createEmptyVNode();
    }
    return child;
  }).filter(child => child !== null);

  const vnode = {
    type,
    props: {
      ...props,
      children: normalizedChildren
    },
    key: props.key || null
  };

  // Internal reference will be set during render
  vnode._dom = null;

  return vnode;
}

/**
 * Creates a text virtual node
 * @param {string | number} textValue
 * @returns {VNode}
 */
function createTextVNode(textValue) {
  return {
    type: 'TEXT_ELEMENT',
    props: {
      nodeValue: String(textValue),
      children: []  // Text nodes have no children
    },
    key: null,
```

```
    _dom: null
  };
}

/**
 * Creates an empty virtual node (for falsey values)
 * @returns {VNode|null}
 */
function createEmptyVNode() {
  return null;  // Will be filtered out
}

export { createTextVNode, createEmptyVNode };
```

**File:** `src/utils/constants.js` **(Complete Starter Code)**

JAVASCRIPT

```javascript
/**
 * Constants used throughout the reconciliation and fiber systems
 */

// Effect tags - what needs to be done to a fiber during commit
export const EFFECT_TAGS = {
  PLACEMENT: 'PLACEMENT',      // New node needs to be added to DOM
  UPDATE: 'UPDATE',            // Existing node needs props updated
  DELETION: 'DELETION',        // Node needs to be removed from DOM
  PLACEMENT_AND_UPDATE: 'PLACEMENT_AND_UPDATE' // Both add and update
};

// Update types for state updates
export const UPDATE_TYPES = {
  STATE: 'STATE',
  EFFECT: 'EFFECT'
};

// Special DOM property names
export const DOM_PROPERTIES = {
  // Event handlers (onClick, onChange, etc.)
  EVENTS: /^on[A-Z]/,
  // Properties that should be set as attributes vs object properties
  ATTRIBUTES: ['className', 'htmlFor', 'value', 'checked', 'selected']
};

// Fiber node types
export const FIBER_TYPES = {
  HOST: 'HOST',           // DOM element (div, span, etc.)
  FUNCTION: 'FUNCTION',   // Function component
  CLASS: 'CLASS',         // Class component (not implemented)
  TEXT: 'TEXT'           // Text node
};
```

**File:** `src/utils/domProperties.js` **(Complete Starter Code)**

JAVASCRIPT

```javascript
/**
 * Utilities for handling DOM properties and attributes correctly
 * Critical for proper event handling and attribute setting
 */

import { DOM_PROPERTIES } from './constants.js';

/**
 * Sets properties on a DOM element, handling special cases
 * @param {DOMElement} domElement - Target DOM element
 * @param {string} propName - Property name
 * @param {any} propValue - Property value
 */
export function setDOMProperty(domElement, propName, propValue) {
  // Handle event listeners
  if (DOM_PROPERTIES.EVENTS.test(propName)) {
    const eventName = propName.toLowerCase().substring(2); // onClick -> click
    // Remove previous listener if it exists
    if (domElement._listeners && domElement._listeners[propName]) {
      domElement.removeEventListener(eventName, domElement._listeners[propName]);
    }
    // Store reference for future removal
    if (!domElement._listeners) domElement._listeners = {};
    domElement._listeners[propName] = propValue;
    domElement.addEventListener(eventName, propValue);
    return;
  }

  // Handle special properties that map to attributes
  if (DOM_PROPERTIES.ATTRIBUTES.includes(propName)) {
    const attributeName = propName === 'className' ? 'class' :
                          propName === 'htmlFor' ? 'for' : propName;
    if (propValue == null) {
      domElement.removeAttribute(attributeName);
    } else {
```

```
      domElement.setAttribute(attributeName, propValue);
    }
    return;
  }

  // Handle style object
  if (propName === 'style' && typeof propValue === 'object') {
    Object.assign(domElement.style, propValue);
    return;
  }

  // Handle children specially (not a DOM property)
  if (propName === 'children') {
    return;
  }

  // Default: set as object property
  // Note: Some properties like 'value' on inputs need both attribute and property
  domElement[propName] = propValue;
}

/**
 * Removes properties from a DOM element
 * @param {DOMElement} domElement - Target DOM element
 * @param {string} propName - Property name to remove
 * @param {any} prevValue - Previous property value
 */
export function removeDOMProperty(domElement, propName, prevValue) {
  // Handle event listeners
  if (DOM_PROPERTIES.EVENTS.test(propName)) {
    const eventName = propName.toLowerCase().substring(2);
    if (prevValue && domElement._listeners && domElement._listeners[propName]) {
      domElement.removeEventListener(eventName, domElement._listeners[propName]);
      delete domElement._listeners[propName];
    }
```

```
    return;
  }

  // Handle special attributes
  if (DOM_PROPERTIES.ATTRIBUTES.includes(propName)) {
    const attributeName = propName === 'className' ? 'class' :
                          propName === 'htmlFor' ? 'for' : propName;
    domElement.removeAttribute(attributeName);
    return;
  }

  // Handle style object
  if (propName === 'style' && typeof prevValue === 'object') {
    Object.keys(prevValue).forEach(styleName => {
      domElement.style[styleName] = '';
    });
    return;
  }

  // Default: delete object property
  // Be careful with built-in properties
  if (propName in domElement) {
    domElement[propName] = null;
  }
}
```

**Language-Specific Hints for JavaScript:**

- Use ES6 modules ( `import` / `export` ) for clean separation of concerns
- Use `document.createElement` and `document.createTextNode` for DOM creation
- Use `requestIdleCallback` for scheduling (with fallback to `setTimeout` for older browsers)
- Store hook state directly on the fiber node using a linked list structure
- Use `instanceof` checks sparingly; prefer duck typing with property checks
- Use `Object.keys()` for prop diffing instead of deep equality for performance

**Next Steps**: With this architectural foundation and file structure in place, you can begin implementing Milestone 1 by creating the `createElement` function (in `src/core/createElement.js` ) that uses the `createVNode` utility, then the `render` function that initializes the first render and starts the reconciliation process.

## 4. Data Model

> **Milestone(s):** This section defines the foundational data structures for Milestone 1 (Virtual DOM), Milestone 3 (Fiber Architecture), and Milestone 4 (Hooks). These structures serve as the backbone of the entire system, representing UI elements, work units, and component state.

The **data model** is the scaffolding upon which our React-like library is built. It defines how we represent user interfaces in memory, how we track work to be performed, and how we preserve component state across renders. These structures transform the declarative JSX code developers write into an internal representation that our reconciliation and rendering algorithms can efficiently process. Think of these data structures as the **DNA of your UI**—they encode everything needed to construct, update, and manage the component tree.

### 4.1 Virtual Node (VNode) Structure

**Mental Model: The Blueprint** Imagine you're an architect designing a house. You wouldn't start by immediately ordering bricks and lumber; instead, you'd first create detailed blueprints—lightweight drawings that specify the structure's layout, materials, and dimensions. A **Virtual Node (VNode)** serves exactly this purpose: it's a lightweight JavaScript object that describes what a particular piece of the UI should look like, without the heavyweight machinery of an actual DOM element. Just as blueprints are cheap to modify and compare, VNodes allow us to compute the minimal changes needed before we touch the expensive, real DOM.

A VNode is a plain JavaScript object with three essential properties that together form a tree structure mirroring your component hierarchy. This tree is the **Virtual DOM**—a complete in-memory representation of your UI.

The following table details every field in a VNode:

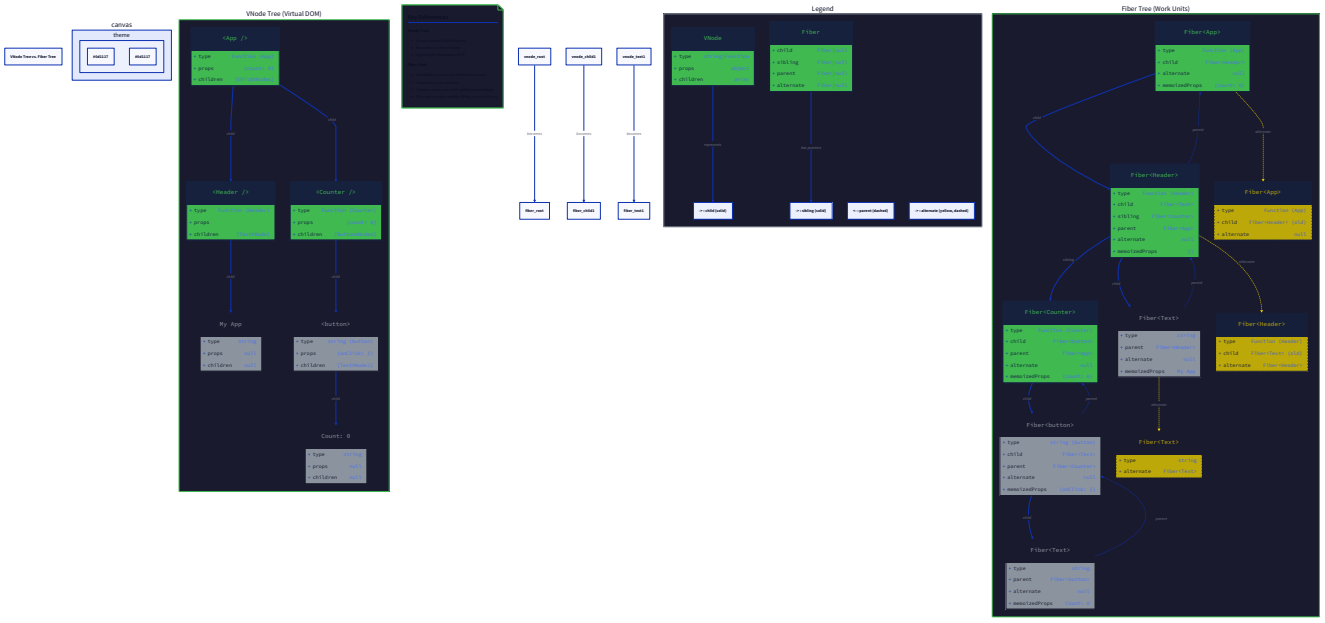
| Field Name | Type | Description |
|---|---|---|
| `type` | `string \| Function` | The core identifier of what this node represents. If `type` is a string (like `"div"`, `"span"`), it's a **host element** (a regular HTML/SVG tag). If `type` is a function, it's a **function component**—a JavaScript function that returns more VNodes. This distinction is critical for the renderer and reconciler. |
| `props` | `object` | A dictionary containing all properties (or "props") that should be applied to the corresponding DOM element or passed to a function component. This includes standard HTML attributes (`className`, `id`), inline styles (`style`), **event listeners** (functions under keys like `onClick`), and a special `children` property if children are passed as a prop. The `props` object is the primary vehicle for passing data and behavior down the component tree. |
| `key` | `string \| number \| null` | An optional stable identifier used by the reconciliation algorithm to track nodes across re-renders, especially within lists. When the order of children changes, keys allow the system to reorder existing DOM nodes rather than destroying and recreating them. If not provided, defaults to `null`. |
| `_dom` (Internal) | `DOMElement \| null` | A **private reference** to the actual DOM element that this VNode currently represents in the real browser DOM. This field is not set when the VNode is first created; it's populated during the initial `render` or during reconciliation updates. It acts as a crucial bridge between the virtual and real worlds, allowing the diffing algorithm to directly update the existing DOM node instead of searching for it. |

> **Design Insight:** The `_dom` field is marked with an underscore to indicate it's an internal implementation detail that library consumers should not directly modify. This follows the principle of encapsulation—hiding internal state to prevent accidental corruption.

**Example VNode Tree Structure:** Consider the JSX expression `<div id="root"><h1>Hello</h1><button>Click</button></div>`. This translates to a VNode tree where the root `div` VNode has a `type` of `"div"`, `props` of `{ id: "root" }`, and two child VNodes under its `children` array (or via `props.children`). The `h1` child has a `type` of `"h1"` and a single text child. Text

nodes themselves are represented as VNodes with `type` equal to the string `"TEXT_ELEMENT"` (a special constant) and `props` containing a `nodeValue` property.

The relationship between VNodes forms a classic parent-child tree, which is simple to create and traverse but insufficient for the pause-and-resume rendering model required by the Fiber architecture. This is why we later introduce the more sophisticated **Fiber node**.




## 4.2 Fiber Node Structure

**Mental Model: The Task Card in a Kanban Board** If VNodes are static blueprints, **Fiber nodes** are dynamic task cards on a Kanban board. Each card represents a unit of work (rendering a component, updating a DOM element) and contains not only the work description but also metadata about its status ( `effectTag` ), relationships to other tasks ( `child` , `sibling` , `parent` ), and a link to the previous version of the task ( `alternate` ). The Fiber architecture allows us to break the rendering work into small, interruptible units, just like you can pause working on one task card and switch to another without losing your place.

A Fiber node is an enhanced representation of a VNode, expanded with additional fields necessary for scheduling, reconciliation, and tracking side effects. Each Fiber corresponds to one VNode, but the Fiber tree is a **linked list** (specifically, a linked list of trees) that enables efficient, linear traversal without recursion. This structure is the heart of React's concurrent rendering capabilities.

The following table details every field in a Fiber node. Fields are grouped by primary responsibility: **identity & structure**, **reconciliation state**, **hooks & state**, and **internal pointers**.

| Field Name | Type | Description |
| --- | --- | --- |
| **Identity & Structure** | | |
| `type` | `string \| Function \| null` | Inherited from the VNode. Identifies the component or host element. For root fibers or placeholder nodes, this may be `null`. |
| `key` | `string \| number \| null` | Inherited from the VNode. Stable identifier for list reconciliation. |
| `props` | `object` | The current props for this fiber. During reconciliation, we compare `oldFiber.props` with `newVNode.props` to detect changes. |
| `stateNode` | `DOMElement \| object \| null` | Reference to the actual instance associated with this fiber. For host fibers (HTML elements), this is the **real DOM node**. For function components, this is `null` (or could hold hooks state via `memoizedState`). For class components (not in our scope), this would be the component instance. |
| **Reconciliation & Scheduling Links** | | |
| `child` | `Fiber \| null` | Pointer to the **first child** fiber. This establishes the primary hierarchical link, analogous to the first child in a tree. |
| `sibling` | `Fiber \| null` | Pointer to the **next sibling** fiber. Together with `child`, this allows us to traverse the entire component tree as a singly-linked list (child-first, depth-first). |
| `parent` | `Fiber \| null` | Pointer to the **parent** fiber. Used primarily during the commit phase to find the DOM parent when inserting or removing nodes. |
| `alternate` | `Fiber \| null` | One of the most critical fields. Points to the **previous version** of this fiber from the last committed render. During reconciliation, we compare the current "work-in-progress" fiber (`workInProgress`) with its `alternate` (the "current" tree) to determine what changed. This is how we avoid recreating the entire tree on every render. |
| `effectTag` | `number` (from `EFFECT_TAGS`) | A flag indicating what type of side effect (DOM mutation) needs to be performed on this fiber during the commit phase. Examples: `PLACEMENT` (needs insertion), `UPDATE` (needs property updates), `DELETION` (needs removal). A fiber can have multiple effect tags combined via bitwise OR. |
| `nextEffect` | `Fiber \| null` | Pointer to the next fiber in a **singly-linked list of fibers with side effects**. This list is built during the render phase and iterated during the commit phase to apply all DOM updates efficiently. It only includes fibers where `effectTag !== NO_EFFECT`. |
| **Hooks & Component State** | | |
| `memoizedState` | `HookInstance \| any \| null` | For function components, this points to the **head of the linked list of Hook instances** (for `useState`, `useEffect`, etc.). For host components, this is `null`. In class components, it would hold the component instance state. |
| `updateQueue` | `UpdateQueue \| null` | A queue of state updates (for `useState`) or effect definitions (for `useEffect`) that need to be processed for this fiber. This queue is flushed during the render phase to compute the new state. |
| **Internal Render Phase State** | | |
| `tag` | `number` (from `FIBER_TYPES`) | Classifies the fiber type: `HOST` (HTML/SVG element), `FUNCTION` (function component), `CLASS` (class component), `TEXT` (text node). This determines how the fiber is processed |

| Field Name | Type | Description |
|---|---|---|
| | | during reconciliation. |
| `pendingProps` | `object` | The new props that have been passed to this fiber during the current render. At the end of the render phase, `pendingProps` becomes `props`. |
| `memoizedProps` | `object` | The props used to create the current output during the last render. We compare `memoizedProps` with `pendingProps` to detect prop changes. |

> **Architecture Decision: Why a Linked List Instead of a Tree?** The traditional VNode tree is a recursive data structure, which requires deep recursion to traverse. Recursion cannot be paused mid-execution in JavaScript—once a recursive function starts, it must run to completion. By transforming the tree into a linked list of fibers (using `child`, `sibling`, and `parent` pointers), we can traverse the component tree using a simple while loop. This loop can be interrupted after processing any single fiber, enabling the cooperative scheduling that is central to the Fiber architecture. The `alternate` pointer then allows us to diff the new list against the old list efficiently.

**Fiber Tree Traversal Example:** Consider a component tree: `App` → `div` → `Header` (sibling) → `Content`. The Fiber representation would have `App` fiber with `child` pointing to `div`. The `div` fiber has `child` pointing to `Header` and `sibling` pointing to `Content`. `Header` has `sibling` pointing to `Content`. This structure allows a depth-first traversal: start at `App`, go to its child `div`, then to `div`'s child `Header`, then to `Header`'s sibling `Content`. The `parent` pointers allow walking back up when needed.

### 4.3 Hook Instance Structure

**Mental Model: Memory Cells on a Component's Shelf** Function components are stateless by nature—they're plain JavaScript functions that run fresh on every render. **Hooks** provide these functions with "memory" by giving them access to persistent storage attached to the component's Fiber node. Imagine each function component has a shelf (the Fiber's `memoizedState`). Each call to `useState` or `useEffect` allocates a new "memory cell" (a Hook instance) on that shelf. These cells are stored in order of invocation, forming a linked list. On subsequent renders, when the component function runs again, the hook system walks down this shelf, handing back the stored values from the corresponding memory cells. This mechanism is why **hooks must always be called in the same order**—it's like reading from a numbered list of shelves; if you skip a shelf, you'll get the wrong data.

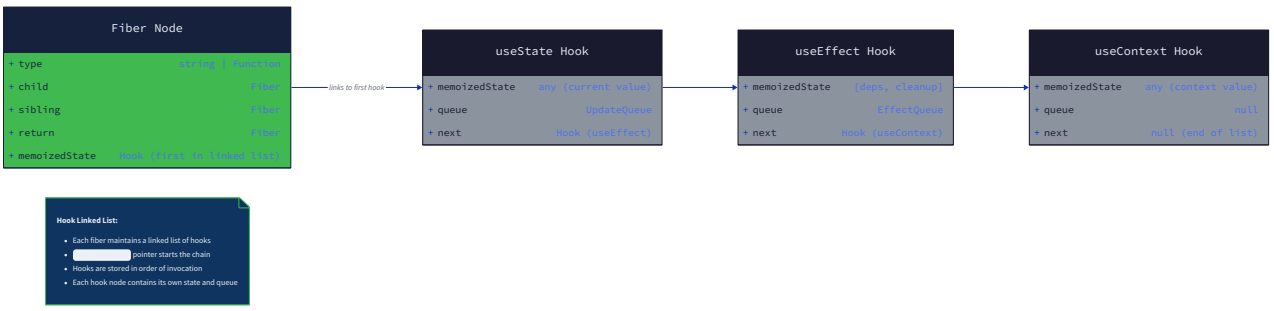
A Hook instance is a small object that stores the stateful information for a single hook call within a function component. Multiple hook instances are linked together in the order they were called, forming a list anchored to the fiber's `memoizedState` field.

The following table details every field in a Hook instance:

| Field Name | Type | Description |
|---|---|---|
| `memoizedState` | `any` | The **current value** of the hook. For `useState`, this is the current state value. For `useEffect`, this is the **dependency array** from the previous render (used to determine if the effect needs to re-run). This field is the primary storage that persists across re-renders. |
| `queue` | `Queue \| null` | For state hooks (`useState`), this is a queue of pending state updates (functions or values) that have been dispatched via the `setState` function but not yet processed. The queue is processed during the render phase to compute the next state value. For effect hooks, this is `null`. |
| `next` | `HookInstance \| null` | Pointer to the **next hook instance** in the linked list. This forms the chain of hooks for a single function component. The list order is critical—it must match the order of hook calls during every render. |
| `cleanup` | `function \| null` | For effect hooks (`useEffect`), this stores the **cleanup function** returned by the effect function from the previous render. This cleanup is invoked before the next execution of the effect (or when the component unmounts). For state hooks, this is `null`. |

> **Design Insight:** The linked list structure for hooks is chosen over an array because it aligns with the overall Fiber architecture's preference for linked structures (easy insertion/deletion in the middle, though we don't use that here) and because it naturally models a sequence of possibly heterogeneous objects (state hooks, effect hooks, custom hooks). However, the most important reason is that a linked list can be built incrementally as we traverse the component tree, without needing to pre-allocate space.

**Hook List Visualization:** A function component that calls `useState(''), useEffect(() => {}, [])`, then `useState(0)` would have a hook list: first a state hook with `memoizedState: ''`, then an effect hook with `memoizedState: []` (dependency array), then a second state hook with `memoizedState: 0`. The `next` pointers chain them together. During the next render, the first call to `useState` receives the value from the first hook instance, the `useEffect` call receives the second, and so on.




**Architecture Decision Record: Hook Storage (Linked List vs. Array)**

> **Decision: Use a Linked List per Fiber for Hook Storage**

- **Context:** Function components need to maintain persistent state across re-renders. We must store this state somewhere associated with the component instance (the Fiber). The storage must support multiple hooks per component, preserve hook order, and allow efficient access during render.
- **Options Considered:**
  1. **Array per Fiber:** Store hooks in an array attached to the fiber. Each hook occupies an index. On each render, a cursor increments to read the next hook.
  2. **Linked List per Fiber:** Store hooks as a linked list, with each hook having a `next` pointer. The fiber points to the head of the list.
- **Decision:** Use a linked list per fiber.
- **Rationale:**
  - **Alignment with Fiber Architecture:** The entire Fiber tree is built on linked lists (`child`, `sibling`, `effect` list). Consistency in data structures reduces cognitive load and allows reuse of traversal utilities.
  - **Dynamic Growth:** While hooks are not dynamically inserted/deleted in order, a linked list naturally supports incremental construction during the render phase without predefining a size (unlike an array which may need resizing).
  - **Memory Efficiency:** For components with many hooks, a linked list avoids the contiguous memory allocation and potential overallocation of dynamic arrays. Each hook is allocated exactly once and remains in place across renders.
- **Consequences:**
  - **Positive:** Unified mental model with the rest of the Fiber system; efficient for the "append-only" pattern of hooks during render.
  - **Negative:** Slightly more complex implementation (pointer management) than a simple array and cursor. Debugging requires following pointers rather than inspecting an array index.

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| Array per Fiber | Simple implementation, easy to debug (visible indices). | Requires managing a cursor variable that must be reset per component; may need resizing; less aligned with overall architecture. | ❌ |
| Linked List per Fiber | Consistent with Fiber's linked structures; memory efficient; natural incremental construction. | More complex to implement and debug (pointer chasing). | ✅ |

### 4.4 The Work-In-Progress Tree

**Mental Model: The Construction Site** When an architect decides to renovate a house, they don't tear down the existing structure immediately. Instead, they set up a **construction site** where the new plans are gradually assembled alongside the old building. The **work-in-progress tree** is exactly that: a parallel Fiber tree being constructed during the render phase, representing the new UI state. Each fiber in this tree has an `alternate` pointer linking back to the corresponding fiber in the **current tree** (the last rendered UI). This dual-tree approach allows us to compare old and new, compute differences, and prepare mutations—all without disturbing the live DOM until we're ready.

The work-in-progress tree is not a separate data structure but a specific **state of the Fiber nodes** during the render phase. It is defined by the following key pointers and state variables:

| Concept | Type | Description |
|---|---|---|
| `workInProgress` (Global variable) | `Fiber \| null` | A pointer to the **currently being processed** fiber in the work-in-progress tree. This is the fiber the reconciliation algorithm is currently examining. It moves through the tree as the work loop progresses. |
| `currentTree` (Global variable) | `Fiber \| null` | A pointer to the root of the **last committed** Fiber tree (the "current" tree displayed in the DOM). This tree is what the user currently sees. |
| `nextUnitOfWork` (Global variable) | `Fiber \| null` | The next fiber that the work loop should process. When the work loop yields (pauses), this variable remembers where to resume. |
| `pendingCommit` (Global variable) | `Fiber \| null` | After the render phase completes for an update, this points to the root of the **finished work-in-progress tree** that is ready to be committed to the DOM. The commit phase then takes this tree and applies all the recorded effects. |

The relationship between these pointers and the two trees is crucial. During the render phase, we traverse the work-in-progress tree, comparing each fiber with its `alternate` (from the current tree). We compute effect tags and build the list of fibers with effects (`nextEffect`). Once traversal finishes, the work-in-progress tree becomes the `pendingCommit`. After the commit phase applies all DOM mutations, the work-in-progress tree becomes the new current tree, and the old current tree is discarded (or recycled as the next work-in-progress tree via `alternate` pointers).

> **Key Insight:** This dual-tree model with `alternate` pointers is what enables **concurrent rendering**. Because we keep the old tree intact while building the new one, we can abort the work-in-progress tree at any time (e.g., if a higher-priority update comes in) and revert to showing the current tree without any visual glitch. The `alternate` pointers also serve as a pool for reusing fiber objects, improving performance by reducing garbage collection.

### 4.5 Summary of Relationships

To crystallize how these data models interconnect:

1. **VNodes** are created by `createElement` (or JSX) and represent the **desired UI state** in a simple tree.
2. **Fibers** are created from VNodes and form a **workable, traversable linked list** that includes reconciliation metadata and links to the real DOM.

3. **Hook instances** are attached to function component fibers via `memoizedState` and store **component-local state and side effects**.
4. The **work-in-progress tree** is the Fiber tree currently being computed, linked to the current tree via `alternate` pointers.

This layered data model transforms the declarative developer input into an efficient, interruptible, and stateful execution plan.

## Implementation Guidance

**Technology Recommendations Table:**

| Component | Simple Option | Advanced Option |
|---|---|---|
| Data Structures | Plain JavaScript objects with explicit property assignment. | Use JavaScript classes or factory functions with closures for encapsulation. |
| Linked List Management | Manual pointer assignment ( `hook.next = nextHook` ). | Create reusable linked list utilities (e.g., `createLinkedListNode` ). |
| Internal Fields | Use underscore prefix ( `_dom` ) to denote privacy. | Use ES2022 private fields ( `#dom` ) for true encapsulation (requires modern environment). |

**Recommended File/Module Structure:**

```
build-your-own-react/
├── src/
│   ├── core/
│   │   ├── vnode.js          # VNode type definition and createElement
│   │   └── constants.js      # EFFECT_TAGS, FIBER_TYPES, DOM_PROPERTIES
│   ├── reconciler/
│   │   ├── fiber.js          # Fiber node definition and creation utilities
│   │   └── hooks.js          # HookInstance definition and hook utilities
│   └── index.js              # Public API (createElement, render, useState, useEffect)
```

**Infrastructure Starter Code:**

`src/core/constants.js` – Complete foundation constants.

JAVASCRIPT

```javascript
// src/core/constants.js

/**
 * Flags for fiber effect tags (bitmask can be used for multiple effects).
 * Use bitwise OR to combine (e.g., PLACEMENT | UPDATE).
 */
export const EFFECT_TAGS = {
  PLACEMENT: 1 << 0,           // Fiber needs to be inserted into the DOM
  UPDATE: 1 << 1,              // Fiber's props/state changed, needs DOM update
  DELETION: 1 << 2,            // Fiber needs to be removed from the DOM
  PLACEMENT_AND_UPDATE: (1 << 0) | (1 << 1) // For nodes that are both new and have updates
};

/**
 * Fiber type classification.
 */
export const FIBER_TYPES = {
  HOST: 'HOST',                // HTML/SVG DOM element
  FUNCTION: 'FUNCTION',        // Function component
  CLASS: 'CLASS',              // Class component (optional)
  TEXT: 'TEXT'                 // Text node
};

/**
 * Special property classification for DOM manipulation.
 */
export const DOM_PROPERTIES = {
  // Regex to identify event handler props (starting with 'on' followed by capital letter)
  EVENTS: /^on[A-Z]/,
  // List of property names that should be set as attributes, not object properties
  ATTRIBUTES: ['className', 'htmlFor', 'value', 'checked', 'selected', 'disabled']
};

/**
 * Update types for the hook queue.
```

```
 */
export const UPDATE_TYPES = {
  STATE: 'STATE',
  EFFECT: 'EFFECT'
};
```

`src/core/vnode.js` – VNode creation utility.

```javascript
// src/core/vnode.js
import { DOM_PROPERTIES } from './constants.js';

/**
 * Creates a Virtual Node (VNode) representing a UI element or component.
 * @param {string|Function} type - Element type (e.g., 'div') or function component.
 * @param {object} props - Properties object (includes children as props.children).
 * @param {...any} children - Child elements (strings, numbers, VNodes).
 * @returns {VNode} A virtual node object.
 */
export function createElement(type, props = {}, ...children) {
  // Flatten children and handle primitive values (string, number) by converting to text VNodes.
  // TODO 1: Normalize `children` array: flatten any nested arrays recursively.
  // TODO 2: Transform each child that is a string or number into a VNode with type 'TEXT_ELEMENT'.
  //         The text content should be stored in props.nodeValue.
  // TODO 3: Assign a unique key if provided in props, else default to null.
  // TODO 4: Return the VNode object with type, props, key, and _dom (initially null).
}
```

**Core Logic Skeleton Code:**

`src/reconciler/fiber.js` – Fiber node creation and utilities.

```javascript
// src/reconciler/fiber.js

import { FIBER_TYPES } from '../core/constants.js';


/**
 * Creates a new Fiber node for a given VNode.
 * @param {VNode} vnode - The virtual node to base the fiber on.
 * @param {Fiber} parent - Parent fiber (or null for root).
 * @param {Fiber} alternate - The previous version of this fiber (for reconciliation).
 * @returns {Fiber} A new fiber node.
 */
export function createFiber(vnode, parent = null, alternate = null) {
  // TODO 1: Determine the fiber's tag based on vnode.type:
  //   - If vnode.type is a string: tag = FIBER_TYPES.HOST
  //   - If vnode.type is a function: tag = FIBER_TYPES.FUNCTION
  //   - Special case: if vnode.type === 'TEXT_ELEMENT': tag = FIBER_TYPES.TEXT
  // TODO 2: Initialize the fiber's core identity fields: type, key, props (from vnode).
  // TODO 3: Set up the linked structure pointers: parent, child, sibling (all null initially).
  // TODO 4: Set the alternate pointer.
  // TODO 5: Initialize state fields: stateNode (null), memoizedState (null), updateQueue (null).
  // TODO 6: Initialize effect fields: effectTag (0), nextEffect (null).
  // TODO 7: Set pendingProps and memoizedProps (both initially from vnode.props).
}
```

`src/reconciler/hooks.js` – Hook instance and queue utilities.

