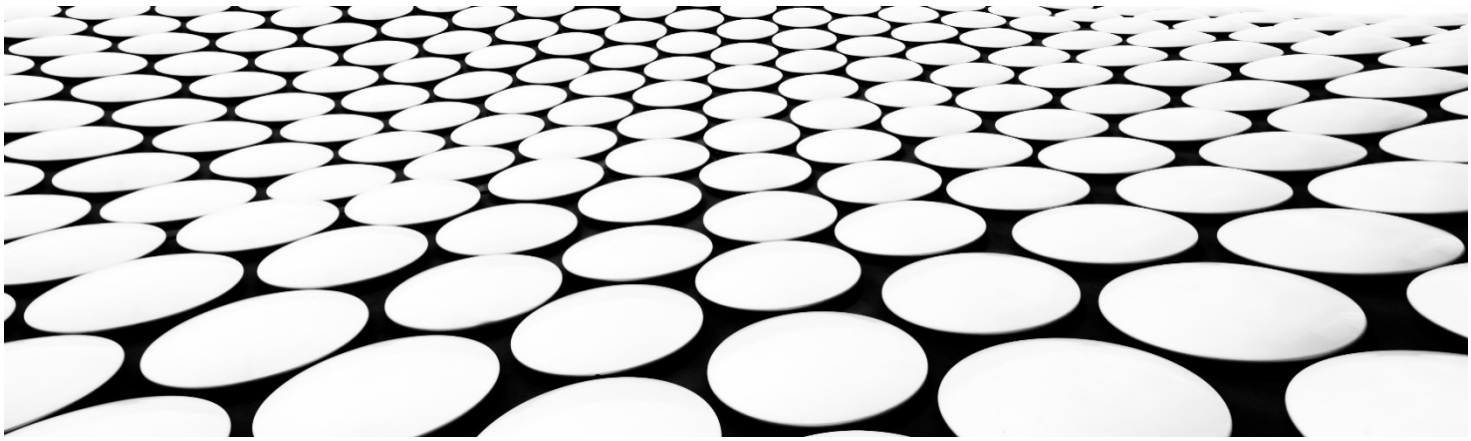
React Native Hookbook

ANDRONOTE STUDIOS

1ST EDITION

2026



*If you'd like to support our work, we'd greatly appreciate it if you clicked **here**, installed Andronote, and considered upgrading to the Pro version*

Copyright

Disclaimer

This book is intended for educational and informational purposes only.

All code examples, explanations, and diagrams are created to help readers understand React Native concepts, rendering behavior, and hooks. While every effort has been made to ensure the accuracy of the content, Andronote Studio makes no guarantees regarding completeness, reliability, or suitability for any particular purpose.

The author and publisher shall not be held responsible for any errors, omissions, or damages arising from the use of the information contained in this book.

Some diagrams and visual materials in this book were created or refined with AI-assisted tools for educational illustration purposes.

Preface

React Hooks are powerful, but they are also one of the most misunderstood parts of React and React Native development.

Many developers learn hooks by memorizing syntax without fully understanding how rendering, state, closures, and dependencies actually work. This often leads to bugs, unnecessary re-renders, stale closures, and performance issues.

This book was created to simplify these concepts through clear explanations, practical examples, and visual diagrams.

Rather than focusing only on API usage, this book focuses on mental models — how React thinks, how components re-render, and why hooks behave the way they do.

The goal is simple:

to help you reason about hooks, not just use them.

Who This Book Is For

This book is designed for:

- Beginner developers learning React Native
- Intermediate developers who want a deeper understanding of hooks
- React developers transitioning into React Native
- Developers struggling with re-renders, dependencies, and hook behavior
- Anyone who wants to build more predictable and efficient mobile applications

Basic JavaScript knowledge and familiarity with React fundamentals will help you get the most out of this book.

How to Use This Book

This book is structured progressively.

The early chapters build the mental foundation of React rendering, snapshots, identity, and closures. These concepts are essential before understanding hooks deeply.

The middle chapters focus on individual hooks such as:

useState

useEffect

useCallback

useMemo

useRef

useLayoutEffect

useReducer

Each chapter includes visual diagrams, conceptual explanations, and practical examples.

The final chapters focus on real-world React Native patterns, debugging strategies, and production-ready mental checklists.

For the best learning experience, it is recommended to read the chapters in order.

