# Стратегический отчет: Архитектурная деконструкция React Hooks для инженеров Unity (Фаза III, День 3)

## 1. Введение: Эпистемологический сдвиг в управлении состоянием

Переход от разработки на Unity к экосистеме React требует не просто изучения нового синтаксиса, но фундаментального пересмотра ментальных моделей управления памятью, жизненным циклом и потоком данных. Для Senior Unity Developer, чье мышление сформировано детерминизмом Game Loop и объектно-ориентированными парадигмами C# (.NET CLR), архитектура React Hooks может показаться контринтуитивной или даже "магической". Однако при детальном рассмотрении через призму низкоуровневого системного программирования, Hooks обнаруживают строгую, детерминированную структуру, сравнимую с механизмами сериализации и управления памятью в игровых движках.

Данный документ представляет собой исчерпывающее техническое руководство, сфокусированное на третьем дне акселерационной программы перехода к Enterprise React Development. Цель этого отчета — демистифицировать механизм Hooks, отказавшись от упрощенных абстракций, принятых в веб-разработке, в пользу глубокого инженерного анализа. Мы будем рассматривать Hooks не как "функции для добавления стейта", а как интерфейс к сложной системе управления памятью Fiber, работающей по принципам, схожим с обработкой командных буферов (Command Buffers) и чтением бинарных потоков (Binary Streams).1

Центральная теза нашего исследования заключается в следующем: функциональный компонент React с хуками — это не просто функция, а проекция состояния, хранящегося во внешней структуре данных (Fiber Node), доступ к которой осуществляется через строгий протокол последовательного чтения/записи, аналогичный работе BinaryReader с потоком байтов. Понимание этой аналогии позволяет инженеру Unity мгновенно диагностировать сложные проблемы "Stale Closures" (устаревших замыканий) и нарушений правил хуков, видя в них не "особенности JS", а классические ошибки смещения указателя памяти или десинхронизации потоков.1

В этом документе мы проведем послойный анализ архитектуры, начиная от модели исполнения JS (Event Loop) в сравнении с Unity Player Loop, заканчивая глубоким разбором аллокаций памяти при создании замыканий в V8 по сравнению с генерацией DisplayClass компилятором Roslyn.

## 2. Среда исполнения: От Game Loop к Reconciliation Loop

Прежде чем анализировать сами хуки, необходимо установить контекст, в котором они существуют. Фундаментальное различие между Unity и React кроется в модели планирования вычислений.

### 2.1 Детерминизм против Реактивности

В Unity разработчик привык к **Game Loop** — бесконечному циклу while(true), который жестко оркестрирует фазы выполнения: Physics, Input, Update, Render. Это императивная модель "Push": движок проталкивает кадры, вызывая методы Update() у всех активных MonoBehaviour каждый кадр (обычно 60 раз в секунду). Состояние объекта (поля класса) персистентно и хранится в куче (Heap) до тех пор, пока жив сам объект GameObject.1

React работает иначе. В нем нет гарантированного вызова "Update" каждый кадр. React — это реактивная система "Pull", управляемая **Fiber Reconciler**. Движок "спит" до тех пор, пока не произойдет изменение состояния (State Change), которое помечает компонент как "грязный" (Dirty). Только тогда запускается процесс рендеринга.

Критическое отличие заключается в том, что в Unity MonoBehaviour — это **экземпляр класса** (Instance), имеющий собственный адрес в памяти и сохраняющий значения полей между кадрами. В React функциональный компонент — это **функция**, которая выполняется и завершается, полностью очищая свой стек (Stack Frame). Локальные переменные функции уничтожаются после возврата значения (JSX).

Возникает архитектурный парадокс: *Как функция может "помнить" состояние (например, здоровье игрока или текст в поле ввода) между вызовами, если она не имеет персистентного экземпляра?*

Ответ кроется в **Hooks**. Хуки — это механизм, позволяющий "подцепиться" (hook into) к внешней, долгоживущей структуре данных, которая существует вне скоупа функции. Эта структура называется **Fiber Node**.

### 2.2 Fiber Node как аналог Serialized Object

Каждый раз, когда React рендерит компонент <Player />, он не просто вызывает функцию Player(). За сценой он обращается к внутренней структуре данных — Fiber Node, которая сопоставлена этому компоненту в виртуальном дереве.

Для C# разработчика Fiber Node можно представить как экземпляр класса, который движок создает и удерживает в памяти, но который *не доступен* напрямую из вашего кода.

C#

// Концептуальная модель Fiber Node в терминах C#  
public class FiberNode {  
 // Уникальный идентификатор (аналог InstanceID в Unity)  
 public int InstanceId;  
   
 // Ссылка на функцию компонента  
 public Delegate ComponentFunction;  
   
 // Односвязный список хуков (аналог сериализованных данных)  
 public HookNode MemoizedState;   
   
 // Ссылки на дерево (аналог Transform.parent/child)  
 public FiberNode Return; // Parent  
 public FiberNode Child;  
 public FiberNode Sibling;  
}

Когда функция компонента вызывается, она получает доступ к этому FiberNode не через this, а неявно, через глобальный диспетчер, который "знает", какой компонент сейчас рендерится. Хуки служат интерфейсом для чтения и записи данных в поле MemoizedState этого узла.2

### 2.3 Модель памяти: Linked Lists и Cursor

Самое важное открытие для инженера, переходящего с Unity: хуки хранятся не в Hash Map (по имени) и не в массиве с произвольным доступом, а в **односвязном списке (Singly Linked List)**.