```javascript
// src/reconciler/hooks.js


/**
 * Creates a new Hook instance for a state hook.
 * @param {any} initialState - The initial state value.
 * @returns {HookInstance} A new hook instance.
 */
export function createStateHook(initialState) {
  // TODO 1: Create a hook object with memoizedState set to initialState.
  // TODO 2: Initialize an empty queue (array or linked list) for pending updates.
  // TODO 3: Set next to null.
  // TODO 4: Set cleanup to null (state hooks have no cleanup).
}


/**
 * Creates a new Hook instance for an effect hook.
 * @param {function} effectFunction - The effect function.
 * @param {Array} deps - Dependency array.
 * @returns {HookInstance} A new hook instance.
 */
export function createEffectHook(effectFunction, deps) {
  // TODO 1: Create a hook object with memoizedState set to deps (or null if none provided).
  // TODO 2: Set queue to null (effects don't have an update queue like state).
  // TODO 3: Set next to null.
  // TODO 4: Set cleanup to null (will be populated after effect runs).
}
```

**Language-Specific Hints:**

- Use `Object.freeze` on constant objects like `EFFECT_TAGS` to prevent accidental mutation in development.
- For the linked list traversal, prefer `while` loops over recursion to avoid call stack limits and enable interruption.
- Use `Symbol` for internal property keys (e.g., `const INTERNAL = Symbol('internal')`) if you want to hide fields from `JSON.stringify` and casual inspection, but the underscore convention is simpler for education.

## 5.1 Component Design: Virtual DOM and Renderer (Milestone 1)

> **Milestone(s):** Milestone 1: Virtual DOM

## Mental Model: Blueprints and Builders

Think of building a complex structure like a house. You could approach this in two ways. The **imperative approach** would involve directly telling builders: "Take a brick, place it here, add mortar, take another brick, place it next to the first..." This is tedious, error-prone, and hard to visualize. The **declarative approach** involves creating a detailed **blueprint** first—a lightweight, abstract representation of the final structure showing walls, windows, and doors. You then give this blueprint to a **builder** who knows how to interpret it and construct the actual physical house.

In our UI library, **Virtual DOM nodes ( `VNode` s)** are these blueprints. They are simple JavaScript objects that describe what the UI should look like: "There should be a `div` with class `container`, containing a `button` that says 'Click me'." They are cheap to create and manipulate in memory. The **`render` function** is the builder. It takes a `VNode` blueprint and knows how to construct the corresponding real **DOM elements**—the actual bricks and mortar of the web page that the browser can display and the user can interact with.

This separation is powerful. We can create, compare, and modify these lightweight blueprints ( `VNode` s) with JavaScript's full speed. Only when we have the final, correct blueprint do we instruct the builder ( `render` ) to make the (relatively expensive) changes to the real DOM. This is the core of the declarative UI paradigm: **describe your desired UI state, and let the system figure out how to get there efficiently.**

## Interface: `createElement` and `render`

These two functions form the foundational public API of our library, analogous to React's `React.createElement` and `ReactDOM.render`. Their contracts are defined below.

**Table: `createElement` Function Signature**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `createElement` | `type` (string \| Function): The type of element (e.g., `'div'`, a function component).<br>`props` (object \| null): An object containing properties and attributes for the element.<br>`...children` (any): Zero or more child arguments, which can be strings, numbers, other `VNode` s, or arrays thereof. | `VNode` | Creates a virtual DOM node ( `VNode` ), which is a plain JavaScript object representing a node in the UI tree. It normalizes its `children` into a flat array and handles primitive values (strings, numbers) by converting them into text `VNode` s. |

**Table: `render` Function Signature**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `render` | `vnode` ( `VNode` ): The root virtual node of the tree to render.<br>`container` ( `DOMElement` ): A real DOM element (e.g., a `div` ) that will act as the container. The rendered content will be placed inside this container, replacing any existing content. | `void` | Orchestrates the initial mount of a virtual DOM tree. It converts the root `VNode` and all its descendant `VNode` s into real DOM elements and appends the resulting tree into the provided `container` . This function also kicks off the initial reconciliation process in later milestones. |

**Table: `VNode` Data Structure (Milestone 1 Focus)**

| Field Name | Type | Description |
| --- | --- | --- |
| `type` | string \| Function | The node type. A string (e.g., `'div'`, `'span'`) represents a standard HTML element. A Function represents a user-defined component (implemented in later milestones). |
| `props` | object | A dictionary of properties. This includes standard HTML attributes (like `id`, `class`), custom data attributes, and event listeners (which are functions under props like `onClick`). It also contains `children` in a finalized, flat array. |
| `children` | Array | **Note:** While `children` are passed as arguments to `createElement`, they are normalized and stored *inside* the `props` object under the key `children`. This is a common React pattern. The `VNode` itself does not have a direct `children` field in our model; we access them via `vnode.props.children`. |
| `key` | string \| number \| null | A special prop used by the reconciliation algorithm to identify elements in a list across re-renders. It is extracted from `props` and stored at the `VNode` level for easy access during diffing. |
| `_dom` | `DOMElement` \| null | A reference to the actual, real DOM element that corresponds to this virtual node. This is `null` initially and is set by the `render` or reconciliation process. It's crucial for avoiding expensive DOM lookups during updates. |

> **Key Insight:** The `VNode` is a **snapshot descriptor**. It's immutable; you don't modify a `VNode` directly. To update the UI, you create a *new* tree of `VNode`s and let the reconciliation process (Milestone 2) compare it with the old one.

### ADR: JSX Transformation

> **Decision: Use Direct `createElement` Function Calls for Educational Clarity**

**Context:** JSX (JavaScript XML) is a syntax extension that looks like HTML inside JavaScript (e.g., `<div className="header">Hello</div>`). Browsers cannot understand JSX natively. It must be transformed into standard JavaScript before execution. In production React, this is done by a build tool (like Babel) which compiles JSX down to calls to `React.createElement`. We must decide how our library will handle this transformation during development and learning.

**Options Considered:**

1. **Implement a Custom Babel Plugin/Transform:** Create a plugin that transforms JSX in the user's source code to calls to our library's `createElement`.
2. **Use a Pragma Comment and Off-the-Shelf Babel Transform:** Rely on the existing `@babel/plugin-transform-react-jsx` and configure it via a pragma comment (`/** @jsx myCreateElement */`) to target our function.
3. **Bypass JSX Entirely, Use Direct Function Calls:** Have users (or example code) write the JavaScript equivalent of JSX directly: `createElement('div', {className: 'header'}, 'Hello')`.

**Decision:** We recommend **Option 3 (Direct Function Calls)** for the educational implementation of this project. The core learning objectives are centered on the Virtual DOM, reconciliation, and fiber architecture—not on build tool configuration. Writing `createElement` calls directly eliminates a significant tooling hurdle and keeps the focus on the library's core algorithms.

**Rationale:**

- **Simplicity and Focus:** Learners can run code directly in a browser or Node.js with no build step. This reduces cognitive load and environmental setup issues.
- **Transparency:** It makes the connection between the declarative-looking code and the underlying data structure (`VNode`) completely explicit. You can `console.log` the result of `createElement` and see the exact `VNode` object.
- **Easier Debugging:** Stack traces and errors point directly to your `createElement` calls, not to generated code.

**Consequences:**

- **Developer Experience (DX):** The code is more verbose and less readable than JSX. This is a trade-off we accept for educational purity.
- **Realism:** It diverges from the typical React developer experience. Learners should be aware that real React uses JSX with a Babel transform (Option 2).
- **Forward Compatibility:** If a learner later wishes to use JSX, they can easily adopt Option 2 by adding Babel and the pragma `/** @jsx createElement */` to their source files. Our `createElement` API is designed to be compatible with this transform.

**Table: JSX Transformation Options Comparison**

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **1. Custom Babel Plugin** | Highest fidelity to React's internals; deep learning about AST transformation. | High complexity; major distraction from core project goals; requires deep Babel knowledge. | No |
| **2. Pragma + Existing Plugin** | Excellent DX with JSX; standard React practice; minimal library-side work. | Requires learners to set up and configure Babel, which can be a significant initial hurdle. | No (but recommended for post-learning) |
| **3. Direct Function Calls** | Zero build step; ultimate simplicity and transparency; perfect for focusing on core algorithms. | Verbose code; poor DX compared to JSX. | **Yes** (for education) |

## Common Pitfalls

### ⚠️ Pitfall: Forgetting to Create Text Nodes

- **The Mistake:** When a child of a `VNode` is a string or number, a learner might try to directly set it as the `textContent` of the parent DOM element or incorrectly append it as a string.
- **Why It's Wrong:** The DOM API's `appendChild` expects a `Node` object, not a string. Furthermore, treating text as a special case breaks the uniform tree structure. If text is not its own node, you cannot later update, replace, or delete it independently using the generic reconciliation algorithm.
- **The Fix:** Always create a dedicated **DOM Text Node** for string/number children. In our `render` function, we must detect primitive children and call `document.createTextNode(text)` to create a real text node, and our `VNode` representation should mark these nodes with a special type (like `'TEXT_ELEMENT'`).

### ⚠️ Pitfall: Incorrect Event Listener Attachment

- **The Mistake:** Trying to set an event handler via `domElement.onclick = props.onClick` or using the wrong property name case (e.g., `domElement.addEventListener('onClick', ...)`).
- **Why It's Wrong:** DOM property names for event handlers are lowercased (`onclick`), but React uses camelCase (`onClick`) in `props`. Setting a property directly misses React's event delegation and pooling features (which we are not implementing). More critically, during updates, you must cleanly remove the old listener and attach the new one to avoid memory leaks and incorrect behavior.
- **The Fix:** Use a consistent strategy for all props. For event listeners (props starting with `"on"`), attach them using `domElement.addEventListener(eventType, handler)`. Store a reference to the previous handler so it can be removed during updates (`domElement.removeEventListener(eventType, prevHandler)`). The event type is derived by lowercasing the prop name and removing the `"on"` prefix (`onClick` -> `'click'`).

### ⚠️ Pitfall: Not Handling `null`, `undefined`, or `boolean` Children

- **The Mistake:** Not filtering out `null` or `undefined` values from the `children` array, leading to attempts to create elements from `null`.
- **Why It's Wrong:** React renders nothing for `null`, `undefined`, `true`, and `false`. These are valid children used for conditional rendering. If not filtered, they will cause errors in the DOM creation logic.

- **The Fix:** In the `createElement` function, when normalizing `children`, filter out any child that is `null`, `undefined`, or a boolean. A robust implementation will also flatten arrays of children (e.g., `children` from `array.map()`).

## Implementation Guidance

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| **DOM Creation** | Standard `document.createElement` and `document.createTextNode` | Use `document.createElementNS` for SVG support, not required for initial milestone. |
| **Prop Setting** | Direct property assignment and `setAttribute` | Use a helper function `setDOMProperty` that intelligently chooses the right method (property vs attribute) and handles special cases (e.g., `className`, `style` object). |

**B. Recommended File/Module Structure** For Milestone 1, we will start with a simple, monolithic structure. As we add reconciliation and fibers, we will refactor.

```
build-your-own-react/
├── index.js              # Public API: export createElement, render
├── vdom.js               # createElement and VNode logic
├── render.js             # render function and DOM utility helpers
└── demo.js               # Example application to test the implementation
```

**C. Infrastructure Starter Code** We will provide a complete helper function for setting DOM properties, which handles events, attributes, and the special `children` prop.

**File: `render.js` - Starter Utilities**

JAVASCRIPT

```javascript
/**
 * Helper to set a property on a DOM element.
 * Handles event listeners, boolean attributes, and regular properties/attributes.
 * @param {HTMLElement} dom - The target DOM element.
 * @param {string} name - The name of the prop (e.g., 'className', 'onClick').
 * @param {any} value - The value to set.
 */
export function setDOMProperty(dom, name, value) {
  // 1. If the prop is 'children', ignore it. Children are handled separately.
  if (name === 'children') {
    return;
  }

  // 2. Check if it's an event listener (prop name starts with 'on' and has a function value).
  if (name.startsWith('on')) {
    // Extract event name: 'onClick' -> 'click'
    const eventName = name.toLowerCase().substring(2);
    // TODO: For now, just add the event listener.
    // In Milestone 2, we will need to store the previous handler to remove it.
    dom.addEventListener(eventName, value);
    return;
  }

  // 3. Handle special DOM properties that are not set via setAttribute.
  // We'll use a simple mapping. A more robust solution would check if the property exists in the DOM element.
  const specialProps = {
    className: 'class',
    htmlFor: 'for',
    value: 'value',
    checked: 'checked',
    selected: 'selected',
  };

  if (name in specialProps) {
    const attrName = specialProps[name];
```

```
    if (typeof value === 'boolean') {
      // Boolean attribute: set if true, remove if false.
      if (value) {
        dom.setAttribute(attrName, '');
      } else {
        dom.removeAttribute(attrName);
      }
    } else {
      dom.setAttribute(attrName, value);
    }
    return;
  }

  // 4. Fallback: treat as a regular attribute.
  // For boolean attributes, follow the same pattern as above.
  if (typeof value === 'boolean') {
    if (value) {
      dom.setAttribute(name, '');
    } else {
      dom.removeAttribute(name);
    }
  } else {
    dom.setAttribute(name, value);
  }
}
```

**D. Core Logic Skeleton Code**

**File:** `vdom.js` - `createElement`

JAVASCRIPT

```javascript
/**
 * Creates a virtual DOM node (VNode).
 * @param {string|Function} type - Element type (e.g., 'div', a component function).
 * @param {object|null} props - Element properties (including children).
 * @param {...any} children - Child elements (strings, numbers, VNodes).
 * @returns {VNode} A virtual DOM node object.
 */
export function createElement(type, props, ...children) {
  // TODO 1: Normalize the children.
  // Flatten the children array and convert primitive values (string, number) into
  // special VNodes for text. Filter out null, undefined, and booleans.
  // Example: A child that is a string 'hello' becomes:
  // { type: 'TEXT_ELEMENT', props: { nodeValue: 'hello', children: [] } }
  const normalizedChildren = [];

  // TODO 2: Create the VNode object.
  // The object should have: type, props, key, and _dom.
  // Extract the 'key' from props (if present) and store it on the VNode.
  // Ensure children are placed inside props (as props.children).

  // TODO 3: Return the VNode object.
}

// A helper function to create a text VNode.
function createTextElement(text) {
  // TODO: Return a VNode with type 'TEXT_ELEMENT' and props containing nodeValue.
}
```

**File: `render.js` - `render` Function**

JAVASCRIPT

```javascript
import { setDOMProperty } from './render.js';

/**
 * Renders a VNode tree into a real DOM container.
 * @param {VNode} vnode - The root virtual node to render.
 * @param {DOMElement} container - The real DOM container (e.g., a div).
 */
export function render(vnode, container) {
  // TODO 1: Clear the container's existing content.
  // This is the initial mount. In future milestones, we will reconcile here.

  // TODO 2: Call a helper function `createDOM` to convert the VNode into a real DOM element.
  const dom = createDOM(vnode);

  // TODO 3: Append the created DOM element to the container.
}

/**
 * Recursively creates real DOM elements from a VNode.
 * @param {VNode} vnode - The virtual node to convert.
 * @returns {DOMElement} The created DOM element.
 */
function createDOM(vnode) {
  // TODO 1: Handle text VNodes.
  // If vnode.type === 'TEXT_ELEMENT', create and return a text node.
  // Use document.createTextNode(vnode.props.nodeValue).

  // TODO 2: Create the element for non-text VNodes.
  // Use document.createElement(vnode.type).

  // TODO 3: Apply all props to the DOM element.
  // Iterate over vnode.props. For each key-value pair, call setDOMProperty(dom, key, value).
  // Remember: 'children' prop is handled separately in the next step.

  // TODO 4: Recursively create DOM for each child and append it.
  // Iterate over vnode.props.children (array).
  // For each child, call createDOM(child) and then dom.appendChild(childDom).
```

```
  // TODO 5: Store a reference to the DOM element on the VNode (vnode._dom = dom).
  // This is crucial for future updates.

  // TODO 6: Return the created DOM element.
}
```

**E. Language-Specific Hints**

- **`document.createElement` vs `document.createElementNS`:** For now, use `document.createElement`. If you later want SVG support, you'll need to check the `type` or a namespace prop and use `document.createElementNS('http://www.w3.org/2000/svg', type)`.
- **Boolean Attributes:** Setting a boolean attribute like `disabled` to `false` should remove the attribute entirely, not set it to `"false"`. Our `setDOMProperty` helper handles this.
- **Style Prop:** In React, the `style` prop is an object (e.g., `{color: 'red', fontSize: '14px'}`). Our current `setDOMProperty` doesn't handle this. As an extension, you could add a case for `name === 'style'` and iterate over the object to set `dom.style[property] = value`.

**F. Milestone Checkpoint**

1. **Create a test file `demo.js`:**

```javascript
import { createElement, render } from './index.js';

const vnode = createElement(
  'div',
  { id: 'app', className: 'container' },
  createElement('h1', null, 'Hello, World'),
  createElement('p', null, 'This is a paragraph.'),
  createElement('button', { onClick: () => alert('Clicked!') }, 'Click Me')
);

const container = document.getElementById('root');
render(vnode, container);
```

2. **Expected Outcome:** When you open `index.html` (which loads `demo.js`), you should see a page with a header, a paragraph, and a button. Clicking the button should trigger an alert.
3. **Verification:** Use browser Developer Tools to inspect the rendered elements. Check that:
   - The `div` has `id="app"` and `class="container"`.
   - The button has a `click` event listener attached.
   - There are no extra text nodes containing `null` or `undefined`.
4. **Signs of Trouble:**
   - **Nothing appears:** Check the console for errors. Likely, `createDOM` is not returning the element, or `appendChild` is failing.
   - **Event listener doesn't work:** Ensure the prop name is `onClick` (camelCase) and that `setDOMProperty` correctly converts it to `'click'` for `addEventListener`.

- **Text is missing or malformed:** Verify that your `createTextElement` helper is being called for string children and that `nodeValue` is set correctly.

**G. Debugging Tips**

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **"appendChild is not a function" error** | Trying to append a non-DOM node (like a string or `undefined`). | Add a `console.log` inside `createDOM` to see what is being passed as a child. | Ensure all children are converted to `VNode`s and `createDOM` returns a DOM element. Filter out `null`/`undefined` in `createElement`. |
| **Event fires multiple times or not at all** | Event listeners are being attached multiple times on updates (not in Milestone 1) or the wrong event type is used. | Check the event listener attachment logic. Log the event name and handler. | In Milestone 1, this shouldn't happen on initial mount. Ensure `addEventListener` is called with the correct, lowercased event name. |
| **Boolean attribute appears as `"false"` string** | Setting the attribute via `setAttribute` with a boolean value. | Inspect the DOM in dev tools. You'll see `disabled="false"` instead of the attribute being absent. | Update `setDOMProperty` to treat boolean attributes specially: remove the attribute if `value` is `false`. |
| **`className` not rendering as `class`** | Setting the property directly instead of as an attribute. | Inspect the DOM. You'll see `<div className="container">`. | Add a special case for `className` in `setDOMProperty` that maps it to the `class` attribute. |

## 5.2 Component Design: Reconciliation Engine (Milestone 2)

> **Milestone(s):** Milestone 2: Reconciliation (Diffing)

This component is the **heart of the library's performance**. While the Virtual DOM provides a blueprint of the desired UI, the reconciliation engine is the **construction manager** that compares the old and new blueprints and directs the minimal set of physical changes needed to update the real DOM. It ensures that complex UI updates remain fast and efficient, a core promise of the React paradigm.

### Mental Model: Tree Diffing and Surgical Updates

Imagine you're an architect with two blueprints for a house: the current version (as built) and a new version (as desired). Your goal is to update the existing house to match the new blueprint with the **least amount of demolition and reconstruction** possible.

You wouldn't tear down the entire house and rebuild from scratch—that's wasteful and slow. Instead, you walk through each room, comparing the two blueprints:

1. **Room Type Changed?** If the old blueprint shows a bedroom but the new one shows a bathroom in the same location, you must completely replace that room. This is a **type change**.
2. **Room Details Changed?** If both blueprints show a bedroom, but the new one has different window placements or an updated electrical plan, you can **update the existing room** with those specific changes. This is a **props update**.
3. **Number or Order of Rooms Changed?** If an entire new wing is added, you must **construct and attach** a new set of rooms. If rooms are swapped (e.g., the kitchen and living room are exchanged), you need to carefully **detach, move, and reattach** the existing structures, minimizing waste. This is **children reconciliation**, and using unique room labels (`key`s) makes this reordering efficient.

This process of **tree diffing** is analogous to running `git diff` on two directory structures. The algorithm's goal is to compute the **minimum edit distance** between two tree structures and then apply those edits as precise, surgical updates to the real DOM. This transforms a naive $O(n^3)$ problem (comparing every node to every other node) into a practical, linear-time $O(n)$ operation through intelligent heuristics.

### Algorithm: Recursive Tree Diff

The reconciliation algorithm operates on the Fiber tree, not directly on `VNode`s. However, the core logic for comparing two versions of a component's output is a recursive diff applied during the Fiber **render/reconcile phase**. The following steps outline the logic executed for a single `Fiber` node when reconciling its children (the output of a function/class component or a host element's `props.children`).

**Input:** A `Fiber` node (`currentFiber`) representing the currently committed version, and the new `pendingProps` (which contain the new children VNodes from `createElement`). **Output:** A new `child` Fiber subtree linked to `currentFiber` (the work-in-progress tree), with `effectTag`s (e.g., `PLACEMENT`, `UPDATE`, `DELETION`) marked on fibers that require DOM mutations.

The algorithm proceeds as follows:

1. **Normalize Children:** Convert the `pendingProps.children` (which may be a single VNode, an array, or a primitive value) into a flat, normalized array of VNodes for consistent processing.
2. **Iterate and Reconcile:** Loop through the array of new child VNodes (`newChildren`) alongside the linked list of current child fibers (`oldFiber = currentFiber.child`). For each new child VNode: a. **Key Matching:** Attempt to find a matching `oldFiber` with the same `key` (if both have keys) or, as a fallback, the `oldFiber` at the same index. b. **Type Comparison:** * If no matching `oldFiber` is found, this is a **new node**. Create a new Fiber with a `PLACEMENT` `effectTag`. * If a matching `oldFiber` is found but its `type` differs from the new VNode's `type`, a **type change** has occurred. Mark the `oldFiber` for `DELETION` and create a new Fiber with a `PLACEMENT` `effectTag`. * If a matching `oldFiber` is found with the same `type`, we can **reuse** the existing DOM node (`stateNode`). Create a new Fiber (an "alternate") by cloning the `oldFiber`, update its `pendingProps`, and mark it with an `UPDATE` `effectTag` if its props have changed. c. **Link into Tree:** Link the resulting fiber (whether new, reused, or cloned) as a child or sibling in the work-in-progress tree. d. **Move to Next Sibling:** Advance the `oldFiber` pointer to its `sibling` for the next iteration.
3. **Mark Deletions:** After processing all `newChildren`, any remaining `oldFiber` nodes that were not matched must be **deleted**. They are added to a deletion list (often linked via the `nextEffect` pointer) for processing in the commit phase.
4. **Prop Diffing (Update Detection):** For fibers marked with `UPDATE`, a detailed property comparison is needed. This involves: a. Iterating over the new props (`pendingProps`) to set new or changed values (e.g., `className`, `style`, `onClick`). b. Iterating over the old props (`memoizedProps`) to identify and remove props that are no longer present.

The following flowchart visualizes the decision logic for a single fiber node during reconciliation:


### Keyed Reconciliation: A Closer Look

The most complex and critical part is step 2a: matching by `key`. Without keys, the algorithm uses index-based matching, which fails if children are reordered (causing unnecessary re-creations). The keyed algorithm can be summarized in this state table:

| Current State | Event (New Child VNode) | Action Taken | Next State |
|---|---|---|---|
| `oldFiber` exists with matching `key` and `type` | Same key & type in new list | Reuse fiber, mark for `UPDATE` if props changed, move existing DOM node | Advance both `oldFiber` and `newChild` indices |
| `oldFiber` exists with matching `key` but different `type` | Same key, different type | Mark `oldFiber` for `DELETION`, create new fiber with `PLACEMENT` | Advance `newChild` index, keep `oldFiber` for potential reuse later |
| `oldFiber` exists but no `key` match (or keys differ) | New child has a key not found in old list | Create new fiber with `PLACEMENT` | Advance `newChild` index, `oldFiber` is held for later matching |
| No `oldFiber` remaining | New child present | Create new fiber with `PLACEMENT` | Advance `newChild` index |

The algorithm typically uses a `Map` of `key` to `oldFiber` for O(1) lookups, falling back to a linear scan for remaining fibers without keys.

### ADR: Diffing Strategy (Full vs. Heuristic)

**Decision: Heuristic O(n) Diff Based on Type and Key**

- **Context:** We need an algorithm to compare two trees of arbitrary depth and breadth (the Virtual DOM). A perfect, minimal diff (finding the true minimum edit distance) is an NP-hard problem (exponential time). For a dynamic UI that re-renders frequently, we need a practical algorithm that is fast, predictable, and produces good enough results.
- **Options Considered:**
  1. **Full Tree Diff (Levenshtein Distance Adaptation):** Treat the tree as a sequence (via flattening or tree traversal) and apply a dynamic programming algorithm like Levenshtein distance to find the minimal edit script. This guarantees minimal mutations but has $O(n^3)$ time complexity, which is prohibitive for UI trees.
  2. **Heuristic O(n) Diff (React's Model):** Make two assumptions: (a) Elements of different types will produce different trees, and (b) Elements can be given a stable identity via a `key` prop. This allows a fast, single-pass, depth-first comparison. It is not guaranteed to be minimal (e.g., it might miss moving a subtree if keys are not used) but is linear time and works exceptionally well in practice.
- **Decision:** Implement the **Heuristic O(n) Diff**.
- **Rationale:**
  - **Performance is Non-Negotiable:** A UI library must be fast. $O(n^3)$ scaling would make even moderately complex applications unusable.
  - **Practical Sufficiency:** The two heuristics (type and key) map directly to developer intent. Changing a component type *should* destroy its subtree. Giving an element a `key` is a direct signal to the diff algorithm about its identity, enabling efficient reordering.
  - **Alignment with React:** This is the core of React's reconciliation model. Implementing it provides deep educational value into how a widely-used library actually works.
- **Consequences:**
  - **Enables:** Fast, predictable updates for the vast majority of real-world UI patterns (list reordering, conditional rendering, prop updates).
  - **Introduces:** The requirement for developers to add `key` props to dynamic list children for optimal performance. The algorithm will still work without keys but may cause unnecessary DOM churn on reorders.

The following table summarizes the trade-off:

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **Full Tree Diff** | Guarantees minimal DOM operations. Theoretically optimal. | $O(n^3)$ time complexity, making it unusable for real-time UI. Extremely complex to implement. | ❌ |
| **Heuristic O(n) Diff** | Linear O(n) time complexity. Simple, intuitive rules based on `type` and `key`. Proven in production by React. | Not guaranteed to be minimal (e.g., may miss certain subtree moves without keys). Requires discipline from developers to use `key`. | ✅ |

## Common Pitfalls

### ⚠️ Pitfall: Using Array Index as `key`

- **Description:** When rendering a list, it's tempting to use the item's index as its `key` (e.g., `key={index}`), especially if the data doesn't have a natural ID.
- **Why It's Wrong:** If the list items can be reordered, filtered, or have items inserted/removed from anywhere but the end, the index is not a stable identity. After an operation, the same `key` might be associated with a completely different data item. This causes the diff algorithm to incorrectly match old and new fibers, leading to:
  - **State Corruption:** Component state (e.g., input text, focus) will "travel" with the incorrectly matched fiber, appearing on the wrong item.
  - **Inefficient Updates:** The algorithm may decide to update an existing DOM node with new props instead of moving it, causing unnecessary DOM operations and potential bugs (like losing internal DOM state).
- **Fix:** Always use a stable, unique identifier from your data as the `key`. If no such ID exists, consider generating a unique key at the time the data is created.

### ⚠️ Pitfall: Losing DOM Node References During Updates

- **Description:** When a fiber is marked for `UPDATE`, the diff algorithm assumes its associated DOM node (`stateNode`) is still present and valid. If you inadvertently detach or replace that node outside of the reconciliation process (e.g., via direct DOM manipulation or a third-party library), subsequent prop updates will fail or apply to the wrong node.
- **Why It's Wrong:** The fiber's `stateNode` pointer becomes a "dangling reference". Calling `setDOMProperty` or `removeDOMProperty` on it leads to errors or silent failures.
- **Fix:** The library must **own all DOM mutations** for nodes it creates. If external mutations are unavoidable, you must force a re-creation of the subtree (e.g., by changing the component's `key`). During implementation, ensure the `alternate` pointer correctly carries over the `stateNode` from the current tree to the work-in-progress tree for fibers that are being reused.

### ⚠️ Pitfall: Incorrect Event Listener Cleanup

- **Description:** When a prop like `onClick` changes from one function to another, the old event listener must be removed from the DOM element before the new one is added. Simply overwriting the `onclick` property can lead to memory leaks (the old function is not garbage collected) or unexpected behavior (both handlers may fire).
- **Why It's Wrong:** Event listeners are held in memory by the browser. Not removing them causes "phantom" handlers that react to user events even after the component that defined them has updated or unmounted.
- **Fix:** In the `removeDOMProperty` function, specifically check for event props (using the `DOM_PROPERTIES.EVENTS` regex). Store the previous handler function and use the DOM's `removeEventListener` API to detach it before attaching the new handler in `setDOMProperty`.

## Implementation Guidance

This guidance provides the skeleton for the reconciliation logic, which will be integrated into the larger Fiber work loop in Milestone 3. For now, focus on the diffing logic itself.

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
| --- | --- | --- |
| Diff Algorithm | Recursive single-pass with index/key map (O(n)) | Multi-pass with longest common subsequence for keys (better moves) |
| Prop Diffing | Iterate over all new and old props (O(n+m)) | Use a pre-defined list of known props to skip shallow equality checks |
| Key Mapping | JavaScript `Map` object for O(1) lookups | Linear search for small lists (less overhead) |

**B. Recommended File/Module Structure** The reconciliation logic is a core part of the reconciler, which will be housed in its own module.