About Andronote Studios

Andronote Studios is an independent software and educational publishing brand focused on mobile application development, software engineering concepts, and practical programming resources.

Its goal is to simplify complex development topics through clear explanations, visual learning materials, and real-world examples.

Andronote Studios is also the creator of Andronote, an offline-first note-taking application built with React Native.

Contents

Chapter 1

How React Really Renders

To understand React Hooks, it is essential to first understand **how React renders**.
Most confusion around hooks does not come from the hooks themselves, but from incorrect assumptions about rendering, state, and time.

1.1 Rendering is not updating the screen

In React (including React Native), **rendering does not mean drawing pixels**.

Rendering means:

Running your component function to calculate what the UI *should look like*.

When React renders a component:

- It **calls the function**
- Executes all hooks **in order**
- Produces a **virtual description** of the UI

At this stage:

- No native views are touched
- No side effects run
- No layout happens

Rendering is **pure computation**.

1.2 The three phases of a render cycle

Every update in React goes through three conceptual phases:

1) Render phase

- Component functions run
- Hooks are evaluated
- JSX is returned

This phase must be:

- Pure
- Deterministic
- Free of side effects

2) Commit phase

- React applies changes to the native UI
- Views are created, updated, or removed
- Layout happens

This phase is synchronous with the UI.

3) Effect phase

- `useEffect` callbacks run
- Subscriptions start
- Async work begins

This phase happens **after** the UI is visible.

Hooks are tightly mapped to these phases:

- `useState, useMemo, useCallback` → render phase
- `useEffect` → effect phase
- `useLayoutEffect` → between commit and paint

1.3 A component is re-executed, not resumed

A critical concept:

React does **not** pause and resume your component — it **re-runs it from the top**.

Every render:

- Recreates local variables
- Recreates functions
- Re-evaluates expressions

Example:

```
function Screen() {
  const count = 0;
  return null;
}
```

On every render, count is **redeclared**.

So how does state persist?

👉 **Hooks store state outside the function**, indexed by call order.

This is why:

- Hooks must be called unconditionally
- Hook order must never change

1.4 Renders are snapshots in time

Each render is a **snapshot** of:

- Props
- State
- Variables
- Closures

Once a render finishes:

- Its values never change
- They become "frozen in time"

If a state update happens:

- React schedules a **new render**
- A **new snapshot** is created

This explains why:

```
setCount(1);
console.log(count); // old value
```

The `count` you log belongs to the **previous snapshot**.

1.5 Why this matters for hooks

Hooks are built on top of this model:

- `useState` selects which snapshot to render next
- `useEffect` runs *after* a snapshot is committed
- Closures capture values from the snapshot they were created in

Every hook rule exists to preserve this consistency.

If you understand rendering as **pure re-execution + snapshots**, most hook behavior becomes predictable.