```
my-react/
  src/
    core/
      index.js              # Public API (createElement, render)
      constants.js          # EFFECT_TAGS, FIBER_TYPES, etc.
    reconciler/
      Reconciler.js         # Main reconciliation logic (diff algorithm)
      ReactDOMComponent.js  # DOM-specific property updates (set/removeDOMProperty)
    hooks/
      ReactHooks.js         # useState, useEffect implementations
    utils/
      domUtils.js           # createDOMElement, setAttribute, etc.
```

**C. Infrastructure Starter Code** The following helper functions are essential for prop diffing and should be implemented fully.

**File:** `src/reconciler/ReactDOMComponent.js`

JAVASCRIPT

```javascript
import { DOM_PROPERTIES } from '../core/constants';

/**
 * Sets a property on a DOM element, handling special cases like event listeners
 * and boolean attributes.
 * @param {DOMElement} domElement - The target DOM element.
 * @param {string} propName - The name of the property (e.g., 'className', 'onClick').
 * @param {any} propValue - The new value for the property.
 */
export function setDOMProperty(domElement, propName, propValue) {
    // Handle event listeners (props starting with 'on')
    if (DOM_PROPERTIES.EVENTS.test(propName)) {
        const eventType = propName.toLowerCase().substring(2); // e.g., 'click'
        // Remove previous listener if it exists (requires storing it)
        const prevListener = domElement._listeners?.[eventType];
        if (prevListener) {
            domElement.removeEventListener(eventType, prevListener);
        }
        // Store and attach new listener
        if (!domElement._listeners) domElement._listeners = {};
        domElement._listeners[eventType] = propValue;
        domElement.addEventListener(eventType, propValue);
        return;
    }

    // Handle special attribute names (e.g., 'className' -> 'class')
    const attributeName = DOM_PROPERTIES.ATTRIBUTES[propName] || propName;

    // Handle boolean attributes (e.g., 'disabled', 'checked')
    if (typeof propValue === 'boolean') {
        if (propValue) {
            domElement.setAttribute(attributeName, '');
        } else {
            domElement.removeAttribute(attributeName);
        }
```

```
        return;
    }

    // Handle regular attributes and properties
    domElement.setAttribute(attributeName, propValue);
}

/**
 * Removes a property from a DOM element.
 * @param {DOMElement} domElement - The target DOM element.
 * @param {string} propName - The name of the property to remove.
 * @param {any} prevValue - The previous value (needed for event listener cleanup).
 */
export function removeDOMProperty(domElement, propName, prevValue) {
    if (DOM_PROPERTIES.EVENTS.test(propName)) {
        const eventType = propName.toLowerCase().substring(2);
        domElement.removeEventListener(eventType, prevValue);
        if (domElement._listeners) {
            delete domElement._listeners[eventType];
        }
        return;
    }

    const attributeName = DOM_PROPERTIES.ATTRIBUTES[propName] || propName;
    domElement.removeAttribute(attributeName);
}
```

**D. Core Logic Skeleton Code** The main reconciliation function is called `reconcileChildren`. It is called for each Fiber whose output (children) needs to be diffed against the previous version.

**File:** `src/reconciler/Reconciler.js`

JAVASCRIPT

```javascript
import { EFFECT_TAGS, FIBER_TYPES } from '../core/constants';
import { createFiber } from './ReactFiber';
import { setDOMProperty, removeDOMProperty } from './ReactDOMComponent';

/**
 * Reconciles the children of a current fiber with new VNode children.
 * This is the core diffing algorithm.
 * @param {Fiber} currentFiber - The current (old) fiber being updated.
 * @param {Array<VNode>|VNode} newChildVNodes - The new children to reconcile against.
 */
export function reconcileChildren(currentFiber, newChildVNodes) {
    // TODO 1: Normalize `newChildVNodes` into a flat array.
    //         If it's a single VNode, wrap it in an array.
    //         If it's a primitive (string, number), create a text VNode for it.

    const newChildren = []; // Placeholder: normalize into this array.

    // TODO 2: Initialize pointers for traversal.
    //         `oldFiber` = currentFiber.child (the first child of the current tree).
    //         `previousNewFiber` = null (will link new fibers as siblings).
    //         `index` = 0 (loop counter).
    //         `keyedOldFibers` = null (will be a Map of key to oldFiber, built lazily).

    // TODO 3: First pass: iterate over newChildren.
    //         For each newChild VNode:
    //           a. Determine its `key` (newChild.key or null).
    //           b. Find a matching `oldFiber` (call `getMatchingOldFiber` helper).
    //           c. Based on match and type comparison, decide action (create, reuse, delete).
    //           d. Call `createReconciledFiber` helper to get the new fiber for this position.
    //           e. Link the new fiber into the work-in-progress tree as a child or sibling.
    //           f. Advance pointers.

    // TODO 4: Second pass: mark remaining old fibers for deletion.
    //         After the loop, any `oldFiber` not matched should have its `effectTag` set to DELETION.
    //         Link them into a deletion list (e.g., currentFiber.deletions).
```

```
    // TODO 5: Helper: `getMatchingOldFiber`.
    //         Builds a Map of key to oldFiber (only once, on first non-index match).
    //         Returns the matching oldFiber or null.

    // TODO 6: Helper: `createReconciledFiber`.
    //         Takes (oldFiber, newChildVNode, effectTag).
    //         If oldFiber exists and type matches, clones it (creating alternate) and updates props.
    //         Otherwise, creates a brand new fiber from the VNode.
    //         Assigns the correct effectTag (PLACEMENT, UPDATE, etc.).
    //         Returns the new fiber.
}

/**
 * Applies the diff results (effectTags) to the DOM during the commit phase.
 * This function is called for each fiber with an effectTag.
 * @param {Fiber} fiber - The fiber containing the effect to commit.
 */
export function commitWork(fiber) {
    // TODO 7: Handle different effectTags.
    //         - PLACEMENT: append the fiber's stateNode to the parent DOM node.
    //         - UPDATE: call `updateDOMProperties` to diff and apply prop changes.
    //         - DELETION: recursively remove the DOM node and cleanup (effects, listeners).
    //         - PLACEMENT_AND_UPDATE: handle both placement and update.
}

/**
 * Diffs the props of a fiber and applies changes to the DOM node.
 * @param {DOMElement} domElement - The DOM element to update.
 * @param {object} prevProps - The previous props (memoizedProps).
 * @param {object} nextProps - The new props (pendingProps).
 */
function updateDOMProperties(domElement, prevProps, nextProps) {
    // TODO 8: Iterate over `nextProps` keys.
    //         For each key, compare `nextProps[key]` with `prevProps[key]`.
    //         If different, call `setDOMProperty(domElement, key, nextProps[key])`.
```

```
    // TODO 9: Iterate over `prevProps` keys.
    //         If a key exists in `prevProps` but not in `nextProps`,
    //         call `removeDOMProperty(domElement, key, prevProps[key])`.
}
```

**E. Language-Specific Hints**

- Use `Map` for `keyedOldFibers` for efficient O(1) lookups. Remember to handle the `null` key case.
- When cloning a fiber to create its alternate, use `Object.assign({}, oldFiber)` to create a shallow copy, then reset the `alternate`, `effectTag`, and `pendingProps` fields.
- Use strict equality (`===`) for prop value comparison in `updateDOMProperties`, but be aware that for object props like `style`, you may need a deeper comparison or treat them as always changed for simplicity.

**F. Milestone Checkpoint** To verify your reconciliation engine is working, create a simple test application that demonstrates efficient updates:

1. **Create a dynamic list:**

```javascript
// In your test HTML file
const app = document.getElementById('app');
let items = ['A', 'B', 'C'];
let nextId = 0;

function renderList() {
    const vnode = createElement('ul', null,
        items.map(item => createElement('li', { key: item.id || item }, item.text || item))
    );
    render(vnode, app);
}

renderList();

// Schedule an update that reorders items
setTimeout(() => {
    items = [items[2], items[0], items[1]]; // Reorder C, A, B
    renderList();
}, 1000);
```

2. **Expected Behavior:** After 1 second, the list should reorder without the `<li>` DOM elements being recreated. You can verify this in the browser's DevTools Elements panel by observing that the actual DOM nodes are moved (their positions change, but their identity/event listeners remain intact).
3. **Signs of Trouble:**

- **The list resets/flashes:** The `<li>` elements are being recreated. Check your key matching logic and ensure you are reusing fibers and their `stateNode`.
- **Event listeners stop working:** Check your `setDOMProperty` and `removeDOMProperty` functions for event handlers. Ensure the previous listener is properly removed and the new one attached.
- **Props don't update:** Verify `updateDOMProperties` is being called for fibers with the `UPDATE` effectTag and that prop diffing correctly identifies changes.

## 5.3 Component Design: Fiber Architecture (Milestone 3)

> **Milestone(s):** Milestone 3: Fiber Architecture

This component is the **scheduling and execution engine** for our React-like library. While the Virtual DOM provides a blueprint and the Reconciliation Engine calculates the differences, the Fiber Architecture determines *when* and *how* those calculations and updates happen. Its primary challenge is enabling smooth, responsive user interfaces by breaking rendering work into incremental units that can be paused, resumed, and prioritized, preventing the main thread from being blocked by large updates.

### Mental Model: Task Lists and Unit-of-Work

Imagine you're a project manager with a massive renovation project (rendering a complex UI). Instead of having your construction crew (the JavaScript engine) work non-stop until the entire project is done—blocking all other activity in the neighborhood (the browser's main thread)—you break the project down into individual tasks (fibers). You create a detailed task list (the fiber tree) where each task knows about its parent task, its child tasks, and its sibling tasks. You (the work loop) then process these tasks one by one, constantly checking if there's time remaining in the workday (the browser's idle periods). If a high-priority emergency arises (user input), you can pause the current task, handle the emergency, and then resume exactly where you left off. Only when all tasks for a phase are complete do you have the crew perform the actual, irreversible construction (the commit phase) all at once, minimizing visible disruption.

In this model:

- A **Fiber** is a single, manageable unit of work representing one component or DOM node.
- The **Work Loop** is the project manager that processes tasks during available time.
- **Scheduling** is the strategy of using the browser's idle periods ( `requestIdleCallback` ) to process work.
- The **Render/Reconciliation Phase** is the planning and blueprint comparison phase (can be interrupted).
- The **Commit Phase** is the final, atomic construction phase where all DOM mutations happen (cannot be interrupted).

This separation of planning and execution is key to achieving non-blocking renders.

### Work Loop and Scheduling

The work loop is the central coordination mechanism. It manages the traversal of the fiber tree, performing work on each fiber unit, and controls the transition between the render and commit phases.

**Phases of Work:**

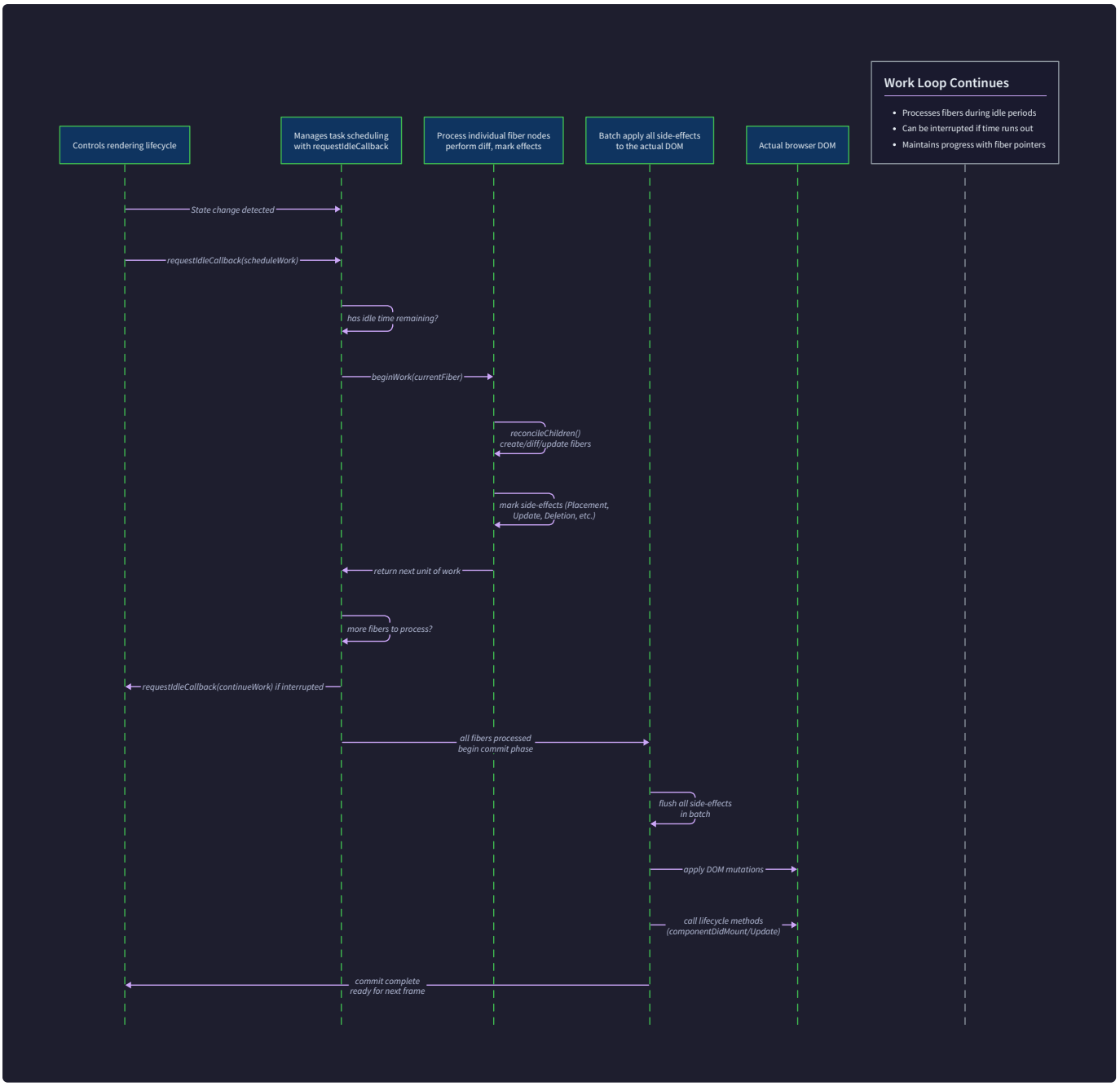
1. **Render/Reconciliation Phase (Interruptible):** This is where the library creates the work-in-progress tree (a copy of the current UI tree with pending changes) and performs the diffing algorithm ( `reconcileChildren` ). The work loop processes fibers in this phase using a **linked list traversal** (child → sibling → uncle) rather than a recursive call stack, making it easy to pause and resume. No DOM mutations occur here.
2. **Commit Phase (Atomic):** After the entire work-in-progress tree is fully reconciled and all changes are calculated, the work loop enters the commit phase. It traverses the list of fibers tagged with effects (like `PLACEMENT` , `UPDATE` , `DELETION` ) and applies all DOM mutations synchronously in a single batch. This phase must not be interrupted to prevent the UI from showing inconsistent, mid-update state.

**Scheduling with requestIdleCallback:** The loop uses the browser's `requestIdleCallback` API to schedule its work. This API allows the browser to run our callback during idle periods, providing a `IdleDeadline` object that tells us how much time we have until the browser needs to take control back for high-priority tasks like animation or input response.

**Core Algorithm Steps:**

1. **Initial Render:** When `render` is first called, it creates a root fiber and sets it as the `nextUnitOfWork`. It then kicks off the work loop.
2. **Work Loop Entry:** The loop runs while there is a `nextUnitOfWork` and the browser's idle deadline hasn't expired.
3. **Perform Unit of Work:** Calls `performUnitOfWork` on the current fiber. This function does three things: a. **Begin Work:** For a host (DOM) fiber, this creates the DOM node (if new). For a function component fiber, it runs the component function, calls its hooks, and returns its child VNode(s). b. **Reconcile Children:** It calls `reconcileChildren` (from Milestone 2) to diff the fiber's children VNodes against its old children (found via the `alternate` pointer) and creates a new linked list of child fibers for the work-in-progress tree. c. **Return Next Unit of Work:** It returns the next fiber to work on using a strict order: first child, then sibling, then parent's sibling (uncle). This depth-first, left-to-right traversal is encoded in the fiber's `child` and `sibling` pointers.
4. **Yield Control:** After processing a fiber, if the `deadline.timeRemaining()` is near zero, the loop pauses, saves the `nextUnitOfWork`, and exits. It will be resumed by another `requestIdleCallback` when the browser is idle again.
5. **Commit Preparation:** Once the `nextUnitOfWork` becomes `null`, it means the entire render phase is complete. The library now has a finished work-in-progress tree with fibers tagged with effects. The loop then proceeds to the commit phase.
6. **Commit Phase:** The loop calls `commitRoot`, which traverses the list of effectful fibers (linked via their `nextEffect` pointer) and applies the DOM mutations by calling `commitWork` for each.
7. **Cleanup and Swap:** After commit, the finished work-in-progress tree becomes the `current` tree (by swapping pointers), ready for the next update cycle.

This process is visualized in the sequence diagram:


Controls rendering lifecycle
Manages task scheduling with requestIdleCallback
Process individual fiber nodes perform diff, mark effects
Batch apply all side-effects to the actual DOM
Actual browser DOM
Work Loop Continues
Processes fibers during idle periods
Can be interrupted if time runs out
Maintains progress with fiber pointers
State change detected
requestIdleCallback(scheduleWork)
has idle time remaining?
beginWork(currentFiber)
reconcileChildren()
create/diff/update fibers
mark side-effects (Placement, Update, Deletion, etc.)
return next unit of work
more fibers to process?
requestIdleCallback(continueWork) if interrupted
all fibers processed
begin commit phase
flush all side-effects in batch
apply DOM mutations
call lifecycle methods (componentDidMount/Update)
commit complete
ready for next frame

### ADR: Scheduling with requestIdleCallback vs. Scheduler

> **Decision: Use `requestIdleCallback` for Simple, Educational Scheduling**
>
> **Context:** We need a way to break rendering work into chunks and yield to the browser to maintain responsiveness. We must choose a scheduling mechanism that fits the educational nature of the project while demonstrating the core principle of cooperative scheduling.
>
> **Options Considered:**
>
> 1. **Native `requestIdleCallback`:** Use the browser's built-in API that provides idle period notifications.
> 2. **Custom Scheduler with `setTimeout` / `MessageChannel`:** Implement a polyfill-like scheduler that simulates yielding using macro-tasks, similar to React's own scheduler package.
> 3. **No Scheduling (Synchronous):** Perform the entire render and commit in one synchronous block, simpler but non-interruptible.
>
> **Decision:** Use `requestIdleCallback` as the primary scheduling mechanism.
>
> **Rationale:**
>
> - **Educational Clarity:** `requestIdleCallback` directly maps to the concept of "working during the browser's idle time." Its API ( `deadline.timeRemaining()` ) is intuitive for checking if we should yield.
> - **Simplicity:** It requires minimal implementation overhead—just a single function call to schedule work. A custom scheduler would introduce significant complexity (task queues, priorities, polyfilling) that distracts from the core fiber architecture concepts.
> - **Adequate Demonstration:** It successfully demonstrates the key architectural principle of **interruptible rendering** and cooperative scheduling, which is the primary learning goal of Milestone 3.
>
> **Consequences:**
>
> - **Browser Support:** `requestIdleCallback` is not available in all browsers (notably Safari). For a production library, a polyfill/custom scheduler is necessary. Our educational implementation may need a simple fallback (like `setTimeout` ) for full compatibility, which we can note in the implementation guidance.
> - **Limited Priority Control:** The native API offers less granular control over task priority compared to a custom scheduler. For our learning purposes, this is an acceptable trade-off.
> - **Simplified Model:** Learners can focus on the fiber tree traversal and phase separation without getting bogged down in scheduler implementation details.

**Option Comparison Table:**

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| `requestIdleCallback` | Native, simple API. Clear conceptual model. Directly demonstrates cooperative scheduling. | Inconsistent browser support (Safari). Less priority control. | **Yes** |
| **Custom Scheduler** | Full control over scheduling and priorities. Consistent cross-browser behavior. Can be more feature-rich. | High complexity. Significant extra code that is tangential to fiber core concepts. | No |
| **Synchronous** | Extremely simple to implement. No scheduling complexity. | Blocks main thread, causing jank. Fails to demonstrate interruptible rendering, a key goal. | No |

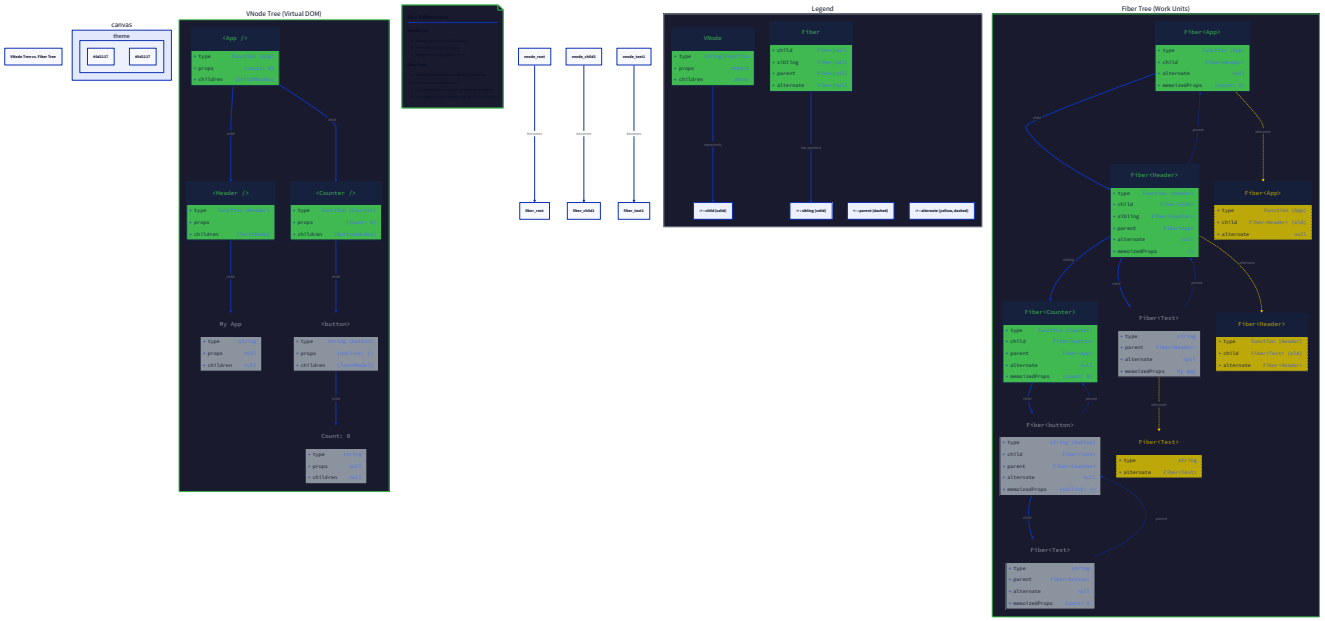
### Fiber Node Structure Deep Dive

A `Fiber` node is an enriched data structure that extends the simple `VNode` . It contains the information needed for reconciliation, linking for traversal, and effect tracking.

**Fiber Node Data Structure:**

| Field Name | Type | Description |
| --- | --- | --- |
| `type` | `string \| Function \| null` | The component type (e.g., `'div'`, a function component). For text nodes, this is `null` (handled by `tag`). |
| `key` | `string \| number \| null` | Unique identifier for child list reconciliation. |
| `props` | `object` | The new props object for this update. Also referred to as `pendingProps`. |
| `stateNode` | `DOMElement \| object \| null` | The associated instance: DOM node for host fibers, component instance for class fibers. |
| `child` | `Fiber \| null` | Pointer to the first child fiber. |
| `sibling` | `Fiber \| null` | Pointer to the next sibling fiber. |
| `parent` | `Fiber \| null` | Pointer to the parent fiber. Also called `return` in React's source. |
| `alternate` | `Fiber \| null` | **Crucial pointer.** Links to the corresponding fiber in the previously committed tree (the "current" tree). This is how we diff old and new. |
| `effectTag` | `number` | A flag indicating what type of change (effect) needs to be applied to the DOM during commit (e.g., `PLACEMENT`, `UPDATE`, `DELETION`). |
| `nextEffect` | `Fiber \| null` | Singly-linked list pointer to the next fiber with an effect. Used to quickly traverse only effectful fibers during commit. |
| `memoizedState` | `HookInstance \| any \| null` | For function components, this points to the linked list of hook instances. For other fibers, it may hold other state (e.g., the final state for a host element). |
| `updateQueue` | `UpdateQueue \| null` | A queue of state updates (for `useState`) or effects (for `useEffect`) that need to be processed. |
| `tag` | `number` | Indicates the fiber type (e.g., `FIBER_TYPES.HOST`, `FIBER_TYPES.FUNCTION`, `FIBER_TYPES.TEXT`). |
| `pendingProps` | `object` | The props passed to this fiber during this render cycle (aliased as `props`). |
| `memoizedProps` | `object` | The props used to create the output during the last render cycle. Used to diff prop changes. |

The relationship between the VNode tree and the Fiber tree is shown in the diagram:




## Common Pitfalls

### ⚠️ Pitfall: Creating Cycles in the Fiber Tree

- **Description:** Incorrectly setting the `parent`, `child`, or `sibling` pointers can create a circular linked list (e.g., setting a fiber's `child` pointer to its own `parent`). This causes infinite loops in the work loop's traversal.
- **Why it's wrong:** The work loop's algorithm for finding the `nextUnitOfWork` relies on a directed acyclic graph (a tree). A cycle means the traversal never terminates, freezing the UI.
- **How to avoid:** Always follow the standard pattern when building the child list in `reconcileChildren`. The parent's `child` points to the first child. Each child's `sibling` points to the next child. Each child's `parent` points back to the parent. Never assign a pointer to an ancestor.

### ⚠️ Pitfall: Incorrectly Managing the `workInProgress` and `alternate` Trees

- **Description:** Losing the reference to the current tree (`alternate` pointers) or incorrectly swapping the trees after commit. This can cause state to be lost, or cause diffing to compare against the wrong old tree.
- **Why it's wrong:** The reconciliation algorithm depends on having access to the old fiber (`fiber.alternate`) to diff against. If this link is broken, every update will be treated as a completely new insertion, destroying and recreating the entire DOM subtree.
- **How to avoid:** When creating a new work-in-progress fiber for a VNode, always attempt to find and reuse its `alternate` from the current tree (by key and type). After the commit phase, systematically swap the `current` root pointer to point to the finished `workInProgress` tree.

### ⚠️ Pitfall: Forgetting to Reset `nextUnitOfWork` After Commit

- **Description:** After the commit phase completes, failing to set `nextUnitOfWork` back to `null`. This causes the work loop to immediately start another render cycle with no pending work, potentially leading to an infinite loop of commits.
- **Why it's wrong:** The work loop condition is `while (nextUnitOfWork && shouldYield())`. If `nextUnitOfWork` is not cleared, the loop will continue trying to process a non-existent unit of work.
- **How to fix:** In the `commitRoot` function, after applying all effects, set the global `nextUnitOfWork` variable to `null`. The next state update (via `setState`) will then schedule a new work loop by setting a new `nextUnitOfWork`.

### ⚠️ Pitfall: Mutating the DOM During the Render Phase

- **Description:** Directly updating a DOM node's properties inside the `performUnitOfWork` function or during reconciliation.
- **Why it's wrong:** The render phase is interruptible. If the browser interrupts after a DOM mutation but before the commit phase finishes, the user could see a partially updated, inconsistent UI. The commit phase is designed to apply all mutations synchronously in

one batch to avoid this.

- **How to avoid:** Never call `setDOMProperty` or any direct DOM manipulation outside of the `commitWork` function. During the render phase, only calculate changes and tag fibers with `effectTag`. The `commitWork` function, called during the non-interruptible commit phase, is the only place where DOM mutations should occur.

## Implementation Guidance

This subsection provides concrete code structure and skeletons to implement the Fiber Architecture.

**A. Technology Recommendations Table**

| Component | Simple Option | Advanced Option |
|---|---|---|
| **Scheduler** | `requestIdleCallback` with `setTimeout` fallback | Custom priority-based task queue using `MessageChannel` |
| **Tree Traversal** | Iterative loop with explicit child/sibling pointers | Recursive with explicit stack management |
| **Effect List** | Singly-linked list via `nextEffect` | Array of effectful fibers for simpler iteration |

**B. Recommended File/Module Structure** Add new files for the fiber core and scheduler. The existing `reconciler.js` from Milestone 2 will be significantly expanded.

```
build-your-own-react/
├── index.js                # Public API (createElement, render)
├── vdom.js                 # createElement, VNode structure (Milestone 1)
├── reconciler.js           # Fiber node, work loop, reconciliation, commit (Milestones 2 & 3)
├── hooks.js               # useState, useEffect implementations (Milestone 4)
├── constants.js           # EFFECT_TAGS, FIBER_TYPES, etc.
└── utils.js               # setDOMProperty, removeDOMProperty, etc.
```

**C. Infrastructure Starter Code** Here is a complete, reusable utility for scheduling work with a `requestIdleCallback` fallback. Place this in `reconciler.js`.

JAVASCRIPT

```javascript
// reconciler.js

// --- Scheduling Utilities ---
const hasIdleCallback = typeof requestIdleCallback !== 'undefined';

/**
 * Schedules a callback to run during the browser's idle periods.
 * Falls back to setTimeout if requestIdleCallback is not available.
 * @param {Function} callback
 */
function scheduleIdleCallback(callback) {
  if (hasIdleCallback) {
    requestIdleCallback(callback);
  } else {
    // Fallback: run as soon as possible but asynchronously
    setTimeout(() => {
      callback({
        timeRemaining: () => 10 // Simulate 10ms of time
      });
    }, 0);
  }
}

/**
 * Simple deadline object for the fallback case.
 */
function createDeadline() {
  return {
    timeRemaining: () => (hasIdleCallback ? 5 : 10) // Simulate some remaining time
  };
}
```

**D. Core Logic Skeleton Code** Below are the key function signatures and detailed TODO comments that map to the algorithm steps described in the prose.

JAVASCRIPT

```javascript
// reconciler.js


// --- Global State for the Renderer ---

let nextUnitOfWork = null;

let currentRoot = null; // Pointer to the root of the last committed fiber tree

let wipRoot = null; // Pointer to the root of the work-in-progress tree

let deletions = []; // Fibers to delete, collected during reconciliation


// --- Fiber Node Creation ---

/**

 * Creates a new Fiber node.

 * @param {VNode} vnode

 * @param {Fiber|null} parent

 * @param {Fiber|null} alternate

 * @returns {Fiber}

 */

function createFiber(vnode, parent, alternate) {

  // TODO 1: Determine the fiber's 'tag' based on vnode.type

  //   - If vnode.type is a string => FIBER_TYPES.HOST

  //   - If vnode.type is a function => FIBER_TYPES.FUNCTION

  //   - If vnode.type is null or undefined? => FIBER_TYPES.TEXT (for text nodes)

  // TODO 2: Create and return a fiber object with all required fields.

  //   - Initialize pointers (child, sibling, parent, alternate) from arguments.

  //   - Set stateNode to null initially.

  //   - Set effectTag to 0 (EFFECT_TAGS.?).

  //   - Set memoizedState and updateQueue to null.

  //   - Set props from vnode.props, and key from vnode.key.

}


// --- Work Loop Core ---

/**

 * The main work loop. Processes units of work until none remain or time runs out.

 * @param {IdleDeadline} deadline

 */

function workLoop(deadline) {
```

```
  // TODO 1: While there is a nextUnitOfWork and time remaining in the deadline...
  //   a. Set nextUnitOfWork = performUnitOfWork(nextUnitOfWork)
  //   b. Check deadline.timeRemaining() and break if <= 0 (yield to browser)
  // TODO 2: If there is NO nextUnitOfWork but we have a wipRoot (render phase done)...
  //   a. Call commitRoot() to apply all changes to the DOM (commit phase)
  // TODO 3: Schedule the next iteration of the work loop if there's still work to do.
  //   Hint: Use scheduleIdleCallback(workLoop) if nextUnitOfWork or wipRoot exists.
}

/**
 * Performs work on a single fiber unit and returns the next fiber to work on.
 * @param {Fiber} fiber
 * @returns {Fiber|null} nextUnitOfWork
 */
function performUnitOfWork(fiber) {
  // TODO 1: Begin work on this fiber. For a FUNCTION component, run the component
  //   function to get its child VNodes. For a HOST component, ensure its DOM node exists.
  //   (This is where hooks for function components are called).
  // TODO 2: Call reconcileChildren(fiber, childVNodes) to diff old and new children
  //   and create the work-in-progress child fiber list.
  // TODO 3: Select and return the next unit of work according to the traversal order:
  //   a. If the fiber has a child, return the child.
  //   b. If no child, but has a sibling, return the sibling.
  //   c. If no child and no sibling, go up to the parent and check its sibling (uncle).
  //   d. Repeat step c until a sibling is found or you reach the root (return null).
}

// --- Commit Phase ---
/**
 * Commits the entire finished work-in-progress tree to the DOM.
 */
function commitRoot() {
  // TODO 1: Process all fibers in the deletions array (call commitDeletion on each).
  // TODO 2: Traverse the effect list starting from wipRoot (or a dedicated firstEffect)
```

```
  //   and call commitWork on each fiber with an effectTag.
  // TODO 3: After all mutations, update currentRoot to point to the wipRoot
  //   (making the finished work-in-progress tree the new current tree).
  // TODO 4: Clean up global state: set wipRoot to null, deletions array empty.
}

/**
 * Applies the DOM changes for a single fiber based on its effectTag.
 * @param {Fiber} fiber
 */
function commitWork(fiber) {
  if (!fiber.parent) {
    // This is the root fiber, no DOM node to attach.
    return;
  }
  const parentDOM = fiber.parent.stateNode;
  // TODO 1: Handle PLACEMENT effect: append fiber.stateNode to parentDOM.
  // TODO 2: Handle UPDATE effect: call updateDOMProperties with old and new props.
  //   (You'll need to get old props from fiber.alternate.memoizedProps).
  // TODO 3: Handle DELETION effect: remove the child from parentDOM.
  //   (Deletions are usually handled in commitDeletion separately).
  // TODO 4: Handle PLACEMENT_AND_UPDATE: do both placement and update.
  // TODO 5: After handling the effect, reset fiber.effectTag to 0.
  // TODO 6: Recursively commit child and sibling effects? (No, the effect list is flat).
}

/**
 * Handles deletion of a fiber and its entire subtree.
 * @param {Fiber} fiber
 */
function commitDeletion(fiber) {
  // TODO 1: If the fiber has a stateNode (DOM node), remove it from its parent.
  // TODO 2: Recursively traverse the fiber's child and sibling tree to call
  //   commitDeletion on each, ensuring all associated DOM nodes are removed.
```

```
  // TODO 3: Run cleanup for any effect hooks in the subtree? (Will be covered in Milestone 4).
}
```

**E. Language-Specific Hints**

- **`requestIdleCallback`:** The callback receives an `IdleDeadline` object. Use `deadline.timeRemaining()` to check how many milliseconds you have left to execute work before yielding. A value > 0 is good.
- **Fallback for Safari:** Our `scheduleIdleCallback` utility provides a simple `setTimeout` fallback. In a real implementation, you'd want a more sophisticated polyfill that mimics idle periods.
- **Traversal Order:** The `performUnitOfWork` function's return logic implements a **depth-first, left-to-right** traversal. Practice this with a small example tree on paper.
- **Effect List:** Building a singly-linked list of effectful fibers (`nextEffect`) during reconciliation optimizes the commit phase. You can skip fibers with no changes entirely during the commit walk.

**F. Milestone Checkpoint Test Application:** Create a simple app that renders a counter and updates it on a button click. **Expected Behavior:**

1. Initial render should correctly display the UI.
2. Clicking the button should trigger a state update.
3. The update should be processed incrementally (you can add `console.log` in `workLoop` to see it yielding).
4. The UI should update to reflect the new counter value without any visible flicker or partial updates (thanks to atomic commit). **Verification:**

- Open your browser's Performance tab and record a click interaction. You should see the JavaScript work broken into multiple small chunks separated by idle periods (or `setTimeout` calls), not one long blocking task.
- Add a `debugger` statement in `performUnitOfWork`. You should be able to step through the fiber tree traversal one unit at a time.

**G. Debugging Tips**

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **UI never updates after state change.** | `nextUnitOfWork` is never set or `workLoop` is not scheduled. | Log inside `workLoop` and the function that sets `nextUnitOfWork`. Check if `scheduleIdleCallback` is being called. | Ensure `setState` triggers a call to schedule work and sets `nextUnitOfWork`. |
| **Infinite loop in console, browser becomes unresponsive.** | Cycle in fiber tree links (`child`, `sibling`, `parent`). | Log fiber links in `performUnitOfWork`. Check if a fiber's `child` or `sibling` points to an ancestor. | Verify `reconcileChildren` correctly links new child fibers without creating cycles. |
| **DOM nodes duplicate on every update.** | `alternate` pointers are lost, so every fiber is treated as new (`PLACEMENT`). | Log `fiber.alternate` during `reconcileChildren`. It should not be null for existing nodes. | Ensure you are reusing the `alternate` fiber when creating new work-in-progress fibers for the same node (same key and type). |
| **State resets to initial value on every render.** | Hook state is not being stored and retrieved from the fiber's `memoizedState`. | Log the `memoizedState` of the function component fiber before and after running the component. | Implement hook state persistence attached to the fiber via the `memoizedState` linked list (Milestone 4). |

## 5.4 Component Design: Hooks System (Milestone 4)

> **Milestone(s):** Milestone 4: Hooks

This component is the **state and lifecycle management system** for our React-like library. While the Fiber architecture provides the physical structure and scheduling, hooks provide the logical memory and side-effect capabilities that make function components truly powerful. This section designs the `useState` and `useEffect` hooks, explaining how they leverage the fiber architecture to persist state across renders and manage side effects.

### Mental Model: Function Component Memory

Imagine a function component as a pure mathematical function: given props, it returns a virtual DOM tree. However, pure functions have no memory—they can't remember values between calls. In our UI library, function components need to "remember" state (like a counter value) and execute side effects (like fetching data) after rendering. This is where hooks come in.

Think of a fiber node representing a function component as having a **personal notebook** attached to it. Every time the component function runs, it can write notes in this notebook or read notes from previous executions. Each hook call (`useState`, `useEffect`) is like writing a new entry in this notebook. The notebook persists across re-renders because it's physically attached to the fiber node in memory, not to the function's local scope which gets discarded after each execution.

The critical insight is that **hooks are not magic—they're a data structure** (a linked list) attached to each fiber, storing the hook's state, queue, and cleanup functions. When a function component runs, the library reads from and writes to this data structure in a consistent order, providing the illusion of persistent local state within a function that technically executes from scratch on every render.

### Interface: useState and useEffect

The hooks system exposes two primary APIs that follow React's familiar patterns. Their exact contracts are defined below:

**Table: Hook Interface Specifications**

| Method | Parameters | Returns | Description |
|---|---|---|---|
| `useState(initialState)` | `initialState`: `any` - The initial state value, or a function that returns the initial state | `[state, setState]`: `[any, function]` - A tuple containing the current state value and a setter function to update it | Declares a state variable that persists between renders. The setter function (`setState`) schedules a re-render of the component with the new state value. The `initialState` is ignored after the first render. |
| `useEffect(effectFunction, dependencyArray)` | `effectFunction`: `function` - A function containing the side effect logic. It may optionally return a cleanup function. `dependencyArray`: `Array<any>` (optional) - An array of values that the effect depends on. If omitted, the effect runs after every render. | `void` | Declares a side effect that runs **after** the component renders and commits to the DOM. The effect runs when the component mounts and again only when any value in the `dependencyArray` changes between renders. If the effect returns a cleanup function, it is invoked before the next effect runs or when the component unmounts. |

**Key Behavioral Details:**

- **Lazy Initialization:** `useState` accepts a function as `initialState`. This function is called only once during the initial render to compute the initial state, useful for expensive computations.
- **State Updates are Batched:** Multiple `setState` calls from the same event handler are batched together, triggering only one re-render. Our implementation will achieve this by scheduling a single reconciliation work unit for the fiber.
- **Effect Timing:** Effects run **after** the commit phase, ensuring the DOM is updated before side effects like measurements or manual DOM manipulations. This matches React's behavior.
- **Dependency Array Comparison:** The `useEffect` hook performs a shallow comparison (using `Object.is` for each element) between the current dependency array and the previous one to decide if the effect should re-run.
- **Cleanup Execution:** Cleanup functions from the previous effect run **before** the new effect runs (if dependencies changed) and also when the component unmounts (during the commit phase when a fiber is deleted).

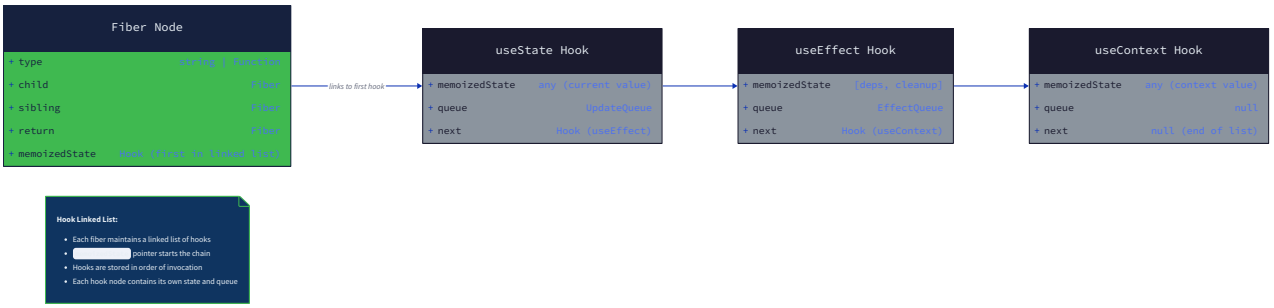
### ADR: Hook Storage (Array vs. Linked List)

> **Decision: Store hooks as a linked list attached to the fiber's `memoizedState` pointer**
>
> - **Context:** Function components can call multiple hooks, and the order of these calls must remain consistent across renders. We need a data structure attached to each fiber to store hook instances that persists between renders. The structure must support sequential access in the same order hooks are called.
> - **Options Considered:**
>   1. **Array per fiber:** Store an array of hook objects in the fiber. Each hook call corresponds to an index in this array.
>   2. **Linked list per fiber:** Store a linked list of hook objects, with the fiber's `memoizedState` pointing to the head node. Each hook has a `next` pointer to the following hook.
> - **Decision:** We will implement a **linked list** structure.
> - **Rationale:** A linked list aligns perfectly with the fiber architecture's existing linked structure (parent, child, sibling). It naturally supports incremental updates: when a component re-renders and calls hooks, we traverse the existing linked list, reusing or updating nodes. Insertions (adding a new hook) are rare and only happen if the component's logic changes, which would require a re-mount anyway. More importantly, a linked list avoids the need to manage array indices and resizing, simplifying the implementation of the hook dispatcher which advances a `currentHook` pointer down the list.
> - **Consequences:**
>   - **Enables:** Simple, pointer-based traversal that mirrors fiber traversal. Easy to attach additional hook metadata (queue, cleanup) as fields in the node.
>   - **Trade-offs:** Slightly more overhead per hook (a `next` pointer) compared to an array index, but this is negligible. The order of hooks is still strictly enforced by the traversal order.

**Table: Hook Storage Options Comparison**

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **Array** | - Simple index-based access (O(1) per hook)<br>- Familiar pattern from React's early implementation | - Requires managing array size and index cursor<br>- Less aligned with fiber's linked structure<br>- Insertion/deletion in middle (though rare) is costly | No |
| **Linked List** | - Natural fit with fiber's linked node concept<br>- Easy incremental traversal with a pointer<br>- Simple to extend with new hook types<br>- No need to pre-allocate or resize | - Sequential access only (O(n) for nth hook)<br>- Slight memory overhead for `next` pointer | **Yes** |

The linked list structure is visualized in the diagram:




## Common Pitfalls

### ⚠️ Pitfall: Stale Closures in Effects

- **Description:** When an effect function captures a state or prop value from its render, but that value changes before the cleanup function runs, the cleanup may reference an outdated ("stale") value. For example, a subscription effect that uses a `userId` prop in its cleanup might use an old `userId` if the prop changed.
- **Why it's wrong:** This can lead to bugs like unsubscribing with the wrong ID, or trying to update state that's no longer relevant. It occurs because the effect function is created during one render and its closure captures the values from that specific render.
- **How to fix:** Our implementation must ensure that the *cleanup function* runs with the closure from the *previous effect*, and the new effect runs with the closure from the *current render*. This is inherently handled by how JavaScript closures work: each effect function is created fresh on each render. The cleanup function stored in the hook instance is the one returned by the *previous effect's function*. By always running the stored cleanup before executing the new effect, we correctly pair cleanups with their corresponding effects.

### ⚠️ Pitfall: Conditional Hooks

- **Description:** Calling hooks inside conditional statements, loops, or nested functions, leading to a different number or order of hook calls between renders.
- **Why it's wrong:** The hook system relies on the invariant that hooks are called in the exact same order every time a component renders. This allows the system to match each hook call with its corresponding stored data in the linked list. Changing the order breaks this association, causing state to "jump" between variables or effects to run incorrectly.
- **How to fix:** Our implementation must detect when the number of hooks called during a render does not match the existing linked list length. We will implement a rule enforcement mechanism: during a function component's render, we track a `hookIndex` (or via pointer traversal). If we run out of hooks (list exhausted) or have extra hooks (list longer than calls), we throw a descriptive error. This mimics React's "Rules of Hooks" error messaging.

### ⚠️ Pitfall: Forgetting Dependency Array in useEffect

- **Description:** Omitting the dependency array argument or providing an empty array when the effect actually depends on props or state, causing the effect to run only once and not when dependencies change.
- **Why it's wrong:** This leads to stale data being used in effects. For example, an effect that fetches data based on a `userId` prop will not re-fetch when `userId` changes if dependencies are missing.
- **How to fix:** While our library cannot automatically detect missing dependencies, we must correctly implement the dependency comparison logic. If no array is provided, the effect runs after every render. If an empty array `[]` is provided, it runs only on mount. We will use shallow comparison ( `Object.is` ) for each element in the array to determine if the effect should re-run. This encourages developers to think about dependencies.

## Implementation Guidance

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
| --- | --- | --- |
| Hook Storage | Linked list of plain JavaScript objects | Same, but with more optimized data structures (e.g., separating state and effect hooks) |
| Dependency Comparison | Shallow comparison using `Object.is` for each element | Custom equality function or deep comparison (not recommended due to performance) |
| Effect Scheduling | Run all effects during the commit phase, after DOM mutations | Priority-based effect scheduling (like React's concurrent features) |

**B. Recommended File/Module Structure**

Our project structure expands to include a dedicated module for hooks logic.

```
build-your-own-react/
├── src/
│   ├── core/
│   │   ├── createElement.js      # Milestone 1
│   │   ├── render.js             # Milestone 1 (initial render)
│   │   └── constants.js          # Shared constants (EFFECT_TAGS, FIBER_TYPES, etc.)
│   ├── reconciler/
│   │   ├── diff.js               # Milestone 2 (reconciliation logic)
│   │   └── patch.js              # Milestone 2 (DOM updates)
│   ├── fiber/
│   │   ├── Fiber.js              # Fiber node class/structure (Milestone 3)
│   │   ├── workLoop.js           # Work loop and scheduler (Milestone 3)
│   │   ├── commit.js             # Commit phase (Milestone 3)
│   │   └── performUnitOfWork.js  # Unit of work processing (Milestone 3)
│   ├── hooks/                    # NEW: Hooks module (Milestone 4)
│   │   ├── hooks.js              # Main hook implementations (useState, useEffect)
│   │   ├── hookDispatcher.js     # Logic to manage the hook linked list during render
│   │   └── hookTypes.js          # Hook instance structures and constants
│   └── index.js                  # Public API (createElement, render)
└── package.json
```

**C. Infrastructure Starter Code**

We provide the complete structure for the hook instance and the dispatcher's state management. This code should be placed in `src/hooks/hookTypes.js` and `src/hooks/hookDispatcher.js`.

**File: `src/hooks/hookTypes.js`**

JAVASCRIPT

```javascript
/**
 * Enum for hook types. Helps in debugging and potential future extensions.
 */
export const HOOK_TYPES = {
  STATE: 'STATE',
  EFFECT: 'EFFECT',
};

/**
 * Base structure for a hook instance in the linked list.
 * @typedef {Object} HookInstance
 * @property {any} memoizedState - The state value for state hooks, or the effect dependencies array for effect hooks.
 * @property {Queue|null} queue - For state hooks: a queue of pending updates. For effect hooks: null.
 * @property {HookInstance|null} next - Pointer to the next hook in the list.
 * @property {function|null} cleanup - For effect hooks: the cleanup function returned by the previous effect. For state hooks: null.
 */

/**
 * An update queue node for state hooks.
 * @typedef {Object} Update
 * @property {any} action - The update function or value.
 * @property {Update|null} next - Next update in the queue.
 */

/**
 * Queue for state updates.
 * @typedef {Object} Queue
 * @property {Update|null} pending - Points to the last update in the circular queue.
 */
```

**File:** `src/hooks/hookDispatcher.js`

JAVASCRIPT

```javascript
import { createStateHook, createEffectHook } from './hooks.js';

// Global variables to track hook state during a fiber's render
let currentlyRenderingFiber = null;
let currentHook = null; // Points to the current hook in the linked list being processed
let nextHook = null;    // Used to advance through the list

/**
 * Sets up the hook dispatcher for a new function component render.
 * Must be called at the beginning of rendering a function component.
 * @param {Fiber} fiber - The fiber representing the function component.
 */
export function beginHookDispatcher(fiber) {
  currentlyRenderingFiber = fiber;
  // Start from the first hook (fiber.memoizedState points to head of linked list)
  currentHook = fiber.memoizedState;
  nextHook = null;
}

/**
 * Advances to the next hook in the linked list.
 * Called after each hook (useState, useEffect) is processed.
 * @throws {Error} If hooks are called conditionally (number mismatch).
 */
function advanceToNextHook() {
  if (currentHook) {
    nextHook = currentHook.next;
    currentHook = nextHook;
  } else {
    // We've run out of hooks, but the component might be calling more hooks than before.
    // This indicates a conditional hook call. We'll handle the error in the hook functions.
    nextHook = null;
    currentHook = null;
  }
}
```

```
/**
 * Gets the hook instance for the current hook call.
 * If no existing hook exists, creates a new one and appends it to the list.
 * @param {string} hookType - One of HOOK_TYPES.
 * @param {any} initialValue - Initial value for the hook.
 * @returns {HookInstance} The hook instance.
 */
function getHookInstance(hookType, initialValue) {
  let hook;
  if (currentHook) {
    // Reuse the existing hook from the previous render
    hook = currentHook;
  } else {
    // This is a new hook call (first render or new hook after conditional)
    hook = createNewHook(initialValue, hookType);
    // Append to the linked list
    if (!currentlyRenderingFiber.memoizedState) {
      currentlyRenderingFiber.memoizedState = hook;
    } else {
      // Find the last hook and append (we could keep a tail pointer, but for simplicity traverse)
      let lastHook = currentlyRenderingFiber.memoizedState;
      while (lastHook.next) {
        lastHook = lastHook.next;
      }
      lastHook.next = hook;
    }
  }
  return hook;
}

function createNewHook(initialValue, hookType) {
  // This is a simplified version. The actual creation is done in the specific hook functions.
  // We'll just return a placeholder; the real implementation is in hooks.js.
```

```
  return null;
}

/**
 * Cleans up the dispatcher after the component render is complete.
 */
export function endHookDispatcher() {
  currentlyRenderingFiber = null;
  currentHook = null;
  nextHook = null;
}
```

**D. Core Logic Skeleton Code**

Now, the core hook implementations. These go in `src/hooks/hooks.js`.

**File:** `src/hooks/hooks.js`

JAVASCRIPT

```javascript
import { scheduleIdleCallback } from '../fiber/workLoop.js';

import { HOOK_TYPES } from './hookTypes.js';


/**

 * Creates a new state hook instance.

 * @param {any} initialState - Initial state value.

 * @returns {HookInstance} A new hook instance for state.

 */

export function createStateHook(initialState) {

  // TODO 1: Create a new hook instance object.

  // TODO 2: Set memoizedState to initialState (if function, call it).

  // TODO 3: Initialize an empty queue for pending updates (queue.pending = null).

  // TODO 4: Set next and cleanup to null.

  // TODO 5: Return the hook instance.

}


/**

 * Creates a new effect hook instance.

 * @param {function} effectFunction - The effect function.

 * @param {Array} deps - Dependency array.

 * @returns {HookInstance} A new hook instance for effects.

 */

export function createEffectHook(effectFunction, deps) {

  // TODO 1: Create a new hook instance object.

  // TODO 2: Set memoizedState to deps (or null if no deps provided).

  // TODO 3: Set queue to null (effects don't have update queues).

  // TODO 4: Store the effectFunction and deps in the instance (you may add extra fields).

  // TODO 5: Set next and cleanup to null.

  // TODO 6: Return the hook instance.

}


/**

 * useState hook implementation.

 * @param {any} initialState - Initial state value or function.

 * @returns {[any, function]} State and setter.
```

```
 */

export function useState(initialState) {

  // TODO 1: Call getHookInstance(HOOK_TYPES.STATE, initialState) to get the hook instance.

  // TODO 2: If this is the first render (hook.queue is empty), set hook.memoizedState to initialState (call if
function).

  // TODO 3: Process any pending updates in hook.queue to compute the current state.

  // TODO 4: Define a setState function that:

  //   a. Creates an update object with the action (could be value or function).

  //   b. Adds the update to the hook's queue (circular linked list).

  //   c. Marks the fiber (currentlyRenderingFiber.alternate or currentlyRenderingFiber) as needing an update.

  //   d. Schedules a new work loop via scheduleIdleCallback.

  // TODO 5: Advance to the next hook (call advanceToNextHook).

  // TODO 6: Return [currentState, setState].

}


/**

 * useEffect hook implementation.

 * @param {function} effectFunction - The effect function.

 * @param {Array} deps - Optional dependency array.

 */

export function useEffect(effectFunction, deps) {

  // TODO 1: Call getHookInstance(HOOK_TYPES.EFFECT, deps) to get the hook instance.

  // TODO 2: Check if dependencies have changed:

  //   a. If no deps provided (argument is undefined), always run.

  //   b. If deps is an array, compare each element with hook.memoizedState using Object.is.

  //   c. If changed or first render, mark the hook as having an effect to run.

  // TODO 3: Store the effectFunction and deps in the hook instance (so we can access them later).

  // TODO 4: During the commit phase, effects will be processed. We need to tag the fiber with an effect.

  //   - Add the fiber to a list of fibers with effects (or set an effectTag on the fiber).

  // TODO 5: Advance to the next hook (call advanceToNextHook).

}
```

**E. Integration with Fiber and Commit Phase**

Hooks must integrate with the existing fiber architecture. The key modifications are:

1. **In `performUnitOfWork`** (when processing a function component fiber): Call `beginHookDispatcher`, invoke the component function (which calls hooks), then call `endHookDispatcher`.
2. **In the commit phase**: After DOM mutations, process effect hooks. We'll add a new function `commitHooks` that runs cleanup functions and then effect functions for fibers tagged with effects.

**F. Language-Specific Hints (JavaScript)**

- Use `Object.is` for dependency comparison (same as React).
- For the state update queue, implement a circular linked list for simplicity. The `queue.pending` points to the last update, and `queue.pending.next` points to the first.
- Store the currently rendering fiber in a module-level variable (like `currentlyRenderingFiber`) as there's only one component rendering at a time in our synchronous work loop.
- When calling the component function to get child VNodes, ensure it's called with the correct `this` context (or use `null`) and pass `props`.

**G. Milestone Checkpoint** After implementing hooks, you should be able to run a test application like this:

```javascript
const { createElement, render, useState, useEffect } = require('./src/index.js');

function Counter() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('hello');

  useEffect(() => {
    console.log('Effect ran: count is', count);
    return () => console.log('Cleanup for count', count);
  }, [count]);

  useEffect(() => {
    console.log('Mount effect only');
    return () => console.log('Unmount cleanup');
  }, []);

  return createElement('div', null,
    createElement('h1', null, `Count: ${count}`),
    createElement('button', { onClick: () => setCount(c => c + 1) }, 'Increment'),
    createElement('input', {
      value: text,
      onInput: (e) => setText(e.target.value)
    }),
    createElement('p', null, `Text: ${text}`)
  );
}