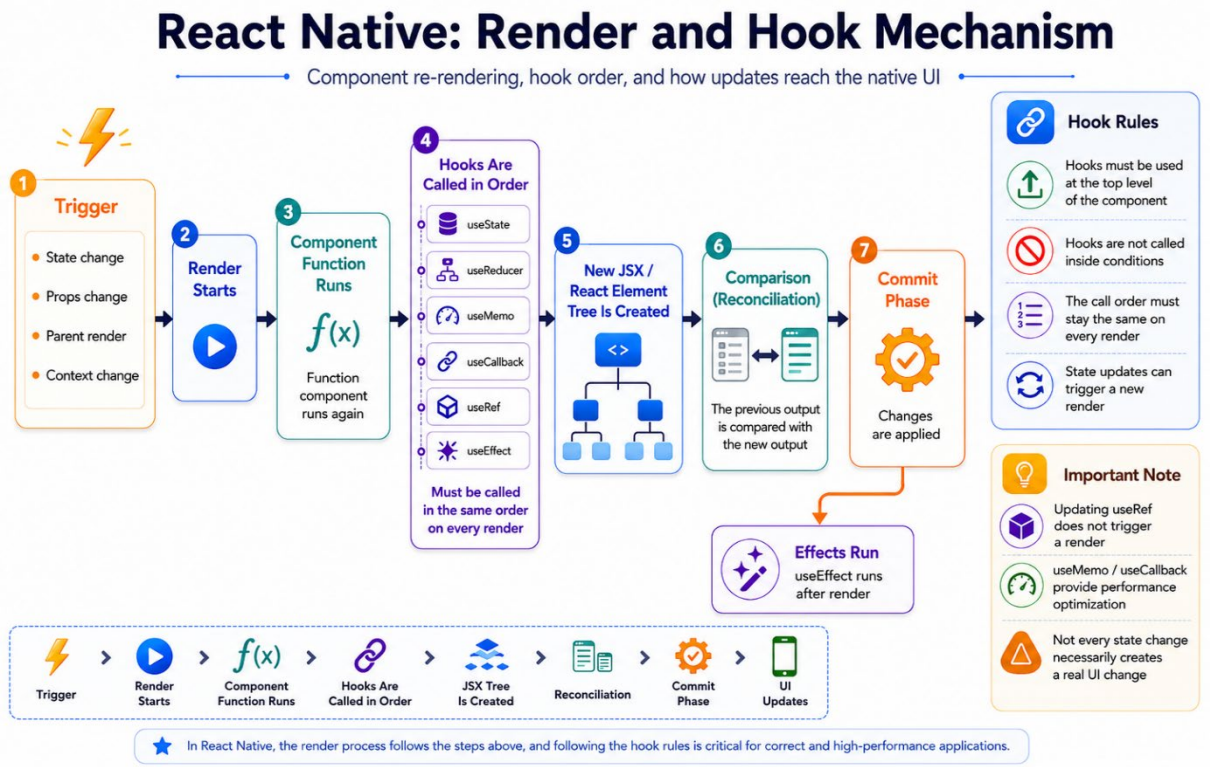


Figure 1.1 *Render and hooks mechanism in React Native.*

Chapter 4

useState: State, Scheduling, and Batching

`useState` is the foundation of all hook-based logic.
It is also the most commonly misunderstood hook.

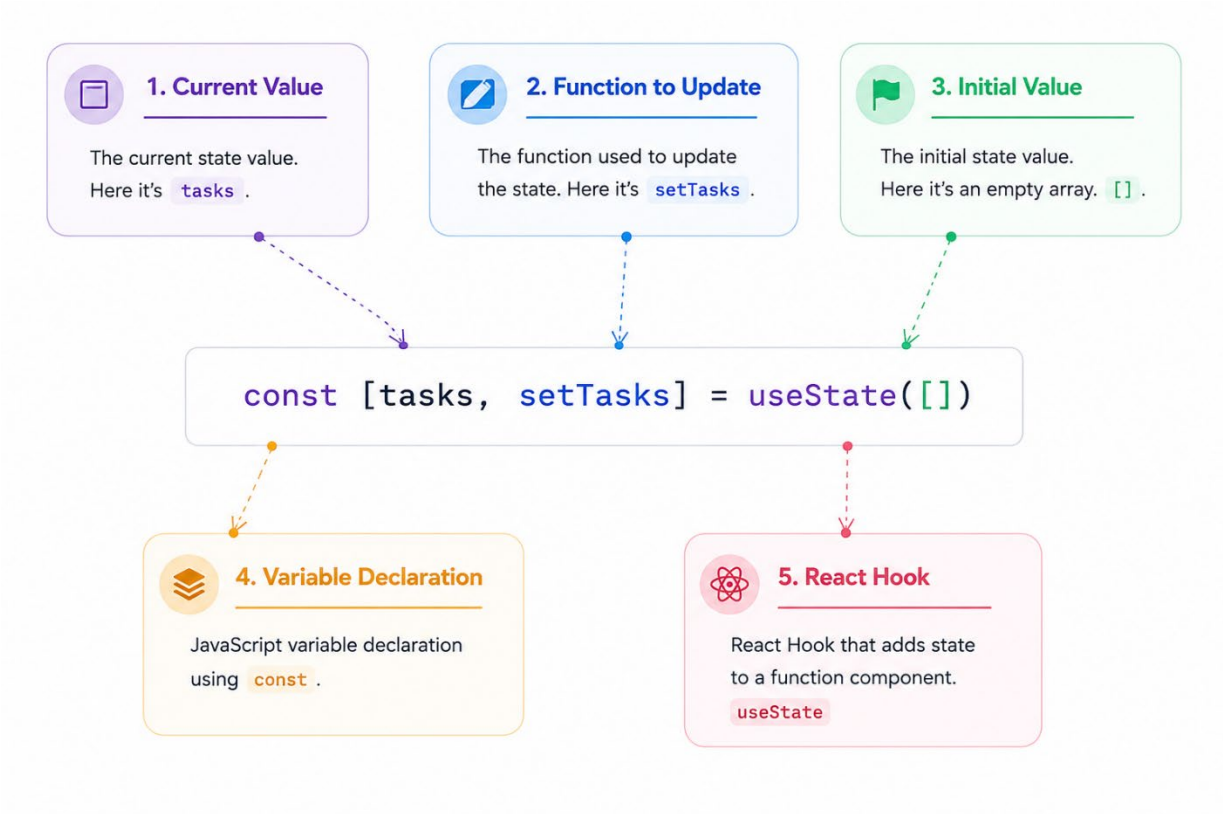


Figure 4.1 *useState variables declaration.*

4.1 What `useState` really does

```
const [value, setValue] = useState(initialValue);
```

This does **not** create a variable.
It creates:

- A state slot stored outside the component
- A setter that schedules a new render

The `initialValue`:

- Is used **only on the first render**
- Is ignored on subsequent renders

4.2 State updates are scheduled, not applied

Calling `setState` does not change the current value.
Instead:

1. React queues the update
2. React schedules a re-render
3. The next render receives the new value

This explains why:

```
setCount(1);
console.log(count); // old value
```

The log belongs to the **current snapshot**, not the next one.

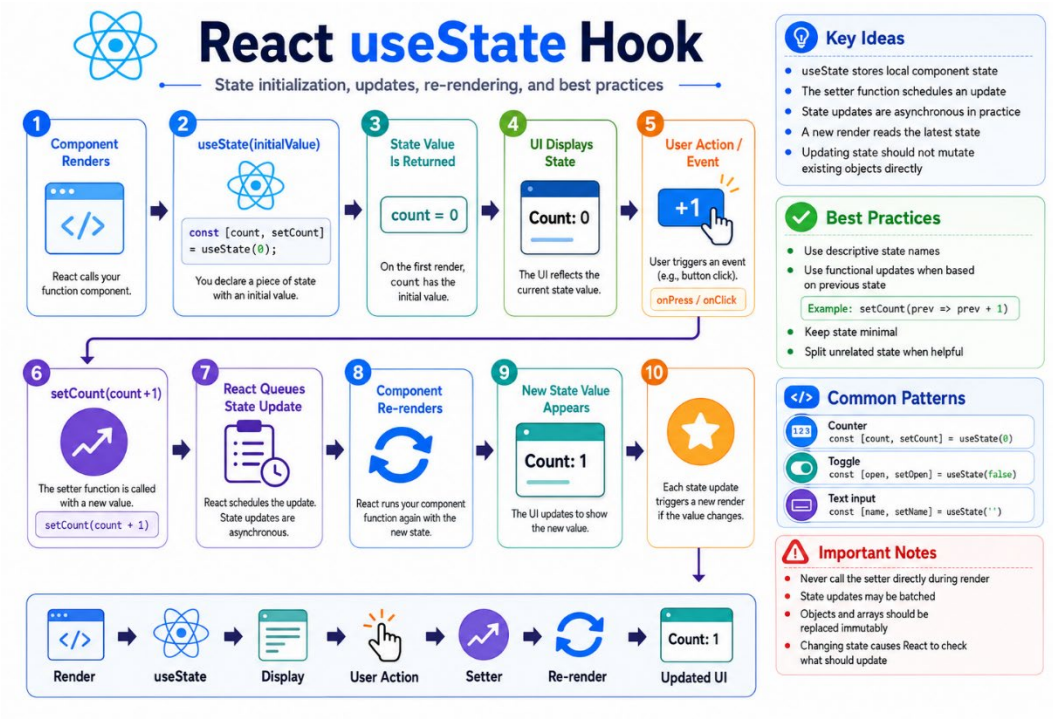


Figure 4.2 *useState hook mechanism in React Native.*

4.3 Batching: why updates appear "delayed"

React batches multiple updates together for performance.

```
setCount(1);
setCount(2);
```

In most cases:

- Only one render happens
- The last update wins

Batching:

- Reduces unnecessary renders
- Keeps UI responsive
- Enables concurrency

You should never rely on intermediate state during the same render cycle.