render(createElement(Counter), document.getElementById('root'));
```

**Expected Behavior:**

1. Initial render: Logs "Mount effect only" and "Effect ran: count is 0".
2. Click increment button: Logs "Cleanup for count 0", then "Effect ran: count is 1".
3. Type in input: No effect logs (text dependency not in effect array).
4. Component unmount (if replaced): Logs "Unmount cleanup" and "Cleanup for count [current]".

**Signs of Correct Implementation:**

- State persists between re-renders.
- Effects run after DOM updates (check console order).
- Cleanup runs before the next effect and on unmount.

- Multiple independent state variables work correctly.

**Common Debugging Signs:**

- **State resets to initial value:** The hook linked list is not being reused correctly, or the hook index is reset incorrectly.
- **Effects run on every render even with deps:** Dependency comparison is not working (maybe shallow comparison bug).
- **Cleanup not running:** The cleanup function is not being stored or invoked during commit phase.
- **Infinite loop on setState:** The setState function is not checking if the state actually changed before scheduling work, or scheduling is unconditional.

## 6. Interactions and Data Flow

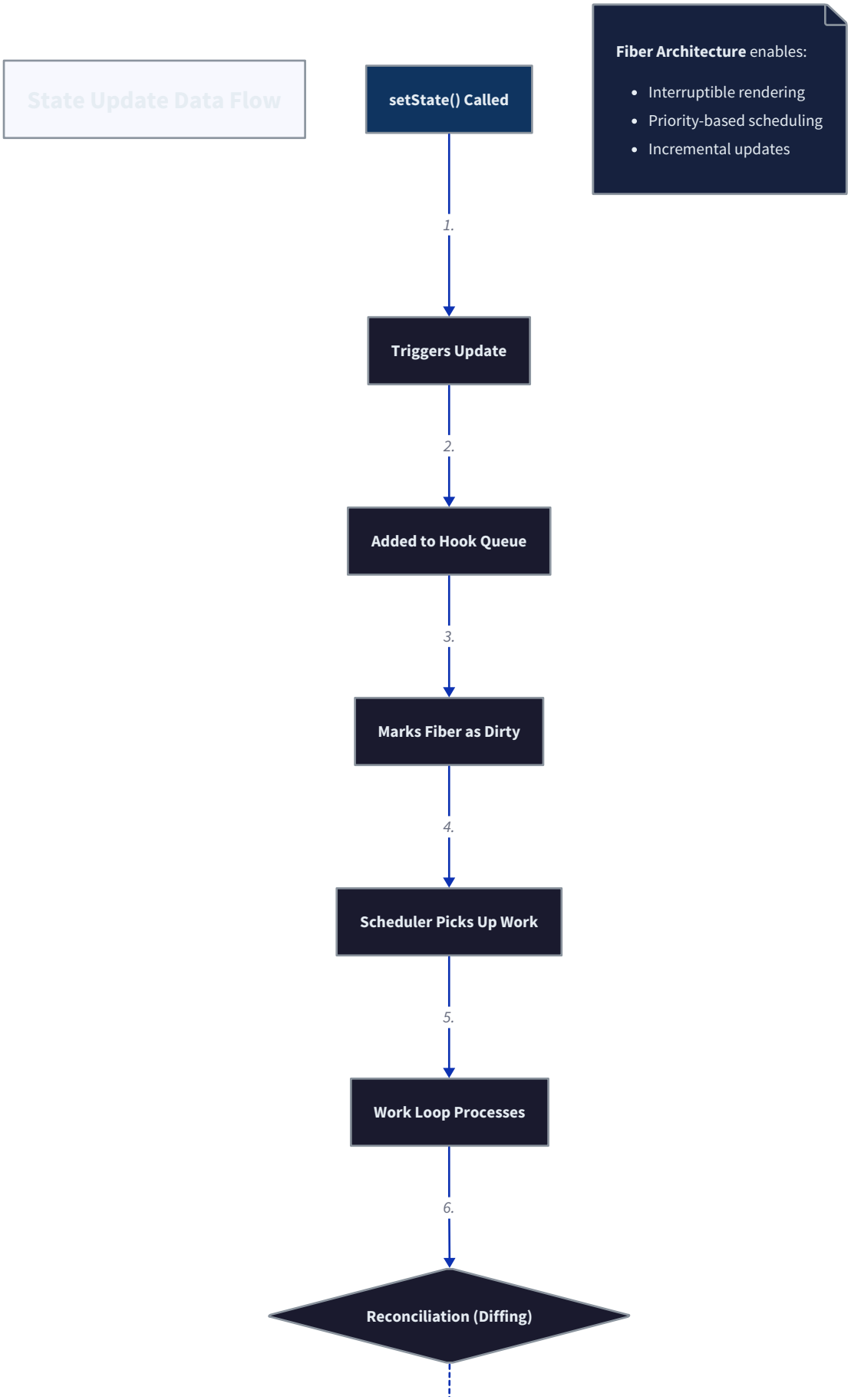
> **Milestone(s):** Milestone 3 (Fiber Architecture), Milestone 4 (Hooks)

This section traces the complete lifecycle of a user interaction—specifically, a state update triggered by a `setState` call—through our library's core systems. Understanding this flow is critical for debugging and for appreciating how the declarative API, the interruptible Fiber scheduler, and the Hooks runtime work together to produce efficient, predictable UI updates. We'll follow two parallel journeys: the main **state update flow** that changes what's on screen, and the **side effect flow** that handles lifecycle operations like data fetching or subscriptions.

The mental model here is a **factory assembly line with quality control**. The initial render is the first production run, building the UI from scratch. A state update is like a customer requesting a modification to the product. The request enters a scheduling system (the work loop), which allocates workers (the reconciliation process) to plan the necessary changes. These workers compare the old design blueprints with the new ones, creating a list of precise modifications. All planned changes are then inspected and applied in a single, coordinated update to the assembly line (the commit phase) to avoid showing a partially finished product. Side effects are like post-assembly tasks (packaging, shipping) that are scheduled to run after the main construction is complete and verified.

### 6.1 Sequence: Triggering a State Update

This subsection details the step-by-step journey from a `setState` call to the corresponding DOM update. The flowchart


State Update Data Flow
setState() Called
Fiber Architecture enables:
Interruptible rendering
Priority-based scheduling
Incremental updates
1.
Triggers Update
2.
Added to Hook Queue
3.
Marks Fiber as Dirty
4.
Scheduler Picks Up Work
5.
Work Loop Processes
6.
Reconciliation (Diffing)


provides a visual companion to this narrative.

### Mental Model: The Modification Request Pipeline

Imagine a publishing house. An author (the component) decides to change a chapter (state). They don't directly call the printers. Instead, they submit a change request ( `setState` ) to an editorial team (the Hook queue). The editorial team marks the book (the Fiber node) as needing revision. A production scheduler (the work loop) picks up the book during the next available planning period. Editors (the reconciler) compare the old and new manuscripts, producing a precise set of edits (effect tags). Finally, in a coordinated overnight print run (the commit phase), all approved edits are applied to the physical books (the DOM) simultaneously, so readers never see a half-updated book.

### Detailed Step-by-Step Walkthrough

Let's trace a concrete example: a counter component with a button. The component has already been rendered once. The user clicks the button, which calls `setCount(count + 1)` .

**Step 1: setState Invocation** The `setState` function returned by `useState` is called with a new value (or an updater function). This `setState` function is a closure that has a reference to the `HookInstance` queue and its associated `Fiber` node.

**Step 2: Update Creation and Enqueueing** The `setState` function creates an `Update` object containing the new state value (e.g., `{ action: 2 }` assuming the previous state was `1` ). It then appends this update to the `pending` queue (a circular linked list) inside the `HookInstance.queue` . The fiber node that hosts this hook is then marked as needing work. In our implementation, this is done by setting the fiber's `updateQueue` or by adding it to a global dirty fibers list for the scheduler to pick up.

> **Design Insight:** The update is queued, not applied immediately. This batching allows multiple `setState` calls from the same event (e.g., within a single event handler) to be processed together in a single render pass, improving performance.

**Step 3: Scheduling Work** The library's scheduler (powered by `requestIdleCallback` via `scheduleIdleCallback` ) is notified that there is pending work. The scheduler will invoke the `workLoop` function during the browser's next idle period. If a render is already in progress, the newly dirtied fiber will be incorporated into the ongoing work-in-progress tree.

**Step 4: Work Loop Activation** The `workLoop` begins its execution. It checks if there is a `nextUnitOfWork` (the root of the work-in-progress tree). If not, it starts from the root of the application (the fiber node associated with the container we originally rendered into). It then enters a loop, calling `performUnitOfWork` on each fiber until either all work is done or the `deadline` (from `requestIdleCallback` ) indicates that time is up. If interrupted, it saves the `nextUnitOfWork` to resume later.

**Step 5: Performing Unit of Work (Reconciliation)** For each fiber, `performUnitOfWork` does two primary things:

1. **Beginning Phase:** For a function component fiber, this calls the component function with its props, which generates a new tree of `VNode` children. During this execution, `useState` hooks are called again. The `getHookInstance` function retrieves the hook from

the fiber's `memoizedState` linked list and applies any pending updates from its queue, calculating the new state value to return. This is how the component gets the updated count.

2. **Reconciliation:** It then calls `reconcileChildren`, which compares the previous list of child fibers (from `currentFiber.alternate.children`) with the new list of child `VNodes`. It creates a new fiber for each child, reusing where possible (same `type` and `key`), and marks fibers with `effectTag`s like `PLACEMENT`, `UPDATE`, or `DELETION`. These effect tags are the "work orders" for the commit phase.

**Step 6: Completing the Work and Preparing for Commit** As `performUnitOfWork` traverses the fiber tree (depth-first, building a list of child-sibling-parent links), it eventually returns `null` when the entire tree has been processed. At this point, the `workLoop` has a complete finished `workInProgress` tree, with all fibers containing their final `effectTag` markings. The root of this tree is assigned to a global variable (often called `pendingCommit` or `workInProgressRoot`). The **render phase** is now complete.

**Step 7: Commit Phase Initiation** The `workLoop` now calls `commitRoot`. This begins the **commit phase**, which is synchronous and cannot be interrupted. It's critical that this phase is fast, as it blocks the main thread and the browser cannot respond to user input until it finishes.

**Step 8: DOM Mutation Execution** `commitRoot` traverses the list of fibers with effects (linked via the `nextEffect` pointer). For each fiber, it calls `commitWork`. This function looks at the fiber's `effectTag`:

- `PLACEMENT`: It calls `appendChild` on the parent DOM node to insert the fiber's `stateNode` (the DOM element).
- `UPDATE`: It calls `updateDOMProperties` to diff the old and new props on the existing DOM element (`fiber.stateNode`), updating attributes, styles, and event listeners.
- `DELETION`: It calls `commitDeletion`, which recursively removes the DOM node and cleans up any associated effects (like `useEffect` cleanup functions).

**Step 9: Post-Commit Cleanup and State Swapping** After all DOM mutations are applied, the library swaps the `current` tree with the finished `workInProgress` tree. The `alternate` pointers are updated so that the next render will compare against this newly committed tree. Any internal bookkeeping (like resetting the `nextUnitOfWork` and the `workInProgressRoot`) is performed. The browser now re-paints the screen, showing the updated count.

**Step 10: Post-Commit Synchronous Effects (Optional, depends on timing)** If any `useEffect` callbacks were scheduled during the commit phase (see next subsection), they are typically invoked **after** the browser has painted, often using a `setTimeout` or `Promise.resolve().then()` to avoid blocking paint. This ensures the screen updates as quickly as possible.

The following table summarizes the key stages, the primary actor (function), and the data transformed at each step:

| Step | Stage | Primary Function/Component | Input | Output | Key Data Structure Mutated |
|---|---|---|---|---|---|
| 1 | Update Trigger | User event handler | New state value | `setState` call | - |
| 2 | Update Queuing | `setState` closure | Update action | Appended `Update` to queue | `HookInstance.queue.pending` |
| 3 | Work Scheduling | `scheduleIdleCallback` | Callback ( `workLoop` ) | Scheduled idle task | Global scheduler state |
| 4 | Work Loop | `workLoop` | `deadline` | `nextUnitOfWork` or `null` | Global `nextUnitOfWork` |
| 5 | Unit of Work | `performUnitOfWork` | A `Fiber` | Next `Fiber` to work on | Fiber's child/sibling links, `effectTag` |
| 6 | State Calculation | `getHookInstance` | Pending updates | New state value | `HookInstance.memoizedState` |
| 7 | Child Reconciliation | `reconcileChildren` | Old fiber children, new VNodes | New child fibers | Child fibers with `effectTag` , `stateNode` |
| 8 | Commit Start | `commitRoot` | Finished work-in-progress root | - | Global `pendingCommit` (set to null) |
| 9 | DOM Mutation | `commitWork` | Fiber with `effectTag` | Mutated DOM | Actual browser DOM tree |
| 10 | Tree Swapping | `commitRoot` (post-mutation) | `workInProgress` tree | New `current` tree | Root's `current` pointer, fiber `alternate` links |

## 6.2 Flow: Side Effect Scheduling and Cleanup

Side effects, managed by `useEffect` , have a lifecycle that is tightly coupled to the render and commit phases but executes primarily *after* the DOM has been updated. This ensures effects operate on the current DOM and can perform operations like data fetching without blocking paint.

### Mental Model: Post-Construction Inspections and Cleanup

Returning to the factory analogy, after the assembly line updates a product (commit phase), a quality assurance team (the effect system) performs additional tasks. These might be attaching a warranty card (logging), sending a shipping notification (data fetch), or subscribing to a product update service. If the product is later modified or discarded, the QA team first performs any necessary teardown (canceling the subscription, cleaning up) before starting new tasks for the updated product.

### How Effects are Tagged and Executed

**During Render: Effect Registration** When a function component calls `useEffect` , the following happens during the render phase:

1. The `createEffectHook` function creates a `HookInstance` (or retrieves the existing one) and stores the user's `effectFunction` and `dependencyArray` on it.
2. The fiber is marked as having effects. This is often done by setting a flag on the fiber (e.g., `fiber.effectTag |= HasEffect` ) or by adding the fiber to a list of fibers with effects.
3. The hook's `memoizedState` stores the `dependencyArray` from the previous render (initially `null` or `undefined` ).

**After Commit: Effect Processing** The actual execution and cleanup of effects occurs in the commit phase, but timing is nuanced. Effects are typically categorized by timing:

- **Layout effects:** Run synchronously immediately after DOM mutations, before the browser paints. (We are not implementing `useLayoutEffect`, but the architecture could be extended.)
- **Passive effects:** Run asynchronously after the browser has painted. Our `useEffect` hooks are passive effects.

Our implementation will schedule passive effects to run after commit. Here is the detailed flow:

**Step A: Collecting Effectful Fibers** During the commit phase, after DOM mutations but before swapping the current tree, we traverse the effect list (the same list used for DOM mutations). For each fiber with a `HasEffect` flag, we examine its `memoizedState` hook linked list. For each `HookInstance` of type `EFFECT`, we check if its dependencies have changed.

> **Dependency Change Algorithm:** We compare the previous `dependencyArray` (stored in `hook.memoizedState`) with the new one provided in the current render. If either is `null` / `undefined`, the effect always runs. Otherwise, we do a shallow comparison of each element. If any element is not strictly equal ( `!Object.is` ), dependencies have changed.

**Step B: Scheduling Effect Execution** If dependencies have changed (or it's the first render), we schedule the effect to run. We do **not** run it immediately during commit. Instead:

1. We store the effect function and its cleanup function (from the previous execution) in a queue of effects to run after paint.
2. We update the hook's `memoizedState` to store the new dependency array for the next comparison.

**Step C: Running Cleanup** *Before* running a new effect, we must run the cleanup function from the previous effect (if it exists). The cleanup function is stored on the `HookInstance.cleanup` property. We run all scheduled cleanups before running any new effects. This order ensures a teardown of the old effect before setting up the new one.

**Step D: Executing Effects** After all cleanups are run, we execute the new effect functions. The effect function may optionally return a new cleanup function, which we store on the `HookInstance.cleanup` for the next cycle.

**Step E: Timing and Browser Yield** To ensure effects run after paint, we schedule the execution of steps C and D using a macro-task like `setTimeout` or a micro-task via `Promise.resolve().then()`. Using a micro-task gets them run as soon as possible after the current JavaScript execution context completes (which is after the commit phase), but still before any user interactions or repaints that might be queued. For educational simplicity, we can use `setTimeout` with a delay of 0.

The following table outlines the state transitions for an effect hook across two renders:

| Render Cycle | Hook State (Before Render) | Action During Render | Action During/After Commit | Hook State (After Commit) |
|---|---|---|---|---|
| First Render | `memoizedState: null`, `cleanup: null` | `createEffectHook(fn, deps)`. Flag fiber. | Dependencies "changed" (no prev). Schedule `fn` to run after paint. `fn` runs, returns `cleanupFn`. | `memoizedState: deps`, `cleanup: cleanupFn` |
| Second Render (deps unchanged) | `memoizedState: prevDeps`, `cleanup: cleanupFn` | `createEffectHook(fn, sameDeps)`. Compare deps, no change. Flag fiber? Possibly not. | No dependency change. Do not schedule `fn`. Cleanup not run. | `memoizedState: sameDeps`, `cleanup: cleanupFn` (unchanged) |
| Third Render (deps changed) | `memoizedState: prevDeps`, `cleanup: cleanupFn` | `createEffectHook(fn, newDeps)`. Compare deps, change detected. Flag fiber. | Schedule cleanup of `cleanupFn`. Schedule `fn` to run after paint. `fn` runs, returns `newCleanupFn`. | `memoizedState: newDeps`, `cleanup: newCleanupFn` |
| Component Unmount | `memoizedState: deps`, `cleanup: cleanupFn` | Fiber is deleted during reconciliation. | During `commitDeletion`, run `cleanupFn` if it exists. | Hook instance is discarded. |

### Common Pitfalls in Side Effect Flow

⚠️ **Pitfall: Running Effects Synchronously During Commit Mistake:** Invoking the effect function directly inside `commitWork`. This can block the browser from painting, leading to jank, especially for slow effects like data fetching. **Why it's wrong:** The commit phase should be as fast as possible to get pixels on screen. Slow effects here directly increase the time the user waits to see an update. **Fix:** Schedule effect execution to occur after the commit phase, using `setTimeout` or `Promise.resolve().then()`.

⚠️ **Pitfall: Incorrect Cleanup Timing Mistake:** Running the cleanup function *after* the new effect function, or not running it at all when dependencies change. **Why it's wrong:** This can lead to resource leaks (e.g., subscriptions aren't canceled before new ones are created) or stale data. **Fix:** Always run the previous cleanup (if it exists) **before** executing the new effect. Ensure this happens when dependencies change and when the component unmounts.

⚠️ **Pitfall: Stale Closures in Effects Mistake:** An effect function captures variables from its render closure (like state or props). If the effect runs only once (empty dependency array), it will forever see the initial values, not the updated ones. **Why it's wrong:** This is a fundamental JavaScript closure issue, not a library bug, but it's a common point of confusion. The effect uses the values from the render it was defined in. **Fix:** Include all values the effect uses inside the dependency array, or use a ref for values that should not trigger re-runs but need to be current.

## Implementation Guidance

This section provides the skeleton for the core functions that orchestrate the data flow described above. The primary focus is on the `commitRoot` function (which handles both DOM mutations and effect scheduling) and the `setState` closure.

### Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Effect Scheduling | `setTimeout(fn, 0)` | `queueMicrotask` or `MessageChannel` for earlier execution, or a custom priority queue |
| Dependency Comparison | Shallow comparison with `Object.is` for each element | Memoization with a custom comparison function (like `areHookInputsEqual` from React) |

### Recommended File/Module Structure

Add the following files to manage the commit phase and effect scheduling:

```
src/
  core/
    commit.js          # Contains commitRoot, commitWork, commitDeletion, and effect scheduling logic
    scheduler.js       # Contains workLoop, scheduleIdleCallback, and related scheduling functions
  hooks/
    dispatcher.js      # Functions for beginning/ending hook dispatcher, getHookInstance
    effects.js         # Functions for scheduling and flushing effects
```

### Infrastructure Starter Code (COMPLETE)

**File:** `src/hooks/effects.js`

JAVASCRIPT

```javascript
// A simple queue for effects to run after paint
let pendingEffects = [];

// Schedule a flush of effects. Use setTimeout for simplicity.
export function scheduleEffect(flushEffects) {
  setTimeout(flushEffects, 0);
}

// Flush all pending effects: run cleanups, then effects.
export function flushEffects() {
  // First, run all scheduled cleanups
  pendingEffects.forEach(effect => {
    if (effect.cleanup) {
      try {
        effect.cleanup();
      } catch (error) {
        // Log error but don't break other effects
        console.error('Error in effect cleanup:', error);
      }
    }
  });

  // Then, run all new effects and store their returned cleanups
  const effectsToRun = pendingEffects;
  pendingEffects = [];
  effectsToRun.forEach(effect => {
    try {
      const cleanup = effect.effect();
      if (typeof cleanup === 'function') {
        effect.hook.cleanup = cleanup;
      } else {
        effect.hook.cleanup = null;
      }
    } catch (error) {
      console.error('Error in effect execution:', error);
```

```
      effect.hook.cleanup = null;
    }
  });
}

// Add an effect to the pending queue
export function enqueueEffect(hook, effect, cleanup) {
  pendingEffects.push({ hook, effect, cleanup });
}
```

### Core Logic Skeleton Code

**File:** `src/core/commit.js`

JAVASCRIPT

```javascript
import { updateDOMProperties, removeDOMProperty } from './dom-utils';
import { enqueueEffect, scheduleEffect, flushEffects } from '../hooks/effects';

/**
 * Commits the entire finished work-in-progress tree to the DOM.
 * This is the second phase (commit phase) and runs synchronously.
 */
export function commitRoot(rootFiber) {
  // TODO 1: Prepare a list of effects from the fiber tree.
  //   - Traverse the fiber tree and collect fibers with effectTag !== null.
  //   - Link them together via `nextEffect` for easy traversal.

  // TODO 2: DOM mutation phase: iterate the effect list.
  //   - For each fiber, call `commitWork(fiber)`.
  //   - `commitWork` will handle PLACEMENT, UPDATE, DELETION.

  // TODO 3: After DOM mutations, swap the current tree with the work-in-progress tree.
  //   - Set the container's `_currentFiber` to the finished root.
  //   - Update the `alternate` pointers throughout the tree.

  // TODO 4: Schedule passive effects (useEffect) to run after paint.
  //   - Collect all fibers that have effect hooks with changed dependencies.
  //   - For each changed effect, enqueue it using `enqueueEffect`.
  //   - Call `scheduleEffect(flushEffects)` to run them after a timeout.

  // TODO 5: Reset global work-in-progress state.
  //   - Set `nextUnitOfWork` to null.
  //   - Set `workInProgressRoot` to null.
}

/**
 * Applies the changes recorded in a fiber's effectTag to the DOM.
 */
export function commitWork(fiber) {
  if (!fiber) return;

  // TODO 1: Find the parent DOM node.
```

```
  //   - Traverse up the fiber tree until you find a fiber with a DOM node (`stateNode`).

  // TODO 2: Handle the effectTag.
  //   - If PLACEMENT: append the fiber's `stateNode` to the parent DOM node.
  //   - If UPDATE: call `updateDOMProperties` with old and new props.
  //   - If DELETION: call `commitDeletion` on the fiber.
  //   - If PLACEMENT_AND_UPDATE: handle both placement and update (rare).
}

/**
 * Deletes a fiber and its subtree, running any cleanup (like effect cleanups).
 */
export function commitDeletion(fiber) {
  // TODO 1: If the fiber is a function component, call cleanup on its effect hooks.
  //   - Traverse the hook list and run any stored cleanup functions.

  // TODO 2: Recursively remove child fibers' DOM nodes.
  //   - If the fiber has a DOM node (`stateNode`), remove it from its parent.
  //   - Recursively call `commitDeletion` on child and sibling fibers.

  // TODO 3: Nullify references to avoid memory leaks.
}
```

**File:** `src/hooks/dispatcher.js`

JAVASCRIPT

```javascript
// Global variable tracking the currently rendering fiber
let currentlyRenderingFiber = null;
// Pointer to the current hook being processed in the linked list
let workInProgressHook = null;

/**
 * Sets up the hook dispatcher for a new function component render.
 */
export function beginHookDispatcher(fiber) {
  // TODO 1: Set `currentlyRenderingFiber` to the given fiber.
  // TODO 2: Set `workInProgressHook` to the fiber's `memoizedState` (the first hook).
}

/**
 * Creates or retrieves the next hook instance in the linked list.
 */
export function getHookInstance(hookType, initialValue) {
  // TODO 1: Determine if we are creating a new hook or reusing.
  //   - If `workInProgressHook` is null, this is the first hook for this fiber.
  //   - Else, we are continuing with the existing linked list.

  // TODO 2: For a new hook:
  //   - Create a new HookInstance with `memoizedState` set to `initialValue`.
  //   - Attach it to the fiber's `memoizedState` (if first) or to the previous hook's `next`.

  // TODO 3: For a reused hook:
  //   - For STATE hook: process any pending updates in the queue to compute new state.
  //   - For EFFECT hook: compare dependencies to decide if effect should run.

  // TODO 4: Advance the `workInProgressHook` pointer to the next hook.
  // TODO 5: Return the hook instance.
}

/**
 * Cleans up the dispatcher after component render.
 */
```

```
export function endHookDispatcher() {

  // TODO: Reset `currentlyRenderingFiber` and `workInProgressHook` to null.

}
```

### Language-Specific Hints

- **Effect Scheduling:** Using `setTimeout` with a delay of 0 is the simplest way to schedule a task for the next event loop iteration, after the current call stack clears and the browser has had a chance to paint. For more precise timing, `queueMicrotask` runs before paint but may still block; `MessageChannel` can be used for immediate but non-blocking scheduling.
- **Dependency Comparison:** Use `Object.is` for comparing each dependency item. Remember that `Object.is` is similar to `===` but treats `NaN` as equal and `+0` vs `-0` as different.
- **Closure Safety:** The `setState` function and effect functions are closures. Ensure they capture the correct fiber and hook references by storing them when the hook is first created (during render).

### Milestone Checkpoint

After implementing the commit phase and effect scheduling, you can test with the following application:

```javascript
// test-app.js

import { createElement, render, useState, useEffect } from './src/index.js';


function Counter() {

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

  const [text, setText] = useState('hello');


  useEffect(() => {

    console.log('Effect ran: count is', count);

    return () => console.log('Cleanup for count', count);

  }, [count]);


  useEffect(() => {

    console.log('Effect ran: text is', text);

  }, [text]);


  return createElement('div', null,

    createElement('h1', null, `Count: ${count}`),

    createElement('button', { onClick: () => setCount(c => c + 1) }, 'Increment'),

    createElement('input', {

      value: text,

      onInput: (e) => setText(e.target.value)

    })

  );

}


const container = document.getElementById('root');

render(createElement(Counter), container);
```

**Expected Behavior:**

1. Initial render logs: `"Effect ran: count is 0"` and `"Effect ran: text is hello"`.
2. Clicking "Increment" button should log `"Cleanup for count 0"` then `"Effect ran: count is 1"`. The text effect should not run.
3. Typing in the input should only log `"Effect ran: text is ..."` for the new text. The count effect should not run.

**Signs of Trouble:**

- **Effects run on every render:** Dependency comparison is not working; check shallow comparison logic.
- **Cleanup runs after new effect:** The order of execution in `flushEffects` is wrong.

- **DOM not updating but effects run:** The commit phase may not be correctly mutating the DOM; check `commitWork` for each `effectTag`.

### Debugging Tips

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| State update causes infinite loop | `setState` called unconditionally inside render or effect | Add console logs inside render and effect. Check if they fire repeatedly. | Ensure `setState` is called only in response to events, not during render. |
| Effect runs on every render even with same deps | Dependency array comparison returns false due to new array each render | Log previous and next deps in `getHookInstance`. | The dependency array passed to `useEffect` should be stable (same reference if values are same). In user code, use `useMemo` or `useCallback`. In our lib, we store the array on the hook for comparison. |
| Cleanup function not called on unmount | `commitDeletion` not traversing hook list | Add a log inside `commitDeletion` to see if it's called. Check if fiber's hook list is being accessed. | Ensure `commitDeletion` calls cleanup for each effect hook in the deleted fiber. |
| DOM updates but effect doesn't run | Effect not scheduled because fiber not flagged | Check if fiber's `effectTag` includes `HasEffect`. Check if dependencies changed. | During render, mark fiber with `HasEffect` if dependencies change. During commit, collect such fibers. |

## 7. Error Handling and Edge Cases

> **Milestone(s):** This section covers error handling and edge cases that span across Milestones 1 (Virtual DOM), 2 (Reconciliation), 3 (Fiber Architecture), and Milestone 4 (Hooks).

In any production-grade UI library, graceful handling of errors and edge cases separates robust systems from brittle ones. While our educational implementation prioritizes core concepts over comprehensive error recovery, understanding these failure modes is essential for building reliable applications. This section addresses two categories: conceptual strategies for catastrophic errors (error boundaries) and practical edge cases in daily operations (VNode creation and diffing).

### Strategy: Error Boundaries (Conceptual)

#### Mental Model: Circuit Breakers for Component Trees

Think of error boundaries like electrical circuit breakers in a house. When a short circuit occurs in one room (a component error), the breaker trips to prevent the entire house from losing power (the whole app crashing). The problematic circuit (faulty component subtree) goes dark while the rest of the house continues functioning normally. In React, error boundaries are components that catch JavaScript errors in their child component tree, log those errors, and display a fallback UI instead of crashing.

**Decision: Error Boundaries as Conceptual Extension**

- **Context**: JavaScript errors during rendering (e.g., accessing undefined properties, runtime exceptions in component logic) would normally propagate up and crash the entire application. Production React provides error boundaries to contain these failures.
- **Options Considered**:
  1. **Implement full error boundaries**: Catch errors during `performUnitOfWork`, mark the faulty fiber, and render fallback UI.
  2. **Let errors propagate**: Allow uncaught errors to bubble up, crashing the app but keeping implementation simple.
  3. **Conceptual-only discussion**: Explain the mechanism without implementation, focusing on how our architecture could support it.
- **Decision**: Option 3 - Conceptual discussion only.
- **Rationale**: Error boundaries add significant complexity (try-catch integration with fiber traversal, cleanup of interrupted work, fallback UI reconciliation) that distracts from core learning objectives. The architectural patterns (fiber traversal, effect tagging) we've established provide the necessary foundation for implementing error boundaries later.
- **Consequences**: Our library will crash on uncaught render errors, but learners understand how to extend the architecture. This keeps the codebase focused on fundamental concepts.

**How Our Architecture Supports Future Error Boundary Implementation**

The fiber tree structure and work loop provide natural integration points for error boundaries:

1. **Error Detection**: Wrap `performUnitOfWork` in try-catch during the render phase. When an error occurs, mark the current fiber with an error effect tag.
2. **Error Propagation**: Traverse up the fiber tree (using the `parent` pointer) to find the nearest fiber that represents an error boundary component (identified by a special component type or flag).
3. **Work Interruption**: If an error boundary is found, interrupt normal reconciliation for that subtree. Mark the boundary fiber for re-render with fallback props instead of its problematic children.
4. **Cleanup**: Use the existing effect system to:
   - Call cleanup functions for hooks in the failed subtree (via `commitDeletion`)
   - Detach DOM nodes of the failed subtree
   - Mount the fallback UI (a new reconciliation subtree)
5. **Recovery**: Store error information on the boundary fiber for logging or display in the fallback UI.

**Error Boundary Integration Points Table**

| Integration Point | Purpose in Error Boundary | Implementation Notes |
|---|---|---|
| `performUnitOfWork` try-catch | Catch render errors | Should catch both component execution and hook evaluation errors |
| Fiber `effectTag` system | Mark fibers with errors and recovery actions | Add new effect tags: `CAPTURE_ERROR`, `RENDER_FALLBACK` |
| Parent pointer traversal | Find nearest error boundary | Similar to how React finds the nearest class component with `getDerivedStateFromError` |
| `commitDeletion` | Clean up failed subtree | Already handles hook cleanup and DOM detachment |
| `reconcileChildren` | Replace failed children with fallback | Would need to accept alternative children VNodes from error boundary |

> **Key Insight**: Error boundaries leverage the same fiber traversal and effect system we've built for normal updates. The main difference is the error-handling wrapper and the decision to render alternative content when errors occur.

## Edge Cases in VNode Creation and Diffing

While error boundaries handle catastrophic failures, edge cases in daily operations require careful handling to ensure correct behavior. These are not bugs but rather special cases that our implementation must explicitly address.

### Handling Falsey Children in createElement

**The Problem**: In JSX expressions like `<div>{condition && <span>Text</span>}</div>`, when `condition` is false, React doesn't render anything—not even an empty text node. Similarly, `null`, `undefined`, `false`, and `true` are all valid React children that should be ignored.

**Our Solution**: The `createElement` function must filter and normalize children:

| Child Value | Treatment | Reason |
|---|---|---|
| `null`, `undefined`, `false`, `true` | Filtered out (not included in children array) | These are React's "empty" values that produce no output |
| String or Number | Converted to text VNode with `type: 'TEXT_ELEMENT'` | Raw primitives become text nodes in the DOM |
| Array | Flattened (recursively) | JSX fragments and array.map() produce arrays of children |
| Object (VNode) | Included as-is | Already a valid virtual node |

**Implementation Consideration**: During flattening, we must recursively process nested arrays while preserving keys. The `createElement` function becomes responsible for this normalization before the VNode is created.

### SVG and MathML Namespace Handling

**The Problem**: SVG elements (`<svg>`, `<circle>`, `<path>`) require the SVG XML namespace. Creating them with `document.createElement('svg')` produces HTML elements that don't render correctly. Similarly for MathML elements.

**Our Solution**: Track namespace context during DOM element creation:

1. **Namespace Propagation**: When creating an SVG element, set the namespace URI to `'http://www.w3.org/2000/svg'`. All descendants of an SVG element should inherit this namespace.
2. **Context Tracking**: During `render` (initial mount) and `updateDOMProperties` (updates), we need to know if we're inside an SVG context. We can:
   - Store namespace on the fiber node as additional metadata
   - Check the parent element's namespace when creating children
   - Special-case SVG element types during creation
3. **Mixed Content**: HTML elements inside SVG (except foreignObject) are invalid, but we can keep implementation simple and let the browser handle errors.

**Namespace Detection Table**

| Element Type | Namespace URI | Creation Method |
|---|---|---|
| `'svg'` | `'http://www.w3.org/2000/svg'` | `document.createElementNS(namespace, type)` |
| `'circle'`, `'path'`, etc. (when parent is SVG) | `'http://www.w3.org/2000/svg'` | `document.createElementNS(namespace, type)` |
| All other elements | HTML namespace (default) | `document.createElement(type)` |

### Removing Event Listeners and Other Props

**The Problem**: When props change from `{onClick: handler1}` to `{onClick: handler2}` or to `{}`, we must remove the old event listener to prevent memory leaks and incorrect behavior. The same applies to other special props like `style`, `className`, and custom attributes.

**Our Solution**: The `updateDOMProperties` function must handle three cases for each prop:

1. **Prop Added**: New prop exists, old prop didn't → call `setDOMProperty`
2. **Prop Updated**: Both old and new exist with different values → call `setDOMProperty` with new value
3. **Prop Removed**: Old prop exists, new prop doesn't → call `removeDOMProperty`

**Event Listener Specifics**:

- **Naming Convention**: Event props are identified by the `on` prefix (case-insensitive in our implementation, though React uses camelCase).
- **Cleanup**: `removeDOMProperty` must call `removeEventListener` with the exact same function reference used during addition.
- **Performance**: We should not remove and re-add listeners if the handler function hasn't changed (shallow equality check).

**Event Handler Update Algorithm**:

1. Extract event name from prop name (`'onClick'` → `'click'`)
2. Compare old and new handler functions by reference
3. If different or removed:
   - Remove old listener with `removeEventListener(eventName, oldHandler)`
   - If new handler exists, add it with `addEventListener(eventName, newHandler)`
4. Store the current handler reference on the DOM element for future removal

### Key Misuse and Reconciliation Edge Cases

**The Problem**: Keys enable efficient list reordering, but misusing them causes subtle bugs:

| Misuse Pattern | Symptom | Correct Approach |
|---|---|---|
| Using array index as key | Items reorder incorrectly after insertions/deletions | Use stable IDs from data |
| Duplicate keys | Undefined behavior (multiple elements with same key) | Ensure keys are unique within siblings |
| Changing keys | Element gets recreated instead of moved | Keep keys stable across renders |
| Missing keys in lists | Inefficient re-renders (full recreation) | Add keys for dynamic lists |

**Our Implementation Responsibility**: `reconcileChildren` with keyed reconciliation must:

1. Validate key uniqueness (console.warn on duplicates in development)
2. Handle the case where some children have keys and others don't (mix of keyed and unkeyed)
3. Properly match old fibers by key, not just by index

### Hook Edge Cases

While Milestone 4's Hooks system has its own error handling, several edge cases affect the broader architecture:

**Stale Closures in Timeouts/Intervals**:

When `useEffect` sets a timeout that references state or props, it captures values from the render when the effect ran. If the effect doesn't re-run (empty dependency array), the timeout handler has stale data. Our implementation cannot prevent this but should document the pattern.

**Memory Leaks from Unclean Effects**:

If a component with a subscription effect ( `useEffect(() => { subscribe(); }, [])` ) unmounts without cleanup, the subscription persists. Our `commitDeletion` must call all pending cleanup functions from the fiber's hook list.

**Concurrent Updates with Multiple Setters**:

When two state updates happen in rapid succession (e.g., in a single event handler), our hook queue processes them sequentially. However, if updates happen from different sources (timeout + click), we must ensure state consistency.

### Null/Undefined Component Types

**The Problem**: What happens when `createElement` receives `null` or `undefined` as the type? React treats these as "holes" that render nothing.

**Our Solution**:

- In `createElement` : Convert `null` / `undefined` type to `'NULL_COMPONENT'` (a special symbol)
- During reconciliation: `NULL_COMPONENT` fibers produce no DOM output and skip their children
- In fiber creation: Treat them similar to text nodes but with no DOM representation

### Text Node Edge Cases

**Empty Text Nodes**: Consecutive text nodes in the DOM get merged by browsers. Our diffing algorithm should handle cases where a text node becomes empty string (should be removed) or appears between elements.

**Special Characters**: Text containing `&` , `<` , `>` must be properly escaped to prevent XSS. Using `textContent` (not `innerHTML` ) handles this automatically.

### Edge Cases Handling Summary Table

| Edge Case | Affected Milestone | Detection Method | Handling Strategy |
|---|---|---|---|
| Falsey children | 1 (VNode creation) | Value check in `createElement` | Filter out `null`, `undefined`, `false`, `true` |
| SVG namespace | 1 (Rendering), 2 (Updates) | Check element type and parent | Use `createElementNS` for SVG elements |
| Event listener removal | 2 (Reconciliation) | Compare old/new props in `updateDOMProperties` | Call `removeEventListener` with stored handler |
| Duplicate keys | 2 (Reconciliation) | Build key map in `reconcileChildren` | Warn in development, undefined behavior |
| Conditional hooks | 4 (Hooks) | Hook call count per render | Throw error on mismatch (Rules of Hooks) |
| Effect cleanup | 4 (Hooks), 3 (Commit) | Check `cleanup` field in `HookInstance` | Call before re-running effect or on unmount |
| Concurrent state updates | 4 (Hooks), 3 (Scheduling) | Multiple updates in hook queue | Process in order, batch in same render if possible |
| Null component type | 1 (VNode creation) | Type check in `createElement` | Treat as no-op component |
| Mixed keyed/unkeyed children | 2 (Reconciliation) | Check for key property | Fall back to index-based diffing for unkeyed |

> **Design Principle**: Handle edge cases at the appropriate abstraction level—VNode creation filters falsey values, diffing handles prop changes, and the commit phase cleans up effects. This separation keeps each component focused and testable.

### Common Pitfalls

**⚠️ Pitfall: Forgetting Text Node Normalization**

**The Mistake**: Treating string children as regular VNodes without converting them to text elements, leading to incorrect DOM structure.

**Why It's Wrong**: The DOM API requires explicit text nodes ( `document.createTextNode()` ). Directly setting string as child to an element throws an error.

**The Fix**: In `createElement`, map string/number children to `{ type: 'TEXT_ELEMENT', props: { nodeValue: text } }`.

**⚠️ Pitfall: Incorrect Event Handler Reference Comparison**

**The Mistake**: Comparing event handler functions using deep equality or string conversion instead of reference equality.

**Why It's Wrong**: `() => console.log('click')` !== `() => console.log('click')` (different function instances). Removing/re-adding on every render causes performance issues.

**The Fix**: Store the actual function reference on the DOM element and compare by `===` in `updateDOMProperties`.

**⚠️ Pitfall: Not Escaping Special Characters in Text**

**The Mistake**: Using `innerHTML` or string concatenation to set text content, enabling XSS vulnerabilities.

**Why It's Wrong**: Text like `<script>alert('xss')</script>` would execute as code if improperly handled.

**The Fix**: Always use `textContent` or `nodeValue` for text nodes, which automatically escapes HTML.

**⚠️ Pitfall: Memory Leaks from Unsubscribed Event Listeners**

**The Mistake**: Forgetting to call `removeEventListener` when a component updates or unmounts.

**Why It's Wrong**: Event handlers keep references to components, preventing garbage collection. The handler continues firing even after the component is gone.

**The Fix**: In `removeDOMProperty`, check if prop is an event and call `removeEventListener` with the exact stored handler.

## Implementation Guidance

### Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Error Boundary Simulation | Console warnings only | try-catch in performUnitOfWork with fallback render |
| Namespace Handling | Ignore (HTML only) | Full SVG/MathML support with namespace tracking |
| Prop Diffing | Simple remove/add all | Smart comparison with shallow equality |
| Key Validation | No validation | Development-mode warnings for duplicates |

### Recommended File/Module Structure

Add edge case handling utilities to existing files:

```
src/
  core/
    createElement.js      # Filter falsey children, normalize text nodes
    render.js            # SVG namespace handling
  reconciler/
    diff.js              # updateDOMProperties with event cleanup
    reconcileChildren.js # Key validation warnings
  hooks/
    hookDispatcher.js    # Rules of Hooks enforcement
  shared/
    constants.js         # NULL_COMPONENT symbol, SVG_NAMESPACE constant
    warnings.js          # Development warning utilities
```

### Infrastructure Starter Code: Warning Utilities

JAVASCRIPT

```javascript
// shared/warnings.js

let isDev = true; // Set to false in production

export function warn(message, ...args) {
  if (isDev && console && console.warn) {
    console.warn(`[Build Your Own React] ${message}`, ...args);
  }
}

export function warnOnce(message, key) {
  if (isDev) {
    if (!warnOnce.cache) warnOnce.cache = {};
    if (!warnOnce.cache[key]) {
      warnOnce.cache[key] = true;
      warn(message);
    }
  }
}

// Key validation
export function validateKey(key, fiber) {
  if (key == null) return;

  // Check for duplicate keys among siblings
  let sibling = fiber.parent.child;
  let seenKeys = new Set();

  while (sibling) {
    if (sibling !== fiber && sibling.key === key) {
      warn(
        `Duplicate key "${key}" found. Two children with the same key ` +
        `will cause undefined behavior.`
      );
      return;
```

```
    }
    sibling = sibling.sibling;
  }
}
```

**Core Logic Skeleton Code: Enhanced createElement with Edge Cases**

```javascript
// core/createElement.js

import { TEXT_ELEMENT } from '../shared/constants.js';
import { warn } from '../shared/warnings.js';

export function createElement(type, config, ...args) {
  // TODO 1: Handle null/undefined type by returning a special NULL_COMPONENT VNode
  // TODO 2: Initialize props object, copying config properties (except key and ref)
  // TODO 3: Normalize children: flatten arrays, filter falsey values, convert strings/numbers to text VNodes
  // TODO 4: Special handling for children that are already VNodes (preserve their structure)
  // TODO 5: Extract key from config if present (default to null)
  // TODO 6: Return the VNode object with type, props, children, and key properties
}

function normalizeChild(child) {
  // TODO 1: If child is null, undefined, false, or true, return null (filtered out)
  // TODO 2: If child is string or number, create text VNode:
  //         { type: TEXT_ELEMENT, props: { nodeValue: child.toString() }, children: [] }
  // TODO 3: If child is array, recursively normalize each element and flatten
  // TODO 4: If child is object (assume VNode), return as-is
  // TODO 5: For any other type, warn and return null
}
```

**Core Logic Skeleton Code: updateDOMProperties with Event Cleanup**

```javascript
// reconciler/diff.js

import { DOM_PROPERTIES } from '../shared/constants.js';

export function updateDOMProperties(dom, prevProps, nextProps) {
  // TODO 1: Merge all prop names from both old and new props
  // TODO 2: For each prop name, determine if it's added, updated, or removed
  // TODO 3: Special handling for event listeners (props starting with 'on'):
  //         - Extract event name (remove 'on' prefix, lowercase)
  //         - Compare old and new handler by reference
  //         - If different or removed: remove old listener
  //         - If added or different: add new listener
  //         - Store current handler on dom._listeners for future removal
  // TODO 4: Handle style prop specially (merge objects, not replace)
  // TODO 5: Handle className prop (set as 'class' attribute)
  // TODO 6: For regular attributes, use setAttribute/removeAttribute
  // TODO 7: For DOM properties (like value, checked), set directly on dom object
}

export function removeDOMProperty(dom, propName, prevValue) {
  // TODO 1: If propName is event listener, removeEventListener with stored handler
  // TODO 2: If propName is 'style', remove individual styles or clear entire object
  // TODO 3: If propName is 'className', remove 'class' attribute
  // TODO 4: For regular attributes, removeAttribute
  // TODO 5: For DOM properties, set to default value (usually null or empty string)
}
```

**Language-Specific Hints**

- **Event Listener Storage**: Attach handlers to `dom._listeners = {}` object for easy cleanup. This pattern mirrors React's internal event system.
- **Text Node Optimization**: Use `nodeValue` instead of creating new text nodes when updating text content.
- **Namespace Detection**: Check parent fiber's `stateNode.namespaceURI` when creating elements to determine if SVG context.
- **Key Validation**: Use a `Map` to track seen keys during sibling traversal for O(n) duplicate detection.

**Debugging Tips**

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| Event fires multiple times | Old listener not removed | Check `updateDOMProperties` logs for remove/add sequence | Ensure `removeEventListener` called before `addEventListener` |
| SVG not rendering | Wrong namespace | Inspect DOM element's `namespaceURI` property | Use `createElementNS` for SVG elements |
| State resets on reorder | Missing or duplicate keys | Log keys during `reconcileChildren` | Ensure unique, stable keys for list items |
| Memory grows over time | Uncleaned effects | Count `useEffect` calls vs cleanup calls | Ensure `commitDeletion` calls all pending cleanups |
| Text disappears | Falsey child filtering too aggressive | Log children before/after normalization | Only filter `null`, `undefined`, `false`, `true`—not empty strings |

## 8. Testing Strategy

> **Milestone(s):** All (Milestone 1: Virtual DOM, Milestone 2: Reconciliation, Milestone 3: Fiber Architecture, Milestone 4: Hooks)

Testing a UI library presents unique challenges—we must verify that in-memory data structures (like the `Fiber` tree and hook linked lists) correctly drive visible changes in the browser DOM. Unlike testing pure functions, we must validate the *side effects* of our library: DOM element creation, attribute updates, event listener attachment, and asynchronous effect execution. The strategy must span from isolated unit tests of core algorithms to integration tests that simulate full component lifecycles.

The mental model is **dual verification**: we test both the internal state of our renderer (the virtual `Fiber` tree and hook lists) and the external output (the actual browser DOM). Think of it like testing a car's engine (internal combustion processes) and its movement (external result) simultaneously. We'll use a combination of static assertions about data structures and dynamic assertions about the rendered UI.

### 8.1 Testing Approach and Tools

We adopt a **layered testing pyramid** with three levels:

1. **Unit Tests** for pure functions and algorithms (`createElement`, diffing logic, hook dispatcher).
2. **Integration Tests** for component lifecycles (mount, update, unmount) and user interactions (clicks triggering state updates).
3. **Visual Verification Tests** (manual) to confirm the library works end-to-end in a browser.

For automated testing, we require a DOM environment. Running tests in a real browser is complex and slow for development. Instead, we use **JSDOM**, a headless implementation of the DOM and web standards in Node.js. It allows us to create `document` objects, mount our rendered output, and assert against DOM nodes—all within a Node test runner.

**Decision: Testing Framework and Environment**

- **Context**: We need a fast, reliable testing setup that can simulate browser DOM APIs for unit and integration tests, without requiring a full browser during development.
- **Options Considered**:
  1. **Jest + JSDOM**: Mature test runner with built-in JSDOM support, snapshot testing, and rich assertion library.
  2. **Mocha/Chai + JSDOM manually**: More configuration control but requires manual JSDOM setup and lacks built-in features like snapshot testing.
  3. **Karma + real browser**: Runs tests in actual browsers (Chrome, Firefox) for highest fidelity but is significantly slower and more complex to configure.
- **Decision**: Use **Jest** with its built-in JSDOM environment.
- **Rationale**: Jest provides a zero-configuration setup for JSDOM, includes powerful snapshot testing (useful for verifying VNode/Fiber structures), and has excellent async support (critical for testing `useEffect` and the Fiber work loop). The speed and simplicity outweigh the need for real-browser testing during development.
- **Consequences**: Tests run quickly in Node, but we must be aware of JSDOM's limitations (e.g., some CSS or layout properties may not be fully implemented). We supplement with occasional manual testing in a real browser.

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| Jest + JSDOM | Fast, built-in DOM, snapshot testing, async support | JSDOM has some missing web APIs | **Yes** |
| Mocha/Chai + JSDOM | More configuration control, lighter weight | Manual setup, fewer built-in features | No |
| Karma + real browser | Highest fidelity, tests real browser engines | Slow, complex configuration, resource-heavy | No |

Our **test organization** mirrors the library's architecture:

- `__tests__/vdom/` – Tests for `createElement` and `VNode` creation.
- `__tests__/reconciler/` – Tests for `reconcileChildren`, `updateDOMProperties`, and keyed list diffing.
- `__tests__/fiber/` – Tests for `performUnitOfWork`, `workLoop`, and `commitRoot`.
- `__tests__/hooks/` – Tests for `useState`, `useEffect`, and custom hooks.
- `__tests__/integration/` – Tests that combine multiple systems (e.g., a component with state and effects).

**Key Testing Patterns:**

1. **Internal State Inspection**: We expose (or use test utilities to inspect) internal structures like the `Fiber` tree's `alternate` pointers or a component's hook linked list. This lets us verify the reconciliation algorithm selected the correct `EFFECT_TAGS` or that hook state is preserved between renders.
2. **DOM Assertions**: Using JSDOM's `document.querySelector` and node properties to check that elements were created, updated, or removed as expected.
3. **Event Simulation**: Using `element.click()` or `element.dispatchEvent` to simulate user interactions and trigger state updates, then asserting on the resulting DOM.
4. **Async Effect Testing**: Using Jest's fake timers ( `jest.useFakeTimers()` ) to control `setTimeout` and `setInterval` within `useEffect`, allowing us to verify cleanup functions and dependency-triggered re-runs.

**Testing the Fiber Scheduler**: The asynchronous, interruptible nature of the Fiber work loop presents a challenge. We test it by:

- Mocking `requestIdleCallback` to control when the browser yields time, allowing us to test incremental rendering.
- Verifying that the `commitRoot` phase happens *after* the entire `workInProgress` tree is complete, ensuring atomic updates.

## 8.2 Milestone Implementation Checkpoints

Each milestone culminates in a **test application**—a small JavaScript program that uses the library's API. Successfully running this application and observing the correct behavior in the browser console and DOM is the primary acceptance test. The table below outlines the checkpoint for each milestone.

| Milestone | Test Application Description | Key User Actions & Expected DOM Outcome | Internal State to Verify (via console.log) |
|---|---|---|---|
| **1: Virtual DOM** | Static "Hello World" and nested element tree. | Initial render produces correct HTML structure. Text nodes appear. | `VNode` tree logged, showing correct `type`, `props`, `children`. |
| **2: Reconciliation** | Counter component with a button. | Clicking button increments number text; only the text node updates. No other DOM nodes recreated. | Diff algorithm logs which nodes received `UPDATE` vs `PLACEMENT` tags. |
| **3: Fiber Architecture** | Component tree with simulated heavy render. | UI remains responsive; incremental rendering can be observed via logs. | Work loop logs each fiber processed, shows yielding via `requestIdleCallback`. |
| **4: Hooks** | Component using `useState`, `useEffect` with dependency. | State persists between renders; effect runs on mount and cleans up on unmount; effect re-runs only when dependency changes. | Hook linked list shows state values and effect dependencies preserved across renders. |

### Milestone 1: Virtual DOM – Static Rendering Checkpoint

**Test Application (`test-milestone-1.html`)**:

```javascript
// Manual creation of VNodes (without JSX)
const app = createElement('div', { className: 'container' },
  createElement('h1', null, 'Hello, Virtual DOM'),
  createElement('p', { style: 'color: blue;' }, 'This is a paragraph.'),
  'Raw text node',
  createElement('input', { type: 'text', value: 'initial' })
);
render(app, document.getElementById('root'));
```

**Verification Steps**:

1. Open the HTML file in a browser.
2. Visually confirm a heading, a blue paragraph, raw text, and an input field appear.
3. Open browser DevTools > Elements panel. Confirm the DOM structure matches the VNode tree, especially that the raw text is a proper `#text` node.
4. In the console, verify that `createElement` calls return objects with `type`, `props`, `children` properties.

**Success Criteria**:

- The DOM is created exactly as specified.
- No errors in the console.
- The `render` function correctly handles both element types and text nodes.

### Milestone 2: Reconciliation – Dynamic Updates Checkpoint

**Test Application ( `test-milestone-2.html` )**:

```javascript
let state = { count: 0 };

function updateState(newState) {

  // Re-render with new state

  const newApp = createElement('div', null,

    createElement('button', { onClick: () => updateState({ count: state.count + 1 }) }, 'Increment'),

    createElement('span', null, `Count: ${state.count}`),

    createElement('ul', null,

      createElement('li', { key: 'a' }, 'Item A'),

      createElement('li', { key: 'b' }, 'Item B')

    )

  );

  // This will call our diffing/patch function

  render(newApp, document.getElementById('root'));

}

// Initial render

updateState(state);
```

**Verification Steps**:

1. Click the "Increment" button 5 times. Observe that the count text updates each time.
2. In DevTools > Elements panel, monitor the `<span>` and `<li>` elements. Their `data-reactid` (if added) or unique identifiers should show that the *same* DOM nodes are being updated, not recreated.
3. Add a console log inside your `updateDOMProperties` function. Verify that on button click, only the text content of the `<span>` is updated, and no other properties or elements are touched.
4. Test keyed reconciliation: Modify the test to reorder the list items (swap keys 'a' and 'b') on a state update. Confirm in the Elements panel that the `<li>` DOM nodes are reordered, not destroyed and recreated.

**Success Criteria**:

- UI updates correctly on interaction.
- DOM element reuse is confirmed via DevTools (no new nodes appearing for unchanged parts).
- Keyed list reordering performs minimal DOM operations.

### Milestone 3: Fiber Architecture – Interruptible Render Checkpoint

**Test Application ( `test-milestone-3.html` )**:

```javascript
function HeavyComponent({ id }) {
  // Simulate heavy computation
  let start = performance.now();
  while (performance.now() - start < 10) { /* block for 10ms */ }
  return createElement('div', null, `Component ${id}`);
}

function App() {
  const items = Array.from({ length: 100 }, (_, i) => i);
  return createElement('div', null,
    items.map(id => createElement(HeavyComponent, { key: id, id }))
  );
}
render(createElement(App), root);
```

**Verification Steps**:

1. Observe the browser's responsiveness. If the render is synchronous, the page will freeze for ~1 second (100 items * 10ms). With Fiber, the work should be chunked.
2. Add logging to `performUnitOfWork` and `workLoop`. You should see logs for processing a few fibers, then a yield, then more processing.
3. Simulate a high-priority user interaction (like a button click that updates state) during the long render. Confirm that the interaction is processed *between* chunks of render work (the UI is not completely frozen).
4. Verify that the final DOM appears all at once after the commit phase (no partial, incomplete UI flashes).

**Success Criteria**:

- Page remains interactive during a long render.
- Work loop logs show yielding.
- Commit phase applies all changes atomically.

### Milestone 4: Hooks – State and Effects Checkpoint

**Test Application ( `test-milestone-4.html` )**:

```javascript
function CounterWithEffect() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('hello');

  useEffect(() => {
    console.log('Effect ran: count =', count);
    // Cleanup function
    return () => console.log('Cleanup for count =', count);
  }, [count]);

  useEffect(() => {
    console.log('Effect ran on mount only');
  }, []);

  return createElement('div', null,
    createElement('p', null, `Count: ${count}`),
    createElement('p', null, `Text: ${text}`),
    createElement('button', { onClick: () => setCount(count + 1) }, 'Increment Count'),
    createElement('button', { onClick: () => setText(text + '!') }, 'Append "!" to Text')
  );
}

render(createElement(CounterWithEffect), root);
```

**Verification Steps**:

1. Open console. On initial mount, you should see both effect logs.
2. Click "Increment Count". The count updates, and the console shows the cleanup for the previous count, then the effect running with the new count. The text-only effect should *not* run again.
3. Click "Append '!' to Text". Only the text updates; no effect runs because the effect's dependency array ( `[count]` ) didn't change.
4. Test hook order violation: Conditionally call a hook inside an `if` statement. The library should throw a clear error.

**Success Criteria**:

- State persists and updates correctly.
- Effects run, clean up, and respect dependency arrays.
- Hook order rule is enforced.
- No stale closures (the effect cleanup logs the correct, previous `count` value).

## Implementation Guidance

### A. Technology Recommendations Table

| Component | Simple Option (Recommended) | Advanced Option |
|---|---|---|
| Test Runner & Framework | **Jest** (includes assertion library, mock system, JSDOM environment) | Vitest (faster, Jest-compatible, but newer) |
| DOM Environment | **JSDOM** (via Jest's `testEnvironment: 'jsdom'`) | HappyDOM (alternative JSDOM implementation) |
| Test Bundling/Execution | Jest's built-in runner | Using Vite/Webpack dev server + test runner |
| Manual Browser Testing | Live Server extension in VS Code | Script to build and open in multiple browsers |

### B. Recommended File/Module Structure for Tests

```
build-your-own-react/
├── src/
│   ├── core/
│   │   ├── vdom.js          # createElement, render
│   │   └── constants.js
│   ├── reconciler/
│   │   ├── diff.js          # reconcileChildren, updateDOMProperties
│   │   └── commit.js        # commitRoot, commitWork, commitDeletion
│   ├── fiber/
│   │   ├── scheduler.js     # workLoop, performUnitOfWork
│   │   └── fiber.js         # createFiber, Fiber node structure
│   └── hooks/
│       ├── hooks.js         # useState, useEffect, dispatcher
│       └── hookInstance.js  # HookInstance structure
├── __tests__/
│   ├── vdom/
│   │   ├── createElement.test.js
│   │   └── render.test.js
│   ├── reconciler/
│   │   ├── diff.test.js
│   │   └── commit.test.js
│   ├── fiber/
│   │   ├── scheduler.test.js
│   │   └── fiber.test.js
│   ├── hooks/
│   │   ├── useState.test.js
│   │   └── useEffect.test.js
│   └── integration/
│       ├── counter.test.js
│       └── hooks-integration.test.js
├── test-apps/               # Manual test HTML files
│   ├── milestone-1.html
│   ├── milestone-2.html
│   ├── milestone-3.html
│   └── milestone-4.html
└── package.json
```

### C. Infrastructure Starter Code (Test Setup File)

Create a Jest setup file to configure global test utilities and ensure JSDOM is properly initialized.

**`jest.config.js`**:

```javascript
module.exports = {
  testEnvironment: 'jsdom',
  collectCoverageFrom: [
    'src/**/*.js',
    '!src/**/*.test.js',
  ],
  // Optional: Setup files to run before each test
  setupFilesAfterEnv: ['./jest.setup.js'],
};
```

`jest.setup.js` (global test helpers):

```javascript
// Optional: Global test utilities
global.requestIdleCallback = global.requestIdleCallback || function (cb) {
  const start = Date.now();
  return setTimeout(() => {
    cb({
      didTimeout: false,
      timeRemaining: () => Math.max(0, 50 - (Date.now() - start)),
    });
  }, 1);
};

global.cancelIdleCallback = global.cancelIdleCallback || function (id) {
  clearTimeout(id);
};

// Helper to inspect internal fiber state (for tests only)
global._getInternalState = (container) => {
  // This would require exposing the root fiber, which we might do only in DEV
  // For illustration:
  // return container._reactRootContainer._internalRoot.current;
  console.warn('Internal state inspection not enabled in production');
};
```

### D. Core Logic Skeleton Code for Test Utilities

For integration tests, we'll create a helper to render components in a JSDOM container and trigger updates.

`__tests__/testUtils.js`:

JAVASCRIPT

```javascript
import { render, createElement } from '../src/core/vdom';

/**
 * Creates a fresh JSDOM container for each test.
 * @returns {HTMLElement} A div container attached to jsdom's document.
 */
export function createTestContainer() {
  // Jest's jsdom environment provides `document` globally
  const container = document.createElement('div');
  document.body.appendChild(container);
  return container;
}

/**
 * Renders a VNode into a container and returns a helper object.
 * @param {VNode} vnode
 * @param {HTMLElement} container
 * @returns {Object} Helper with container and methods to update.
 */
export function renderTestApp(vnode, container) {
  render(vnode, container);
  return {
    container,
    // In a real implementation, we might expose a way to trigger re-renders
    // by re-calling render with a new VNode on the same container.
    rerender: (newVnode) => render(newVnode, container),
  };
}

/**
 * Simulates a click event on a DOM element.
 * @param {HTMLElement} element
 */
export function simulateClick(element) {
  const event = new MouseEvent('click', { bubbles: true });
```

```
  element.dispatchEvent(event);
}
```

### E. Language-Specific Hints (JavaScript)

- **Jest Mocks**: Use `jest.spyOn()` to mock `requestIdleCallback` and control the Fiber work loop's timing in tests.
- **Async Effects**: Use `jest.useFakeTimers()` and `jest.runAllTimers()` to fast-forward timers inside `useEffect`.
- **Snapshot Testing**: Jest's `toMatchSnapshot()` can be used to capture the structure of `VNode` or `Fiber` trees (serializable representation) to detect unintended changes.
- **DOM Assertions**: Use `expect(element.textContent).toBe(...)`, `expect(element.getAttribute('class')).toBe(...)`, etc.
- **Testing Hook Order**: To test the "hooks called in different order" error, wrap the render in a `try/catch` and assert the error message.

### F. Milestone Checkpoint Verification Commands

After implementing each milestone, run the corresponding tests and manual verification.

| Milestone | Command | Expected Test Output | Manual Verification (open test-apps/*.html) |
|---|---|---|---|
| 1 | `npm test -- __tests__/vdom/` | All tests pass. `createElement` creates correct VNode trees. `render` creates DOM elements. | Open `test-apps/milestone-1.html`. See correct static UI. |
| 2 | `npm test -- __tests__/reconciler/` | Diffing tests pass. Key reconciliation tests pass. | Open `test-apps/milestone-2.html`. Click button; only count updates. |
| 3 | `npm test -- __tests__/fiber/` | Work loop yields. Commit phase batches updates. | Open `test-apps/milestone-3.html`. UI remains responsive during long render. |
| 4 | `npm test -- __tests__/hooks/` | State updates, effects run and clean up, hook order error thrown. | Open `test-apps/milestone-4.html`. Observe console logs for effect lifecycle. |

### G. Debugging Tips for Test Failures

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **DOM not updating** | Diff algorithm not detecting changes, or commit phase not applying updates. | Log `effectTag` on fibers during reconciliation. Check if `commitRoot` is being called. | Ensure `reconcileChildren` assigns correct `effectTag` (UPDATE, PLACEMENT). Ensure `commitWork` processes the effect tags. |
| **Infinite re-render loop** | `setState` inside an effect with no dependencies, or incorrect state update triggering. | Add console logs to `useState` setter and effect. Check if dependency array is empty `[]`. | Ensure effects with empty deps run only once. Ensure state setter doesn't unconditionally set the same value. |
| **State resets on every render** | Hook linked list not being preserved across renders, or new hook instance created each time. | Log the `memoizedState` of the fiber before and after render. Check `getHookInstance` logic. | Ensure hook instances are retrieved from the `alternate` fiber during re-render, not newly created. |
| **Event listeners not firing** | Props diffing incorrectly removes/updates event listeners (e.g., `onClick` vs `onclick`). | Log `prevProps` and `nextProps` in `updateDOMProperties`. Check the property name mapping. | Use the correct prop name (`onClick`). Ensure `removeDOMProperty` cleans up old listeners and `setDOMProperty` attaches new ones. |
| **Effects running on every render** | Dependency array shallow comparison failing, or effect not being cleaned up before re-run. | Log dependency array values each render. Check if cleanup function is being called. | Ensure `useEffect` compares each dependency with `Object.is`. Ensure cleanup is executed before the next effect run. |
| **"Rendered fewer hooks than expected" error** | Conditional hook call (e.g., inside an `if` statement) changing the order of hooks between renders. | Check the component logic for any conditional hook calls. | Ensure hooks are called at the top level of the component, unconditionally. |

## 9. Debugging Guide

> **Milestone(s):** All (Milestone 1: Virtual DOM, Milestone 2: Reconciliation, Milestone 3: Fiber Architecture, Milestone 4: Hooks)

This section provides a practical guide for diagnosing and fixing common implementation bugs. Building a React-like library involves complex interactions between multiple subsystems, and bugs often manifest in subtle ways. Think of debugging this system like being a mechanic for a custom-built car: you need to understand how all the parts connect, have good diagnostic tools, and know the common failure points. The following tables and techniques will help you systematically identify and resolve issues.

### Common Bug Symptoms and Fixes

The table below organizes bugs by their observable symptoms, providing a clear path from symptom to diagnosis to fix. Many bugs stem from incorrect connections between the data structures (`VNode`, `Fiber`, `HookInstance`) or violations of the expected flow between render and commit phases.

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **DOM not rendering at all** (blank screen) | 1. `render` not being called or called with invalid `VNode`.<br>2. `createElement` returning malformed `VNode` (e.g., missing `type`).<br>3. `TEXT_ELEMENT` not handled in `render` or `createFiber`. | 1. Check console for errors.<br>2. Log the output of `createElement` for your root component.<br>3. Add a log at the start of `render` to confirm it's called.<br>4. Inspect the container element in browser dev tools—is it empty or does it have comment nodes? | 1. Ensure `render` is called with a valid `VNode`.<br>2. Verify `createElement` normalizes children correctly, converting strings/numbers to text `VNode`s with `TEXT_ELEMENT` type.<br>3. In `render` and `createFiber`, add a case for `TEXT_ELEMENT` to create a DOM text node. |
| **Text content missing or showing `[object Object]`** | Text children not being converted to text `VNode`s. The renderer is trying to render a JavaScript object as a child. | 1. Inspect the `children` array of a parent `VNode`—are strings/numbers present, or are they already `VNode`s?<br>2. Check `createElement` logic: it should map over `children` and convert primitives to text `VNode`s. | In `createElement`, transform each child: if `child` is a string, number, or boolean, create a `VNode` with `type: TEXT_ELEMENT` and `props: { nodeValue: child }`. |
| **Event listeners not firing** | 1. Event prop name incorrect (using `onclick` instead of `onClick`).<br>2. Event listener attached incorrectly (not using `addEventListener`).<br>3. Listener removed during reconciliation due to prop diffing bug. | 1. Check the generated DOM element's properties in dev tools—is the listener attached?<br>2. Compare `prevProps` and `nextProps` in `updateDOMProperties` for the event prop.<br>3. Confirm `DOM_PROPERTIES` configuration maps `onClick` to `click` event. | 1. Ensure event prop names use React convention (`onClick`, `onChange`).<br>2. In `setDOMProperty`, for props starting with "on", use `addEventListener` with the lowercased event name (minus "on").<br>3. In `removeDOMProperty`, ensure you call `removeEventListener` with the same reference. |
| **DOM not updating after `setState` call** | 1. `setState` not marking the fiber as dirty or not scheduling work.<br>2. Work loop not processing the dirty fiber.<br>3. Reconciliation incorrectly determines no changes (`effectTag` remains `null`).<br>4. Commit phase not applying the update. | 1. Log inside `setState`: is an `Update` added to the queue? Is `scheduleIdleCallback` called?<br>2. Check `workLoop`: does it pick up the work?<br>3. Inspect the fiber's `effectTag` after `reconcileChildren`—is it set to `UPDATE`?<br>4. Does `commitWork` run for the fiber? | 1. In `setState`, add the update to the hook's `queue.pending` and mark the fiber's `alternate` (or the fiber itself) as needing work. Call `scheduleIdleCallback`.<br>2. Ensure the work loop's `nextUnitOfWork` is set to the root of the dirty subtree.<br>3. In `reconcileChildren`, correctly compare old and new props, setting `effectTag = UPDATE` when props differ.<br>4. Verify `commitRoot` iterates through the effect list and calls `commitWork`. |
| **Infinite re-render loop** | 1. `setState` called unconditionally inside render (e.g., directly in function body).<br>2. `useEffect` with no dependency array or with a dependency that changes every render calls `setState`. | 1. Check for `setState` in render path (function component body).<br>2. Check `useEffect` dependencies—is a new object/array created each render?<br>3. Add logs to `workLoop` and | 1. Move `setState` calls to event handlers or effects.<br>2. For `useEffect`, provide stable dependencies (use `useState`/`useMemo` if needed) or add conditional logic.<br>3. Ensure `performUnitOfWork` returns the next fiber correctly (child, |

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| | 3. Work loop incorrectly re-creates work for already completed fibers. | `performUnitOfWork` —is the same fiber repeatedly processed? | sibling, then parent's sibling) and doesn't create cycles. |
| **State resets to initial value on every render** | 1. `useState` initializer runs on every render, overwriting memoized state.<br>2. Hook linked list mismatched between renders (hook order not stable).<br>3. New fiber created on each render, losing `memoizedState`. | 1. Log the `initialState` value in `useState` —is it being called each time?<br>2. Log the `fiber.memoizedState` chain before and after hooks are called.<br>3. Check if component `type` changes between renders, causing a new fiber. | 1. In `getHookInstance`, only use `initialState` when creating a new hook (when `hook` is `null`).<br>2. Ensure hooks are always called in the same order (no conditional hooks).<br>3. Ensure reconciliation reuses fibers when `type` is the same (via `alternate`). |
| **`useEffect` runs on every render, not just when dependencies change** | Dependency array shallow comparison incorrectly returns `true` (always finds a difference). | 1. Log the `prevDeps` and `nextDeps` in `useEffect` comparison.<br>2. Check if dependency array is `undefined` (treated as "run after every render"). | Implement shallow comparison: iterate over arrays, return `false` if any element is not strictly equal ( `!Object.is` ). If `deps` is `undefined`, treat as "always different". |
| **`useEffect` cleanup not running** | 1. Cleanup function not stored in `HookInstance`.<br>2. Cleanup not called before running effect again.<br>3. Effect not tagged during commit phase. | 1. Inspect the `HookInstance` after effect runs—is `cleanup` set?<br>2. Does the commit phase call cleanup for effects with tag `EFFECT`?<br>3. Check if the fiber's `effectTag` includes effect flags. | 1. When scheduling an effect, store the cleanup function on the hook's `cleanup` field.<br>2. In `commitWork`, before applying new effects, call existing cleanups.<br>3. Ensure `useEffect` marks the fiber with an effect tag (e.g., by setting `fiber.effectTag |
| **List items reorder incorrectly (keys ignored)** | 1. `key` prop not being extracted from `VNode` to `Fiber`.<br>2. `reconcileChildren` not matching fibers by `key`.<br>3. Using array index as `key` causes unnecessary recreation. | 1. Log the `key` property on child `VNode`s and the corresponding fibers.<br>2. Trace through `reconcileChildren` —does it create a map of old fibers by key?<br>3. Observe DOM elements in dev tools—are they being removed/added instead of moved? | 1. In `createFiber`, copy `vnode.key` to `fiber.key`.<br>2. Implement keyed reconciliation: create a map of existing fibers by key, then iterate new children, reusing matching fibers.<br>3. Advise using stable unique IDs as keys, not indices. |
| **Memory leak (DOM nodes not removed)** | 1. Deleted fibers not being detached from DOM.<br>2. `commitDeletion` not called or not removing nodes.<br>3. Event listeners not removed from deleted DOM nodes. | 1. Check if `commitDeletion` is called for fibers with `effectTag = DELETION`.<br>2. Use memory profiler in dev tools to see if detached DOM nodes persist.<br>3. Add log in `removeDOMProperty` for event listeners. | 1. Ensure `reconcileChildren` marks old fibers for deletion when no matching new child.<br>2. In `commitDeletion`, recursively remove DOM nodes and call cleanup on hooks.<br>3. In `removeDOMProperty`, remove event listeners using the stored handler reference. |

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **SVG/MathML elements render as HTML elements** | Missing namespace propagation. SVG elements created without `http://www.w3.org/2000/svg` namespace. | Inspect the SVG element in dev tools—does it have correct namespace? Is it in HTML namespace? | In `render`, when creating DOM elements, check if `type` is `'svg'` or child of SVG. Pass the `SVG_NAMESPACE` to `document.createElementNS`. |
| **Component renders only once, then updates don't happen** | Work loop stops after first commit ( `nextUnitOfWork` never set again). | Add logs in `workLoop` : after `commitRoot` , is `nextUnitOfWork` set to process pending work? | Ensure `scheduleIdleCallback` is called again if there are pending updates (dirty fibers). Keep a `pendingCommit` flag or queue. |
| **"Hooks must be called in the same order" error falsely triggered** | Hook dispatcher's current hook index not reset properly between component renders. | Log the `currentHook` index in `beginHookDispatcher` and `advanceToNextHook` . Does it reset to `null` for each component? | In `beginHookDispatcher` , set the fiber's `memoizedState` (hook list head) as the current hook. In `endHookDispatcher` , reset the global current hook tracker. |
| **Stale state in event handlers or effects** | Closure capturing outdated state value from previous render. | Log state value inside handler and compare to current `fiber.memoizedState` . | The `setState` function from `useState` should always reference the latest state via the hook's queue and fiber. Ensure updates are applied before next render. |

## Debugging Techniques: Logging and Inspection

When the above table doesn't directly solve your issue, you'll need to actively inspect the internal state of your library. Think of this like adding diagnostic sensors throughout the engine—you can monitor the system's operation in real time.

### Strategic Console Logging

Add console logs at critical junctions in your code to trace execution flow and data transformations. The key is to log structured data that reveals the state of your algorithms.

**Work Loop and Fiber Tree Traversal:**