4.4 Functional updates: the correct mental model

When new state depends on old state, this is mandatory:

```
setCount(prev => prev + 1);
```

Why this works:

- React passes the **latest committed state**
- Independent of render timing
- Safe under batching and concurrency

This is not an optimization — it is **correctness**.

4.5 Objects and arrays: immutability is not optional

State identity matters.

❌ Incorrect:

```
state.value = 1;
setState(state);
```

Why this fails:

- Identity does not change
- React sees "same object"
- No meaningful update detected

✅ Correct:

```
setState(prev => ({
  ...prev,
  value: 1,
}));
```

Every update must:

- Create a new object or array
- Preserve unchanged fields
- Change identity intentionally

4.6 Why state sometimes "resets"

State resets when:

- A component unmounts
- A key changes
- A different component replaces it

React ties state to:
Component identity in the tree, not variable names.
Changing structure changes identity.
This is expected behavior — not a bug.

4.7 When not to use `useState`

Do not use `useState` for:

- Derived values → use computation
- Cached values → useMemo
- Mutable references → useRef
- Global state → context or external store

Overusing state increases re-render complexity.

4.8 `useState` in the lifecycle

In a typical screen:

- Initial render → default state
- Effect runs → async update
- State update → re-render
- UI reflects new snapshot

State drives rendering.
Effects respond to state changes.
Never invert this relationship.

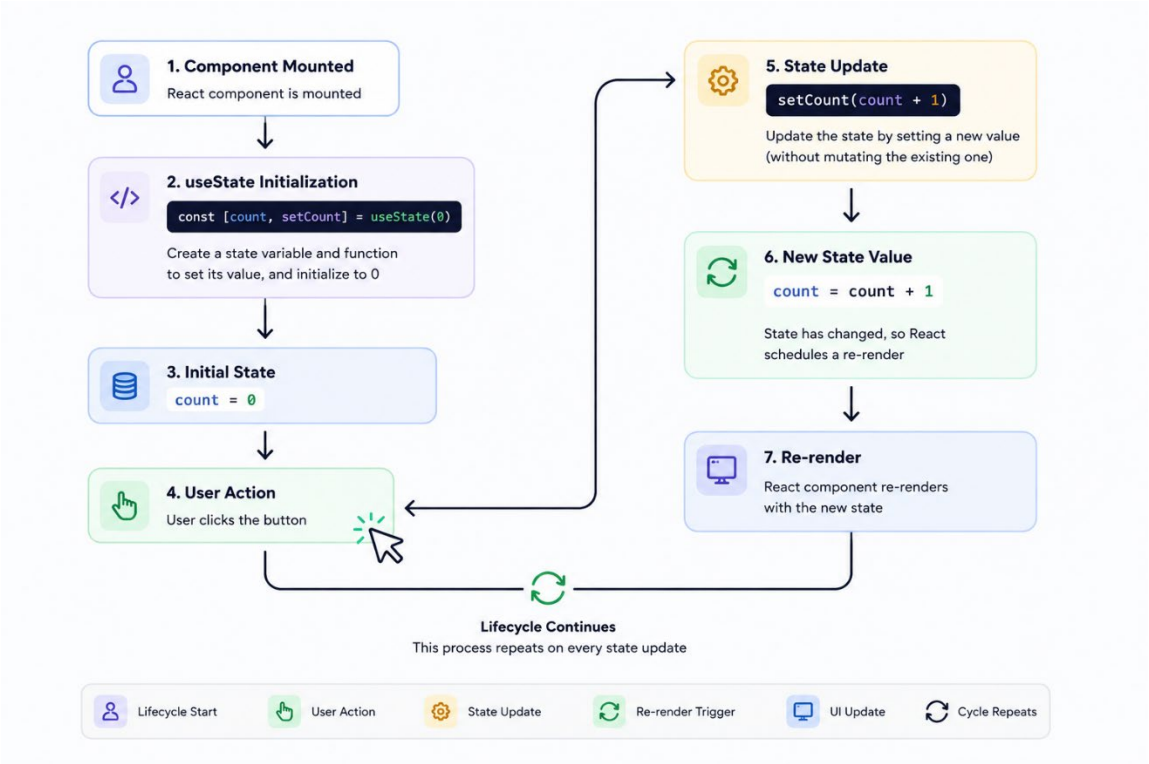


Figure 4.3 *How useState triggers re-renders.*