```javascript
function workLoop(deadline) {
  console.log('🔁 workLoop start', { timeRemaining: deadline.timeRemaining() });
  while (nextUnitOfWork && deadline.timeRemaining() > 1) {
    console.group('Processing fiber:', nextUnitOfWork.type);
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    console.groupEnd();
  }
  if (!nextUnitOfWork && pendingCommit) {
    console.log('✅ Render phase complete, committing...');
    commitRoot();
  }
  scheduleIdleCallback(workLoop);
}
```

**Fiber Reconciliation:** Inside `reconcileChildren`, log the old and new children arrays, their keys, and the resulting effect tags. This helps verify the diffing logic.

**Hook Execution:** In `getHookInstance`, log the current fiber, hook index, and memoized state. This helps track hook order and state values.

**Commit Phase:** In `commitWork`, log the fiber type and effect tag to see what DOM operations are being applied.

### Browser Dev Tools Inspection

Since your library manages its own internal data structures, you can expose them to the global scope for easy inspection during development.

**Expose the Fiber Root:** Modify your `render` function to attach the root fiber to the DOM container:

```javascript
function render(vnode, container) {
  // ... initial setup
  container._reactRootFiber = rootFiber;  // Now you can inspect in console
}
```

Then in browser console, you can examine the entire fiber tree:

```javascript
// Get the root fiber
const root = document.getElementById('app')._reactRootFiber;

// Walk the tree manually
let fiber = root;
while (fiber) {
  console.log(fiber.type, fiber.key, fiber.effectTag);
  if (fiber.child) fiber = fiber.child;
  else if (fiber.sibling) fiber = fiber.sibling;
  else fiber = fiber.parent?.sibling;
}
```

**Inspect Hook Linked List:** Since hooks are stored as a linked list on `fiber.memoizedState`, you can traverse them:

```javascript
const fiber = /* get a function component fiber */;
let hook = fiber.memoizedState;
let index = 0;
while (hook) {
  console.log(`Hook ${index}:`, { memoizedState: hook.memoizedState, queue: hook.queue });
  hook = hook.next;
  index++;
}
```

**DOM Mutation Breakpoints:** Use the browser's "Break on" feature to pause when a DOM subtree is modified. Right-click an element in the Elements panel, select "Break on" → "Subtree modifications". This will pause execution in your `commitWork` function when that element's children change, helping you trace which fiber update caused the change.

### Visualizing the Fiber Tree

Create a simple debug utility that recursively prints the fiber tree structure. This can be called from console when needed:

```javascript
function printFiberTree(fiber, indent = 0) {
  const prefix = ' '.repeat(indent);
  const tag = fiber.tag === FIBER_TYPES.HOST ? 'HOST' :
              fiber.tag === FIBER_TYPES.FUNCTION ? 'FUNCTION' : 'TEXT';
  console.log(`${prefix}${fiber.type} [${tag}] ${fiber.effectTag ? `EFFECT:${fiber.effectTag}` : ''}`);
  if (fiber.child) printFiberTree(fiber.child, indent + 2);
  if (fiber.sibling) printFiberTree(fiber.sibling, indent);
}
```

### Performance Profiling

Use the browser's Performance tab to record interactions. Look for:

- Long tasks blocking the main thread (your work loop should yield frequently)
- Excessive DOM mutations (batched commits should minimize reflows)
- Memory growth over time (look for detached DOM nodes not being garbage collected)

The key insight is that most bugs arise from inconsistencies between the three parallel trees: the Virtual DOM (what you want), the Fiber tree (work in progress), and the actual DOM (what's rendered). By making these trees visible and tracing the transformations between them, you can pinpoint where the disconnect happens.

---

## Implementation Guidance

This implementation guidance provides concrete tools and patterns for debugging your React-like library. While the core learning is in implementing the library itself, having robust debugging capabilities will save you countless hours.

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Debugging Output | `console.log` with custom formatters | Custom DevTools panel using browser extensions API |
| State Inspection | Expose global `__REACT_DEBUG__` object | Integrate with React DevTools via `__REACT_INTERNAL` hooks |
| Performance Monitoring | `console.time` / `timeEnd` markers | User Timing API with performance marks and measures |
| Error Tracking | `try/catch` with detailed error context | Error boundaries with component stack traces |

### B. Recommended File Structure

Add a dedicated debug module to your project:

```
project-root/
  src/
    core/
      debug.js          ← Debug utilities and inspectors
    index.js           ← Main library entry (exports debug tools)
```

### C. Debug Infrastructure Starter Code

Here's a complete, ready-to-use debug module that you can import and use immediately:

JAVASCRIPT

```javascript
// src/core/debug.js

/**
 * Debug utilities for inspecting the internal state of the React-like library.
 * Attach to window for easy console access during development.
 */

export const ReactDebug = {
  // Store references to all root fibers for inspection
  roots: new Set(),

  // Track render cycles for performance monitoring
  renderCount: 0,

  /**
   * Enable debugging by attaching to window and setting up global helpers.
   */
  enable() {
    window.__REACT_DEBUG__ = {
      getRoots: () => Array.from(this.roots),
      printFiberTree: this.printFiberTree,
      printHookList: this.printHookList,
      startProfiling: this.startProfiling,
      stopProfiling: this.stopProfiling,
    };
    console.log('🔧 React debugging enabled. Use __REACT_DEBUG__ in console.');
  },

  /**
   * Register a root fiber for debugging.
   */
  registerRoot(container, rootFiber) {
    container._reactRootFiber = rootFiber;
    this.roots.add({ container, fiber: rootFiber });
```

```
},

/**
 * Print a visual representation of the fiber tree.
 */
printFiberTree(fiber, indent = 0, maxDepth = 10) {
  if (!fiber || indent > maxDepth * 2) return;

  const prefix = ' '.repeat(indent);
  const tagNames = { 0: 'HOST', 1: 'FUNCTION', 2: 'CLASS', 3: 'TEXT' };
  const effectNames = { 1: 'PLACEMENT', 2: 'UPDATE', 4: 'DELETION' };

  const type = fiber.type === TEXT_ELEMENT ? '#text' :
               typeof fiber.type === 'function' ? fiber.type.name :
               fiber.type;
  const tag = tagNames[fiber.tag] || 'UNKNOWN';
  const effect = fiber.effectTag ? effectNames[fiber.effectTag] : '-';
  const key = fiber.key ? `key="${fiber.key}"` : '';

  console.log(`${prefix}${type} [${tag}] ${key} ${effect} ${fiber.stateNode ? '🟢' : '🔴'}`);

  // Print hooks for function components
  if (fiber.tag === 1 /* FUNCTION */ && fiber.memoizedState) {
    this.printHookList(fiber.memoizedState, indent + 2);
  }

  // Recursively print children
  if (fiber.child) this.printFiberTree(fiber.child, indent + 2, maxDepth);
  if (fiber.sibling) this.printFiberTree(fiber.sibling, indent, maxDepth);
},

/**
 * Print the hook linked list for a fiber.
```

```
 */
printHookList(hook, indent = 0) {
  const prefix = ' '.repeat(indent);
  let index = 0;
  let current = hook;

  while (current) {
    const type = current.queue ? 'useState' : current.cleanup ? 'useEffect' : 'Unknown';
    console.log(`${prefix}Hook ${index} [${type}]:`, {
      memoizedState: current.memoizedState,
      queueSize: current.queue ? this._countQueue(current.queue) : 0,
      hasCleanup: !!current.cleanup
    });
    current = current.next;
    index++;
  }
},

/**
 * Count updates in a hook queue.
 */
_countQueue(queue) {
  if (!queue || !queue.pending) return 0;
  let count = 0;
  let update = queue.pending.next;
  while (update && update !== queue.pending) {
    count++;
    update = update.next;
  }
  return count;
},

// Performance profiling state
```

```
_profileStartTime: null,

_profileMeasures: [],


/**
 * Start profiling the next render cycle.
 */
startProfiling(label = 'render') {
  this._profileStartTime = performance.now();
  this._profileLabel = label;
  console.log(`⚡ Profiling started: ${label}`);
},


/**
 * Stop profiling and log results.
 */
stopProfiling() {
  if (!this._profileStartTime) return;
  const duration = performance.now() - this._profileStartTime;
  this._profileMeasures.push({ label: this._profileLabel, duration });
  console.log(`⚡ ${this._profileLabel}: ${duration.toFixed(2)}ms`);
  this._profileStartTime = null;
},


/**
 * Wrap a function with performance measurement.
 */
profile(fn, label) {
  return (...args) => {
    const start = performance.now();
    const result = fn(...args);
    const duration = performance.now() - start;
    console.log(`⏱️ ${label}: ${duration.toFixed(2)}ms`);
    return result;
```

```
    };
  }
};

// Auto-enable in development
if (typeof window !== 'undefined' && process.env.NODE_ENV === 'development') {
  setTimeout(() => ReactDebug.enable(), 0);
}

export default ReactDebug;
```

### D. Integration with Render Function

Modify your `render` function to register roots for debugging:

```javascript
import { ReactDebug } from './core/debug.js';

function render(vnode, container) {
  // TODO: Existing render logic...

  // After creating root fiber
  const rootFiber = createFiber(/* ... */);

  // Register for debugging
  if (typeof process !== 'undefined' && process.env.NODE_ENV === 'development') {
    ReactDebug.registerRoot(container, rootFiber);
  }

  // TODO: Schedule initial work...
}
```

### E. Debug-Enhanced Work Loop

Add performance markers and debug logs to your work loop:

JAVASCRIPT

```javascript
function workLoop(deadline) {
  // TODO 1: Start performance mark if profiling
  if (window.__REACT_DEBUG__?.isProfiling) {
    performance.mark('workLoop-start');
  }

  // TODO 2: Log loop start with time remaining
  console.debug('🔄 workLoop', {
    timeRemaining: deadline.timeRemaining(),
    nextUnitOfWork: !!nextUnitOfWork
  });

  while (nextUnitOfWork && deadline.timeRemaining() > 1) {
    // TODO 3: Process unit of work with debug wrapper
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);

    // TODO 4: Optional: break after processing many fibers for debugging
    if (window.__REACT_DEBUG__?.breakAfter &&
        ++processedCount > window.__REACT_DEBUG__.breakAfter) {
      console.warn('⏸️ Debug break: Processed limit of fibers');
      break;
    }
  }

  // TODO 5: Log completion and schedule next frame
  if (!nextUnitOfWork && pendingCommit) {
    console.debug('✅ Render phase complete');
    commitRoot();
  }

  scheduleIdleCallback(workLoop);

  // TODO 6: End performance mark if profiling
```

```
  if (window.__REACT_DEBUG__?.isProfiling) {
    performance.mark('workLoop-end');
    performance.measure('workLoop', 'workLoop-start', 'workLoop-end');
  }
}
```

### F. JavaScript-Specific Debugging Hints

1. **Object Inspection:** Use `console.dir(object, { depth: null })` to see deeply nested fiber trees.
2. **Breakpoints:** Add `debugger;` statements in strategic locations (like `setState` or `commitWork`) to pause execution.
3. **Monkey Patching:** Temporarily override functions to add logging without modifying source:

```javascript
const originalSetState = setState;
window.setState = function(...args) {
  console.log('setState called with:', args);
  return originalSetState.apply(this, args);
};
```

4. **DOM Inspection:** Use `inspect(element)` in Chrome console to jump to the element in Elements panel.
5. **Memory Snapshots:** Take heap snapshots before and after operations to detect leaks.

### G. Debugging Milestone Checkpoints

After implementing each milestone, use these debugging scenarios to verify correctness:

**Milestone 1 Checkpoint:**

1. Run a simple app that creates a button with text.
2. In console: `__REACT_DEBUG__.printFiberTree(document.getElementById('app')._reactRootFiber)`
3. Expected: Should show a tree with button host fiber and text fiber.
4. Bug sign: Text shows as `[object Object]` → Check text VNode creation.

**Milestone 2 Checkpoint:**

1. Create a counter app (button + count display).
2. Click the button 3 times.
3. In console: `document.querySelector('button')._listeners` (if exposed)
4. Expected: Click handler attached, count updates without full DOM replacement.
5. Bug sign: Entire button recreated each click → Check reconciliation `effectTag` assignment.

**Milestone 3 Checkpoint:**

1. Create a long list (100 items).
2. Profile performance while scrolling (if interactive) or updating.
3. Check browser Performance tab for long tasks.
4. Expected: Work loop yields control, no jank.
5. Bug sign: UI freezes → Work loop doesn't respect `deadline.timeRemaining()`.

**Milestone 4 Checkpoint:**

1. Create component with `useState` and `useEffect`.
2. Trigger state update.
3. In console: `__REACT_DEBUG__.printHookList(componentFiber.memoizedState)`
4. Expected: Hook state updates, effect runs conditionally.
5. Bug sign: Infinite loop → Check `useEffect` dependencies or `setState` in render.

By incorporating these debugging tools and techniques, you'll transform the frustrating process of hunting subtle bugs into a systematic investigation. The key is making the invisible internal state visible, allowing you to verify each transformation in the pipeline from VNode to Fiber to DOM.

## 10. Future Extensions

> **Milestone(s):** This section goes beyond the four core milestones to explore how the foundational architecture we've built could be extended with advanced React-like features. These extensions demonstrate how the core concepts of Virtual DOM, Fiber reconciliation, and Hooks provide a flexible platform for building increasingly sophisticated UI capabilities.

Having implemented the core architecture—Virtual DOM representation, reconciliation diffing, interruptible Fiber rendering, and Hooks for state and effects—we now have a solid foundation that mirrors React's essential capabilities. This architecture wasn't designed as a dead end but as a **pluggable platform** that can be extended in numerous directions. The Fiber tree structure, the hook storage system, and the reconciliation algorithm provide the **conceptual hooks** (pun intended) where additional features can be attached.

Think of our current library as a **basic car chassis with engine, transmission, and wheels**—it drives and gets you places. The extensions in this section are like adding power steering, air conditioning, navigation systems, and advanced safety features. Each builds upon the fundamental mechanical systems without requiring a complete redesign of the vehicle.

### Potential Features to Implement

The following features represent natural extensions that build upon our existing architecture. They're arranged from "closest to what we already have" to "more ambitious architectural additions." Each extension demonstrates how specific patterns in React emerge from combining the core primitives we've implemented.

#### 1. useReducer Hook: State Management with Reducer Pattern

> **Mental Model:** Think of `useReducer` as a **state machine with rules**. Instead of directly setting state (which is like telling someone exactly what to do), you dispatch "intentions" (actions) that get processed by a predefined set of rules (the reducer). This is like submitting a work request form rather than walking into an office and rearranging furniture yourself.

**Description:** The `useReducer` hook provides an alternative to `useState` for managing complex state logic. It follows the reducer pattern popularized by Redux, where state transitions are centralized in a pure function that takes the current state and an action, returning the next state. This is particularly useful when the next state depends on the previous state in complex ways, or when multiple values need to be updated atomically.

**Architectural Impact:**

- **Hook System:** Requires a new hook type ( `HOOK_TYPES.REDUCER` ) in the `HookInstance` structure
- **State Updates:** Uses the same update queue mechanism as `useState` but with action objects instead of direct values
- **Dispatcher:** The returned `dispatch` function must be stable across renders (same reference)

**Key Implementation Steps:**

1. Extend the `HookInstance` structure to support reducer-style updates (action queue instead of direct value updates)
2. Create a `createReducerHook` function similar to `createStateHook` but with reducer logic
3. Modify the `getHookInstance` dispatcher to recognize and handle reducer hooks

4. Implement the `dispatch` function that queues actions and triggers re-renders
5. Process queued actions during the render phase to compute the new state

**Relationship to Existing Architecture:** `useReducer` shares almost identical infrastructure with `useState`—both use the same hook storage mechanism, same update queue structure, and same re-render triggering. The only difference is the transformation logic between queued updates and the final state value.

**Example Use Case:**

```javascript
// Instead of multiple useState calls with interdependent logic:
const [count, setCount] = useState(0);
const [isLoading, setIsLoading] = useState(false);

// Use a reducer that manages related state atomically:
const initialState = { count: 0, isLoading: false };
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, count: state.count + 1 };
    case 'startLoading':
      return { ...state, isLoading: true };
    case 'stopLoading':
      return { ...state, isLoading: false };
    default:
      return state;
  }
}
const [state, dispatch] = useReducer(reducer, initialState);
```

**Data Structure Extension:**

| Field Name | Type | Description |
|---|---|---|
| `reducer` | `function(state, action) => newState` | Pure function that calculates next state from current state and action |
| `queuedActions` | `Array<Action>` | Array of pending actions to process during next render |
| `dispatch` | `function(action)` | Stable function reference that queues actions and schedules work |

### 2. useContext Hook and Context API: Prop Drilling Solution

> **Mental Model:** Imagine a **family heirloom** passed down through generations. Instead of each parent handing it directly to each child (prop drilling), the family places it in a special display case (context) in the living room. Any family member can access it directly when needed, regardless of their position in the family tree.

**Description:** The Context API provides a way to share values (themes, user data, preferences) across the component tree without explicitly passing props through every intermediate component. The `useContext` hook allows function components to subscribe to context changes, re-rendering when the context value updates.

**Architectural Impact:**

- **Fiber Tree:** Requires context values to be stored on fibers and propagated during tree traversal
- **Reconciliation:** Context consumers need to re-render when context values change, requiring dependency tracking
- **Hook System:** Adds a new hook type ( `HOOK_TYPES.CONTEXT` ) that subscribes to context changes

**Key Implementation Steps:**

1. Create `createContext` function that returns a context object with `Provider` and `Consumer` components
2. Store context value on the fiber of the nearest `Provider` ancestor
3. During fiber reconciliation, propagate context values from parent to child fibers
4. Implement `useContext` hook that reads the current context value from the fiber and subscribes to changes
5. When a context value changes, mark all consuming components as needing re-render

**Relationship to Existing Architecture:** Context leverages the Fiber tree's parent-child links for value propagation. The Provider is just a special component type that doesn't render anything itself but provides context values to its descendants. Consumers use the same subscription mechanism as effects to re-render when dependencies change.

**Performance Consideration:** Naïve context implementation causes all consumers to re-render when any context value changes. React's optimization involves multiple context "versions" and selective subscription, which could be a more advanced extension.

**Data Structure Extension:**

| Structure Name | Fields | Description |
|---|---|---|
| `Context` | `value: any` , `Provider: FunctionComponent` , `Consumer: FunctionComponent` | Context object created by `createContext` |
| `Fiber.contextDependencies` | `Map<Context, Fiber>` | Tracks which contexts this fiber depends on for selective re-rendering |
| `ProviderFiber.contextValue` | `any` | The current value provided by this Provider fiber |

### 3. Concurrent Features: startTransition and useDeferredValue

> **Mental Model:** Think of a **restaurant kitchen during rush hour**. The chef has two types of orders: urgent (steak that will get cold) and deferrable (dessert that can wait). Instead of making customers wait for all orders, the chef prioritizes urgent tasks while working on deferrable ones during lulls. `startTransition` marks work as "deferrable," and `useDeferredValue` provides a slightly stale value while computing the fresh one.

**Description:** Concurrent features allow the renderer to work on multiple "versions" of the UI simultaneously and interrupt low-priority work for high-priority updates (like user input). This prevents the UI from freezing during expensive renders. `startTransition` marks state updates as non-urgent, while `useDeferredValue` returns a value that may be one render behind for expensive computations.

**Architectural Impact:**

- **Scheduler:** Requires priority levels for different types of work (urgent vs. transition)
- **Fiber Tree:** Needs to maintain multiple concurrent trees (current, work-in-progress, and potentially others)
- **Hooks:** `useDeferredValue` needs to schedule a second render pass with lower priority

**Key Implementation Steps:**

1. Extend the scheduler to support multiple priority levels (Immediate, UserBlocking, Normal, Low)

2. Modify `workLoop` to check `deadline.timeRemaining()` and yield for low-priority work
3. Implement `startTransition` that schedules updates with low priority
4. Create `useDeferredValue` hook that renders with stale value first, then schedules a re-render with fresh value
5. Add lane-based update model to track which updates belong to which priority "lane"

**Relationship to Existing Architecture:** This builds directly on the Fiber architecture's interruptibility. Our current implementation can already yield between units of work; concurrent features add prioritization to determine *when* to yield.

**Complexity Warning:** Full concurrent rendering is one of React's most complex features. A simplified version could implement only priority-based yielding without full lane architecture.

**Data Structure Extension:**

| Field Name | Type | Description |
|---|---|---|
| `Fiber.lanes` | `number` | Bitmask representing which priority lanes this fiber belongs to |
| `Update.lane` | `number` | The priority lane of this update |
| `Scheduler.currentPriorityLevel` | `number` | The current priority level being processed |
| `DeferredValueHook` | `{ baseValue: any, deferredValue: any }` | Stores both current and deferred values |

### 4. Custom JSX Pragma: Beyond createElement

> **Mental Model:** Think of JSX as **shorthand notation** that needs translation. By default, JSX `<div />` translates to `createElement('div')`. A custom pragma is like installing a different translator that understands additional syntax or optimizations—like having a translator who not only converts words but also suggests more idiomatic phrases.

**Description:** A JSX pragma allows developers to customize the function called to process JSX elements. While we currently use `createElement` by default, a custom pragma could enable automatic memoization, different element representations, or compile-time optimizations.

**Architectural Impact:**

- **Build Configuration:** Requires Babel/TypeScript configuration to use a different transform function
- **Element Creation:** The pragma function must return valid `VNode` structures
- **Compatibility:** Should maintain backward compatibility with standard `createElement`

**Key Implementation Steps:**

1. Create an alternative to `createElement` (e.g., `jsx` or `jsxs` for static children)
2. Configure Babel with `/** @jsx MyLibrary.jsx */` pragma comment
3. Implement compile-time optimizations like hoisting static elements
4. Add development/production builds with different pragma functions

**Relationship to Existing Architecture:** This is mostly a compile-time feature that changes what code gets sent to the runtime. The runtime still receives `VNode` structures and processes them through the same reconciliation pipeline.

**React 17+ Compatibility:** Modern React uses the new JSX transform that automatically imports `jsx` or `jsxs` from React. Implementing this would make our library compatible with React's modern toolchain.

**Implementation Example:**

```javascript
// Custom pragma with automatic key generation for static elements
function jsx(type, props, key) {
  // Static elements (no props or children) get automatic keys
  if (props == null && key == null && isStaticType(type)) {
    key = `__static_${staticCounter++}`;
  }
  return createElement(type, props, key);
}
```

### 5. Error Boundaries: Graceful Failure Handling

> **Mental Model:** Think of error boundaries as **circuit breakers** in an electrical system. When a short circuit occurs in one room, the circuit breaker trips for that circuit only, preventing a house-wide blackout. The rest of the house continues to function normally, and you can investigate the faulty circuit separately.

**Description:** Error boundaries are components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of crashing the entire application. They implement a `static getDerivedStateFromError()` or `componentDidCatch()` lifecycle method (or their Hook equivalents).

**Architectural Impact:**

- **Render Phase:** Need to wrap rendering in try-catch blocks
- **Fiber Tree:** Must track error boundaries in the tree hierarchy
- **Commit Phase:** Need special handling for error state commits

**Key Implementation Steps:**

1. Add error boundary detection during fiber creation (check for special static methods)
2. Wrap `performUnitOfWork` in try-catch when processing a fiber with error boundary parent
3. When error is caught, mark the error boundary fiber with special effect tag
4. During commit, render fallback UI instead of the failed subtree
5. Implement `componentDidCatch` lifecycle for error reporting

**Relationship to Existing Architecture:** Error boundaries leverage the Fiber tree's parent-child relationships to propagate error information upward. They're essentially components with special reconciliation behavior when errors occur.

**Simplified Approach:** A minimal implementation could catch errors during render and replace the entire subtree with an error message component, without full lifecycle support.

**Data Structure Extension:**

| Field Name | Type | Description |
| --- | --- | --- |
| `Fiber.errorInfo` | `{ error: Error, errorBoundary: Fiber }` | Stores caught error and which boundary will handle it |
| `EFFECT_TAGS.CAPTURE_ERROR` | `number` | Special effect for error boundary fallback rendering |
| `ErrorBoundaryComponent` | `{ getDerivedStateFromError: function, componentDidCatch: function }` | Special component type |

### 6. Portals: Rendering Outside Component Hierarchy

> **Mental Model:** Imagine a **picture-in-picture television**. The main content renders in the normal TV screen, but you can have a smaller window showing a different channel that's controlled by the same remote. The portal content is logically part of your component tree but physically rendered somewhere else in the DOM.

**Description:** 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 modals, tooltips, and popovers that need to break out of container overflow constraints or z-index stacking contexts.

**Architectural Impact:**

- **Reconciliation:** Portal children need special handling during diffing since their DOM parent differs from their Fiber parent
- **Commit Phase:** Portal DOM operations must target the portal container instead of the parent fiber's DOM node
- **Event Bubbling:** Events from portal content should bubble through the React tree, not the DOM tree

**Key Implementation Steps:**

1. Create a special Portal fiber type ( `FIBER_TYPES.PORTAL` )
2. Store the target DOM container reference on the portal fiber
3. During reconciliation, treat portal children as normal but with different DOM parent
4. In `commitWork` , append portal children to the target container instead of parent DOM node
5. Implement event system that respects React tree hierarchy over DOM hierarchy

**Relationship to Existing Architecture:** Portals reuse almost all existing reconciliation logic—the only difference is the target DOM container during the commit phase. The Fiber tree structure remains unchanged, maintaining parent-child relationships for context and updates.

**Event Handling Complexity:** The most complex aspect is making events from portal content bubble through the React component hierarchy rather than the actual DOM hierarchy, which may require modifying our event system.

**Implementation Sketch:**

```javascript
function createPortal(children, container) {
  return {
    $$typeof: Symbol.for('react.portal'), // Special type identifier
    children: children,
    container: container
  };
}

// In reconciler: handle portal type specially
if (fiber.type === PortalSymbol) {
  // Reconcile children normally, but during commit:
  // Append to fiber.container instead of fiber.parent.stateNode
}
```

### 7. Lazy Loading and Suspense: Code Splitting and Async Rendering

> **Mental Model:** Think of Suspense as a **loading dock with a temporary placeholder**. When a truck (async component) hasn't arrived yet, the dock shows a "Loading..." sign instead of being empty. Once the truck arrives, the goods are unloaded and the real content replaces the placeholder—all without blocking other docks from operating.

**Description:** `React.lazy()` enables code splitting by dynamically importing components, while `Suspense` displays a fallback UI while waiting for the lazy component to load. This can be extended to support data fetching scenarios where components "suspend" while waiting for asynchronous data.

**Architectural Impact:**

- **Reconciliation:** Needs to handle "incomplete" fibers that are waiting for promises
- **Scheduler:** Must be able to pause and resume rendering when promises resolve
- **Commit Phase:** Should commit fallbacks first, then re-render when promises settle

**Key Implementation Steps:**

1. Implement `lazy()` function that returns a special component that throws a promise while loading
2. Wrap rendering in try-catch that catches "promise throws" from lazy components
3. When promise is caught, mark the Suspense boundary as "pending" and render fallback
4. When promise resolves, re-render the Suspense boundary with the loaded component
5. Extend to support multiple pending promises within a single Suspense boundary

**Relationship to Existing Architecture:** Suspense builds on the Fiber architecture's ability to interrupt and resume work. The "throwing a promise" pattern is essentially a way to signal to the scheduler that this fiber isn't ready to render yet.

**Simplified Version:** Initially implement only lazy component loading without data fetching Suspense, which is significantly more complex.

**Data Flow:**

1. Render encounters lazy component → throws promise
2. Nearest Suspense boundary catches promise → renders fallback
3. Promise added to pending set, scheduler notified
4. When promise resolves, Suspense boundary re-renders
5. This time lazy component doesn't throw → real component renders

### 8. Server-Side Rendering (SSR): Hydration

> **Mental Model:** Hydration is like **pouring water into a pre-built ice cube tray**. The server sends down the HTML "mold" (pre-rendered DOM), and the client-side JavaScript "fills it in" with interactivity (event handlers, state). The key is matching the existing DOM structure exactly without tearing it down and rebuilding.

**Description:** Server-side rendering generates HTML on the server and sends it to the client for faster initial page loads. Hydration is the process where the client-side React attaches to existing server-rendered DOM, adding event listeners and enabling interactivity without re-creating the DOM.

**Architectural Impact:**

- **Render Function:** Needs a `hydrate` function that reuses existing DOM nodes instead of creating new ones
- **Reconciliation:** Must compare Virtual DOM to existing DOM rather than previous Virtual DOM
- **Error Handling:** Mismatches between server and client render must be handled gracefully

**Key Implementation Steps:**

1. Create `hydrate(vnode, container)` function that reuses `container`'s existing children

2. During reconciliation, compare `VNode` to existing DOM node instead of previous `VNode`
3. Implement "hydration mismatch" detection and recovery (client-side re-render)
4. Add special handling for text content differences and attribute mismatches
5. Ensure event handlers attach correctly to existing DOM nodes

**Relationship to Existing Architecture:** Hydration is essentially a special mode of reconciliation where one of the "trees" being compared is the actual DOM instead of a previous Virtual DOM. The diffing algorithm remains conceptually similar but with different comparison targets.

**Performance Consideration:** Hydration should be optimized to skip checks for content that definitely matches (like static text) to improve performance.

**Implementation Approach:**

```javascript
function hydrate(vnode, container) {
  // Store that we're in hydration mode
  isHydrating = true;
  // Use existing DOM node as the "previous" tree
  const rootFiber = createFiberFromDOM(container.firstChild, vnode);
  // Proceed with normal work loop but with DOM reuse logic
  // ...
}
```

### 9. Synthetic Event System: Normalized Browser Events

> **Mental Model:** Think of synthetic events as a **universal translator for browser quirks**. Different browsers speak different dialects of the event language (IE vs Chrome vs Firefox). The synthetic event system listens to all these dialects and translates them into a standard language that your components understand.

**Description:** React's synthetic event system normalizes events across browsers, provides event pooling for performance, and implements event delegation for efficiency. Instead of attaching individual event listeners to every DOM node, React attaches a single listener at the root and dispatches events through the Virtual DOM.

**Architectural Impact:**

- **Event Handling:** Replace direct `addEventListener` calls with a centralized event system
- **Performance:** Needs event pooling to reduce garbage collection
- **Prop System:** Event props ( `onClick` , etc.) would register with synthetic system instead of direct DOM binding

**Key Implementation Steps:**

1. Create event registry mapping event names to handlers
2. Attach single event listeners at the root container for each event type
3. Implement event dispatch that finds the target fiber and calls the appropriate handler
4. Add event normalization (e.g., `event.target` , `event.currentTarget` , `stopPropagation` )
5. Implement event pooling to reuse event objects

**Relationship to Existing Architecture:** This would replace the current `setDOMProperty` handling of event listeners. Instead of `element.addEventListener(name, handler)` , we'd register the handler in a central system and let it handle dispatch.

**Complexity Note:** Full synthetic event system with pooling and all browser normalizations is complex. A minimal version could start with simple event delegation without pooling.

**Data Structure Extension:**

| Structure Name | Fields | Description |
|---|---|---|
| `EventRegistry` | `Map<string, Array<{fiber: Fiber, handler: Function}>>` | Maps event types to fibers with handlers |
| `SyntheticEvent` | `{ nativeEvent: Event, type: string, target: DOMElement, currentTarget: DOMElement }` | Normalized event object |
| `EventPool` | `Array<SyntheticEvent>` | Pool of reusable event objects |

### 10. Ref API: useRef and forwardRef

> **Mental Model:** Refs are like **name tags on coat hooks**. The coat hook (DOM element or component instance) exists in the coat room (DOM), and you put a name tag (ref) on it so you can find it later when you need your coat (direct access). `forwardRef` is like having a helper who passes the name tag along to the actual coat hook inside their room.

**Description:** Refs provide a way to access DOM nodes or component instances directly. `useRef` creates a mutable object that persists across renders, while `forwardRef` allows components to pass refs to their children. This is useful for imperative DOM operations (focus, measurements) that fall outside React's declarative model.

**Architectural Impact:**

- **Hook System:** Add `HOOK_TYPES.REF` with a simple mutable `.current` property
- **Reconciliation:** Need to attach DOM nodes to ref objects after commit
- **Component API:** `forwardRef` creates a special component type that receives ref as a prop

**Key Implementation Steps:**

1. Implement `useRef(initialValue)` that returns `{ current: initialValue }`
2. Store ref objects in hook memoized state
3. During commit phase, attach DOM nodes to ref `.current` property for host components
4. Create `forwardRef` function that wraps a component to accept ref as second parameter
5. Handle ref cleanup when components unmount (set `.current` to null)

**Relationship to Existing Architecture:** Refs are relatively simple—they're just persistent JavaScript objects attached to fibers. The main complexity is ensuring refs are updated at the correct time (after DOM mutations in the commit phase).

**Implementation Details:**

```javascript
function useRef(initialValue) {
  const hook = getHookInstance(HOOK_TYPES.REF);
  if (!hook.memoizedState) {
    hook.memoizedState = { current: initialValue };
  }
  return hook.memoizedState;
}

// In commitWork for host components:
if (fiber.ref) {
  fiber.ref.current = fiber.stateNode;
}
```

### 11. Memoization: React.memo, useMemo, useCallback

> **Mental Model:** Memoization is like a **calculator with a memory**. If you ask "what's 247 × 159?" it computes the answer (39,273). If you ask again with the same inputs, it remembers the previous answer instead of recalculating. `React.memo` remembers entire component renders, `useMemo` remembers values, and `useCallback` remembers functions.

**Description:** Memoization optimizes performance by avoiding expensive recalculations when inputs haven't changed. `React.memo` memoizes component renders, `useMemo` memoizes computed values, and `useCallback` memoizes function references to prevent unnecessary child re-renders.

**Architectural Impact:**

- **Reconciliation:** Need shallow comparison of props for `React.memo`
- **Hook System:** Add `HOOK_TYPES.MEMO` and `HOOK_TYPES.CALLBACK` with dependency tracking
- **Fiber Tree:** Memoized components get special tags to skip reconciliation when possible

**Key Implementation Steps:**

1. Implement `React.memo(Component, areEqual)` that wraps component with prop comparison
2. Add shallow comparison utility for default `areEqual` function
3. Create `useMemo(factory, deps)` that recomputes only when dependencies change
4. Implement `useCallback(fn, deps)` as syntactic sugar for `useMemo(() => fn, deps)`
5. During reconciliation, skip children reconciliation for memoized components when props equal

**Relationship to Existing Architecture:** Memoization hooks use the same dependency tracking mechanism as `useEffect`. `React.memo` is essentially a component-level optimization that sits on top of the existing reconciliation algorithm.

**Performance Trade-off:** Memoization has its own cost (comparison, storage). It should only be used when the computation cost exceeds the memoization overhead.

**Dependency Comparison:** Like `useEffect`, memoization hooks need reference equality checks for dependencies. A common pitfall is creating new arrays/objects in the dependency list, causing unnecessary recomputations.

### Summary Table of Future Extensions

| Feature | Complexity | Builds Upon | Key Benefit |
|---|---|---|---|
| `useReducer` | Low | `useState` infrastructure | Complex state transitions, predictable updates |
| Context API | Medium | Fiber tree propagation | Prop drilling elimination, global state |
| Concurrent Features | High | Fiber interruptibility | Responsive UI during heavy computation |
| Custom JSX Pragma | Low | `createElement` function | Compile-time optimizations, better tooling |
| Error Boundaries | Medium | Fiber tree, try-catch wrappers | Graceful error handling, partial failures |
| Portals | Medium | Reconciliation, commit phase | Modal/popover rendering outside hierarchy |
| Suspense | High | Fiber interruption, promise integration | Async rendering, code splitting |
| SSR Hydration | Medium | Reconciliation algorithm | Faster initial load, SEO benefits |
| Synthetic Events | High | Event system, prop handling | Cross-browser consistency, performance |
| Ref API | Low | Hook system, commit phase | Direct DOM access, imperative APIs |
| Memoization | Medium | Reconciliation, dependency tracking | Performance optimization, reduced re-renders |

### Implementation Guidance

While full implementation of all these features is beyond this design document's scope, here's a starting point for one of the simpler extensions: `useReducer`. This demonstrates how new features can be integrated into the existing hook architecture.

#### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| New Hook Types | Extend `HOOK_TYPES` enum with new values | Dynamic hook registration system |
| Update Queue | Reuse existing `Queue` and `Update` structures | Specialized queue types per hook type |
| Reducer Logic | Simple switch statement in hook | Middleware support, devtools integration |

#### B. Recommended File/Module Structure

Add new files for extended functionality:

```
my-react/
  src/
    core/
      index.js              # Main exports (add useReducer)
      createElement.js      # No changes
      render.js             # No changes
    reconciler/
      scheduler.js          # No changes
      workLoop.js           # No changes
      commitWork.js         # No changes
    hooks/
      index.js              # Export all hooks (add useReducer)
      useState.js           # No changes
      useEffect.js          # No changes
      useReducer.js         # NEW: useReducer implementation
      shared.js             # Extend with reducer hook helpers
    shared/
      constants.js          # Add HOOK_TYPES.REDUCER
      utils.js              # Add reducer-specific utilities
```

### C. Core Logic Skeleton Code

```javascript
// src/hooks/useReducer.js

/**
 * useReducer - Alternative to useState for complex state logic
 * @param {function} reducer - Pure function (state, action) => newState
 * @param {any} initialArg - Initial state or initializer argument
 * @param {function} [init] - Optional initializer function
 * @returns {[any, function]} - Current state and dispatch function
 */
function useReducer(reducer, initialArg, init) {
  // Get the current hook instance for this render
  const hook = getHookInstance(HOOK_TYPES.REDUCER);

  // TODO 1: Initialize hook state on first render
  // - If this is the first render (hook.memoizedState is null):
  //   - Calculate initialState: init ? init(initialArg) : initialArg
  //   - Store { state: initialState, reducer } in hook.memoizedState
  //   - Initialize empty queue if not exists

  // TODO 2: Process any queued updates (actions)
  // - While hook.queue.pending exists:
  //   - Get the next action from the update queue
  //   - Apply reducer: newState = reducer(currentState, action)
  //   - Update currentState
  //   - Mark fiber as needing re-render

  // TODO 3: Create stable dispatch function
  // - Return same function reference across renders
  // - Dispatch should:
  //   - Create new update with action
  //   - Add to hook's update queue
  //   - Schedule work on the fiber
  //   - Return void (dispatch doesn't return anything)
```

```
  // TODO 4: Return [state, dispatch]
  // - State is hook.memoizedState.state
  // - Dispatch is the stable function created above

  // Implementation hint: The structure is very similar to useState
  // but with reducer transformation between action and state
}

// src/hooks/shared.js - Extend getHookInstance to handle reducer hooks
function getHookInstance(hookType, initialValue) {
  // ... existing code ...

  // Add case for HOOK_TYPES.REDUCER
  if (hookType === HOOK_TYPES.REDUCER) {
    if (!currentHook) {
      // First call in this component - create hook instance
      const newHook = {
        memoizedState: null, // Will store { state, reducer }
        queue: null,         // Queue of action updates
        next: null,
        type: HOOK_TYPES.REDUCER
      };
      // TODO: Link into hook list, initialize state from initialValue
    }
    // Return existing or new hook
  }

  // ... rest of existing code ...
}

// src/shared/constants.js - Add new hook type
const HOOK_TYPES = {
  STATE: 0,
```

```
  EFFECT: 1,
  REDUCER: 2,  // NEW
  // Future: CONTEXT, REF, MEMO, CALLBACK
};
```

### D. Language-Specific Hints (JavaScript)

1. **Stable Function References:** Use `useMemo` or `useCallback` pattern (once implemented) to ensure `dispatch` function has stable identity across renders. For now, store it on the hook instance.
2. **Reducer Purity:** The reducer must be a pure function—no side effects. This matches React's expectation and enables predictable state updates.
3. **Action Objects:** While Redux convention uses `{ type: string, payload: any }` action objects, `useReducer` can work with any action format. Document expected conventions.
4. **Initializer Function:** The third parameter ( `init` ) is useful for lazy initialization, similar to `useState(() => initialValue)` .

### E. Testing Strategy for useReducer

Create test cases that verify:

1. Initial state is set correctly (with and without initializer function)
2. Dispatch updates state via reducer function
3. Multiple queued actions are processed in order
4. Dispatch function identity is stable across renders
5. Reducer errors are thrown (not caught by React)

Example test:

JAVASCRIPT

```javascript
// Simple counter reducer test
function reducer(state, action) {
  switch (action.type) {
    case 'increment': return state + 1;
    case 'decrement': return state - 1;
    default: return state;
  }
}

function Counter() {
  const [count, dispatch] = useReducer(reducer, 0);
  // Render count and buttons that dispatch actions
  // Test that clicking increments/decrements correctly
}
```

### F. Debugging Tips for Hook Extensions

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| State not updating | Reducer not processing actions | Log actions in reducer, check queue linkage | Ensure updates are being added to hook.queue |
| Infinite re-render loop | Dispatch called during render | Check for dispatch in render body or effect without dependencies | Move dispatch to event handler or effect with proper deps |
| Old state in dispatch callback | Stale closure in dispatch | Log state value inside dispatch function | Use functional update pattern or store latest state in ref |
| Reducer called multiple times | Multiple components sharing reducer | Check reducer function identity across renders | Memoize reducer with useCallback or define outside component |

## Conclusion

The extensions outlined above demonstrate how our foundational architecture provides numerous "extension points" for advanced functionality. Each feature builds logically upon concepts already present:

- **Hooks extensions** ( `useReducer` , `useContext` , `useRef` ) leverage the existing hook storage and update queue system
- **Performance features** (memoization, concurrent rendering) build upon Fiber's interruptible reconciliation
- **Rendering extensions** (portals, SSR) extend the commit phase logic
- **Error handling** (error boundaries) wraps the existing render pipeline

By implementing even a subset of these extensions, you'll gain deeper insight into how React itself evolves while maintaining backward compatibility and consistent mental models. The architecture we've built isn't just a React clone—it's a platform for exploring the frontiers of declarative UI programming.

---

# 11. Glossary

> **Milestone(s):** All (Milestone 1: Virtual DOM, Milestone 2: Reconciliation, Milestone 3: Fiber Architecture, Milestone 4: Hooks)

This glossary provides definitions for key terms, concepts, data structures, and functions used throughout this design document. Understanding this shared vocabulary is essential for implementing and extending the library.

## Terminology Reference

The following table organizes key terms alphabetically for quick reference:

| Term | Definition | Related Concepts |
| --- | --- | --- |
| **Alternate** | A pointer from a `Fiber` node in the work-in-progress tree to its counterpart in the current (committed) tree. This enables the reconciler to compare old and new trees during updates without losing the committed state. | Fiber, Work-in-progress tree, Reconciliation |
| **Commit Phase** | The atomic DOM mutation phase where all changes accumulated during the render phase are applied to the real DOM in a single batch. This phase cannot be interrupted and ensures the UI updates consistently. | Render Phase, Fiber, Work Loop |
| **Concurrent Rendering** | The ability to work on multiple UI versions with prioritization, allowing high-priority updates (like user input) to interrupt and preempt lower-priority updates (like background data fetching). In our implementation, this is simulated using `requestIdleCallback`. | Fiber, Work Loop, Scheduling |
| **Context** | A mechanism for sharing values across the component tree without having to explicitly pass props through every level (prop drilling). A `Context` object provides a `Provider` component that supplies a value and a `Consumer` (or `useContext` hook) that reads it. | ProviderFiber, useContext, Prop Drilling |
| **Declarative** | A programming paradigm where you describe *what* the desired UI state should be, rather than *how* to achieve it step-by-step. Our library uses a declarative API: developers return VNodes describing the UI, and the library determines the minimal DOM updates needed. Contrasts with Imperative. | Virtual DOM, Reconciliation |
| **Dependency Array** | An optional array of values passed to `useEffect` that determines when the effect should re-run. If provided, the effect only re-executes when any value in this array changes between renders. An empty array ( `[]` ) means the effect runs only once after the initial mount. | useEffect, Stale Closure |
| **Diffing** | Synonym for Reconciliation. The algorithmic process of comparing two trees to identify differences. | Reconciliation, Virtual DOM, Key Reconciliation |
| **DOMElement** | A native browser DOM element (e.g., `HTMLDivElement` , `SVGSVGElement` ). Our library creates and updates these real elements based on the Virtual DOM representation. | Virtual DOM, Render, setDOMProperty |
| **Effect** | A side effect (such as data fetching, subscriptions, or manual DOM mutations) that needs to run after a component renders. Effects are managed by the `useEffect` hook and are executed in the commit phase. | useEffect, Side Effect, Cleanup Function |
| **Error Boundary** | A React component that catches JavaScript errors anywhere in its child component tree, logs those errors, and displays a fallback UI instead of the component tree that crashed. While our implementation doesn't include error boundaries, the architecture could be extended to support them. | ErrorBoundaryComponent, getDerivedStateFromError |
| **EventPool** | A pool of reusable `SyntheticEvent` objects to avoid frequent allocation and garbage collection of event objects during event handling. | SyntheticEvent, Event Handling |
| **EventRegistry** | A global registry mapping event types (e.g., `'click'` ) to arrays of `{fiber, handler}` pairs. Used to manage event listener attachment and detachment during reconciliation. | SyntheticEvent, Event Handling, Reconciliation |
| **Fiber** | A unit of work in our renderer, representing a node in the component tree with additional scheduling links. Unlike simple VNodes, `Fiber` nodes form a linked list (via `child` , `sibling` , and `parent` pointers) that can be traversed incrementally, enabling interruptible rendering. | Work-in-progress tree, Alternate, Unit of Work |

| Term | Definition | Related Concepts |
| --- | --- | --- |
| **HookInstance** | A node in a linked list attached to a `Fiber`'s `memoizedState`. Each `HookInstance` stores the stateful data for a single hook call (like the current state value for `useState` or the effect function for `useEffect`). | Hooks, useState, useEffect, memoizedState |
| **Hooks** | Functions that let function components "hook into" state and lifecycle features. Hooks (like `useState` and `useEffect`) are stored as a linked list of `HookInstance` nodes on the component's `Fiber` and are called in the same order on every render. | Function Component, HookInstance, useState, useEffect |
| **Hydration** | The process of attaching event listeners and other client-side functionality to existing server-rendered DOM nodes, rather than creating them from scratch. This improves perceived performance. Our implementation does not include hydration, but the architecture could be extended. | Server-Side Rendering, DOMElement |
| **Imperative** | A programming paradigm where you explicitly describe *how* to achieve a result through step-by-step instructions. Direct DOM manipulation (`document.createElement`, `appendChild`, `setAttribute`) is imperative. Contrasts with Declarative. | DOM Manipulation, Declarative |
| **JSDOM** | A headless implementation of the DOM and web standards in Node.js, used in our testing strategy to run DOM-related tests without a browser. | Testing, DOM |
| **Key Reconciliation** | The process of efficiently reordering lists of child elements by using stable identifiers (`key` props). During reconciliation, elements with the same key are matched between renders, allowing for minimal DOM moves rather than recreations. | Reconciliation, Diffing, List Reordering |
| **Memoization** | An optimization technique where the result of an expensive computation is cached and reused when the inputs haven't changed. Our library could implement `useMemo` and `useCallback` using this technique. | useMemo, useCallback, Optimization |
| **Namespace** | The XML namespace for SVG or MathML elements. When creating SVG elements, the renderer must use the `SVG_NAMESPACE` (`'http://www.w3.org/2000/svg'`) to ensure proper browser behavior. | SVG, createElement, render |
| **Portal** | A component that renders its children into a different DOM subtree (outside the parent component's DOM hierarchy) while maintaining React context and event bubbling through the virtual tree. | createPortal, DOM Hierarchy |
| **Prop Diffing** | The process of comparing old and new props on a VNode or Fiber to determine which DOM attributes, styles, and event listeners need to be added, updated, or removed. Implemented in `updateDOMProperties`. | Reconciliation, setDOMProperty, removeDOMProperty |
| **Reducer** | A pure function that calculates the next state given the current state and an action. The pattern is used by `useReducer` for more complex state logic. | useReducer, State Management |
| **Ref** | A mutable object (created by `useRef`) with a `current` property that can hold any value. Commonly used to access DOM nodes or persist values across renders without triggering re-renders. | useRef, DOM Access, Mutable Value |
| **Reconciliation** | The process of comparing two Virtual DOM trees (the previous render and the new render) and calculating the minimal set of DOM operations needed to update the UI. Also called diffing. | Virtual DOM, Diffing, Fiber |
| **Render Phase** | The pure computation phase where VNodes are created (by function components) and the Fiber tree is reconciled (differences are identified). No DOM mutations occur during this phase, and work can be interrupted. | Commit Phase, Fiber, Work-in-progress tree |

| Term | Definition | Related Concepts |
| --- | --- | --- |
| **Snapshot Testing** | A testing technique where the output of a function or data structure is captured and compared to a stored reference snapshot. Useful for ensuring the VNode or Fiber tree structure doesn't change unexpectedly. | Testing, Jest |
| **Stale Closure** | A function closure that captures outdated values from a previous render. A common pitfall with `useEffect` and `useCallback` when the dependency array is incorrectly specified, causing the callback to reference old state or props. | useEffect, Dependency Array, Closures |
| **Suspense** | A React mechanism for handling asynchronous operations (like data fetching or code splitting) during render, allowing components to "wait" for something before rendering. | Concurrent Rendering, Lazy Loading |
| **SyntheticEvent** | A normalized cross-browser event wrapper that provides consistent properties and behavior across different browsers. Our library could implement a simple version to handle event listener attachment. | Event Handling, EventPool, EventRegistry |
| **Unit of Work** | A single `Fiber` node to be processed by the work loop. The `performUnitOfWork` function processes one fiber at a time, making the rendering interruptible. | Fiber, Work Loop, performUnitOfWork |
| **Virtual DOM** | An in-memory JavaScript object representation (tree of `VNode`s) of the real DOM tree. It's lightweight to create and compare, allowing the library to compute minimal updates before touching the slower real DOM. | VNode, Reconciliation, Declarative |
| **VNode** | A Virtual DOM node. A plain JavaScript object with `type`, `props`, `children`, and optionally `key` properties. Represents an element, text node, or function component in the virtual tree. | Virtual DOM, createElement, Render |
| **Work-in-progress tree** | The `Fiber` tree currently being constructed during the render phase. It's built incrementally as `performUnitOfWork` processes fibers, and it's eventually committed to become the new current tree. | Fiber, Alternate, Render Phase |

## Constants and Enum Reference

The following constants and enums are used throughout the codebase to tag fibers, effects, and hook types:

| Constant/Enum | Value/Description | Purpose |
| --- | --- | --- |
| `EFFECT_TAGS` | An enum object defining effect types for fibers: `PLACEMENT`, `UPDATE`, `DELETION`, `PLACEMENT_AND_UPDATE` | Tags fibers with the type of DOM mutation needed during commit. |
| `UPDATE_TYPES` | An enum for update types in hook queues: `STATE`, `EFFECT` | Distinguishes between state updates and effect scheduling. |
| `DOM_PROPERTIES` | A configuration object mapping special prop names (like `className`, `htmlFor`) to their DOM property equivalents (`class`, `for`). | Used by `setDOMProperty` and `removeDOMProperty` to handle prop-to-DOM attribute/property conversion. |
| `FIBER_TYPES` | An enum for fiber tags: `HOST` (DOM element), `FUNCTION` (function component), `CLASS` (class component), `TEXT` (text node), `PORTAL` | Identifies the type of component a fiber represents, guiding how it's processed. |
| `HOOK_TYPES` | An enum for hook types: `STATE`, `EFFECT`, `REDUCER`, `CONTEXT`, `REF`, `MEMO`, `CALLBACK` | Used internally to distinguish hook instances and their behavior. |
| `TEXT_ELEMENT` | A special type string (e.g., `'TEXT_ELEMENT'`) used for text VNodes. | Allows the renderer to treat text nodes differently from element nodes. |
| `NULL_COMPONENT` | A special symbol (e.g., `Symbol.for('null-component')`) representing a null or undefined component type. | Used as a placeholder for empty slots in the fiber tree. |
| `SVG_NAMESPACE` | The string `'http://www.w3.org/2000/svg'` | Passed to `document.createElementNS` when creating SVG elements. |
| `EFFECT_TAGS.CAPTURE_ERROR` | A numeric value (e.g., `7`) for tagging fibers that represent error boundaries that caught an error. | Used for error boundary implementation (future extension). |

### Data Structure Quick Reference

| Structure | Key Fields | Purpose |
|---|---|---|
| **`VNode`** | `type` (string\|Function), `props` (object), `children` (array), `key` (string\|number\|null), `_dom` (DOMElement\|null) | Lightweight virtual representation of a UI element or component. |
| **`Fiber`** | `type`, `key`, `props`, `stateNode` (DOMElement\|object\|null), `child`, `sibling`, `parent`, `alternate`, `effectTag`, `nextEffect`, `memoizedState` (HookInstance\|any), `updateQueue`, `tag`, `pendingProps`, `memoizedProps` | Unit of work with links for traversal, alternate for diffing, and effect tagging. |
| **`HookInstance`** | `memoizedState` (any), `queue` (Queue\|null), `next` (HookInstance\|null), `cleanup` (function\|null) | Node in a linked list storing hook-specific data (state, effect, etc.). |
| **`Update`** | `action` (any), `next` (Update\|null) | Node in a linked list of pending updates (e.g., state changes) for a hook. |
| **`Queue`** | `pending` (Update\|null) | Linked list head for pending updates on a hook. |
| **`Context`** | `value` (any), `Provider` (FunctionComponent), `Consumer` (FunctionComponent) | Object for creating and providing context values. |
| **`ProviderFiber`** | `contextValue` (any) | Extended fiber type for context provider fibers (stores the provided value). |
| **`SyntheticEvent`** | `nativeEvent` (Event), `type` (string), `target` (DOMElement), `currentTarget` (DOMElement) | Normalized event object. |
| **`EventRegistry`** | Map<string, Array<{fiber: Fiber, handler: Function}>> | Global map of attached event listeners for cleanup. |
| **`EventPool`** | Array | Pool of reusable synthetic event objects. |
| **`DeferredValueHook`** | `baseValue` (any), `deferredValue` (any) | Hook data structure for `useDeferredValue`. |
| **`ErrorBoundaryComponent`** | `getDerivedStateFromError` (function), `componentDidCatch` (function) | Extended component type for error boundaries. |

### Key Function Signatures

| Function | Signature | Purpose |
|---|---|---|
| **Core API** | `createElement(type, props, ...children) returns VNode` | Creates virtual nodes, normalizing children and handling falsey values. |
| | `render(vnode, container) returns void` | Mounts a virtual DOM tree into a real DOM container, handling namespaces. |
| **Hooks** | `useState(initialState) returns [state, setState]` | Hook for managing local component state with a setter that triggers re-render. |
| | `useEffect(effectFunction, dependencyArray) returns void` | Hook for scheduling side effects after commit, with cleanup. |
| **Reconciliation** | `reconcileChildren(currentFiber, newChildVNodes) returns void` | Core diffing algorithm that compares old and new child VNodes, creating fibers with effect tags. |
| | `updateDOMProperties(domElement, prevProps, nextProps) returns void` | Diffs and applies prop changes, handling event listener attachment/removal. |
| | `setDOMProperty(domElement, propName, propValue) returns void` | Sets a property on a DOM element, handling special cases like events and styles. |
| | `removeDOMProperty(domElement, propName, prevValue) returns void` | Removes a property, with special handling for event listener cleanup. |
| **Fiber Scheduler** | `performUnitOfWork(fiber) returns Fiber | null` | |
| | `workLoop(deadline) returns void` | Main loop that processes units of work during idle time using `requestIdleCallback`. |
| | `commitRoot() returns void` | Commits all finished work (effects) in the work-in-progress tree to the DOM. |
| | `commitWork(fiber) returns void` | Applies a fiber's effectTag (PLACEMENT, UPDATE, DELETION) to the DOM. |
| | `commitDeletion(fiber) returns void` | Handles deletion of a fiber and its subtree, calling cleanup functions. |
| | `scheduleIdleCallback(callback) returns void` | Schedules a callback for browser idle periods with a fallback to `setTimeout`. |
| **Fiber Creation** | `createFiber(vnode, parent, alternate) returns Fiber` | Creates a new fiber node for a given VNode, linking it to parent and alternate. |
| **Hook Runtime** | `beginHookDispatcher(fiber) returns void` | Sets up the hook dispatcher for a new function component render. |
| | `endHookDispatcher() returns void` | Cleans up the dispatcher after component render. |
| | `advanceToNextHook() returns void` | Advances the dispatcher to the next hook in the linked list. |
| | `getHookInstance(hookType, initialValue) returns HookInstance` | Gets or creates the hook instance for the current hook call. |
| | `createStateHook(initialState) returns HookInstance` | Creates a new hook instance for a state hook. |

| Function | Signature | Purpose |
|---|---|---|
| | `createEffectHook(effectFunction, deps) returns HookInstance` | Creates a new hook instance for an effect hook. |
| **Validation** | `validateKey(key, fiber) returns void` | Warns about duplicate keys in development mode (future extension). |
| **Future Extensions** | `useReducer(reducer, initialArg, init) returns [state, dispatch]` | Hook for reducer-based state management. |
| | `createContext(defaultValue) returns Context` | Creates a new context object. |
| | `useContext(context) returns any` | Hook to read the current context value. |
| | `startTransition(callback) returns void` | Marks updates as non-urgent for concurrent rendering. |
| | `useDeferredValue(value) returns any` | Returns a deferred version of a value that may lag behind. |
| | `createPortal(children, container) returns Portal` | Creates a portal for rendering children outside parent DOM hierarchy. |
| | `lazy(componentPromise) returns Component` | Creates a lazy-loaded component. |
| | `hydrate(vnode, container) returns void` | Attaches to server-rendered DOM (hydration). |
| | `useRef(initialValue) returns {current: any}` | Hook for mutable ref object. |
| | `forwardRef(component) returns Component` | Forwards a ref to a child component. |
| | `memo(Component, areEqual) returns Component` | Memoizes a component based on props. |
| | `useMemo(factory, deps) returns any` | Memoizes a computed value. |
| | `useCallback(fn, deps) returns function` | Memoizes a function reference. |