Внутри FiberNode поле MemoizedState указывает на первый хук в списке. Каждый хук имеет поле next, указывающее на следующий хук.

C#

// Структура узла хука (концептуально)  
public class HookNode {  
 public object MemoizedState; // Значение стейта (например, int health)  
 public object Queue; // Очередь обновлений (Pending Updates)  
 public HookNode Next; // Указатель на следующий хук  
}

При каждом рендере (вызове функционального компонента) React инициализирует внутренний курсор (Cursor) на начало этого списка.

1. Первый вызов useState(...) читает данные из первого узла списка и сдвигает курсор на node.Next.
2. Второй вызов useEffect(...) читает данные из второго узла и сдвигает курсор дальше.

Это поведение абсолютно идентично чтению бинарного файла с помощью BinaryReader в.NET. Если вы попытаетесь прочитать Int32, когда курсор стоит на позиции, где записан Boolean, вы получите мусор или ошибку десериализации. В React это приводит к нарушению целостности данных между рендерами.2

## 3. Деконструкция useState: Сериализация и Иммаутабельность

Хук useState является фундаментальным примитивом управления состоянием. В Unity мы привыкли к мутабельности: transform.position.x += 1. В React и современной архитектуре ПО (включая Unity DOTS) преобладает концепция иммутабельности.

### 3.1 Механизм работы: useState как Reader/Writer

Рассмотрим код компонента:

JavaScript

function Player() {  
 const [health, setHealth] = useState(100); // Хук 1  
 const [ammo, setAmmo] = useState(30); // Хук 2  
 //...  
}

С точки зрения внутренней реализации React (упрощенно на C#), это выглядит так:

C#

public static class ReactInternal {  
 // Глобальная переменная, указывающая на текущий обрабатываемый Fiber  
 private static FiberNode \_workInProgressFiber;  
 // Курсор, указывающий на текущий хук в списке  
 private static HookNode \_workInProgressHook;  
  
 public static T useState<T>(T initialValue) {  
 HookNode hook;  
   
 if (\_isFirstRender) {  
 // Фаза Mount: Создаем новый узел в списке  
 hook = new HookNode { MemoizedState = initialValue, Next = null };  
   
 if (\_workInProgressFiber.MemoizedState == null) {  
 \_workInProgressFiber.MemoizedState = hook;  
 } else {  
 \_lastHook.Next = hook;  
 }  
 \_lastHook = hook;  
 } else {  
 // Фаза Update: Читаем существующий узел  
 hook = \_workInProgressHook;  
 // СДВИГАЕМ КУРСОР!  
 \_workInProgressHook = \_workInProgressHook.Next;  
 }  
  
 return (T)hook.MemoizedState;  
 }  
}

Этот псевдокод демонстрирует критическую зависимость от порядка вызовов. Если в одном кадре вы вызовете useState(100), а в следующем — нет, то следующий за ним хук useState(30) получит данные от первого (значение 100), что приведет к катастрофической ошибке логики. Именно поэтому существуют **Rules of Hooks**: хуки нельзя вызывать внутри циклов, условий (if) или вложенных функций. Они должны всегда вызываться на верхнем уровне компонента в строго определенном порядке, чтобы "бинарный поток" данных совпадал с "схемой чтения".7

### 3.2 Аналогия с BinaryReader

Представьте, что Fiber Node — это файл сохранения на диске.

Render 1 (Запись/Инициализация):

React "записывает" структуру:

[Int32: 100] -> [Int32: 30] ->

Render 2 (Чтение/Обновление):

React "читает" структуру:

1. Вызов useState (Health): Читает первые 4 байта -> получает 100.
2. Вызов useState (Ammo): Читает следующие 4 байта -> получает 30.

Если вы обернете второй хук в if (isAlive) и isAlive станет false:

1. Вызов useState (Health): Читает первые 4 байта -> получает 100.
2. Следующий вызов (например, useEffect): Пытается прочитать следующие 4 байта (где лежит 30), но ожидает увидеть делегат эффекта.

Происходит InvalidCastException или логическая ошибка. В React это проявляется как смещение стейта, когда переменная ammo вдруг получает значение от useEffect или наоборот. Движок React обычно отлавливает несоответствие количества хуков и выбрасывает ошибку "Rendered fewer hooks than expected", но понимание механизма "курсора" дает инженеру ясное представление о природе этой ошибки.3

### 3.3 Иммаутабельность и Сравнение Ссылок

В Unity при работе со структурами (Vector3, Quaternion) мы часто меняем их поля напрямую, если они являются открытыми полями класса, или создаем новые, если они свойствами (так как свойства возвращают копию структуры).

В React стейт **иммутабелен**. Это означает, что hook.MemoizedState никогда не мутируется напрямую.

JavaScript

// НЕПРАВИЛЬНО: Мутация (как в Unity C#)  
const = useState({ hp: 100 });  
state.hp = 90; // React не узнает об этом изменении!

React использует Shallow Comparison (поверхностное сравнение) object.ReferenceEquals (в терминах C#), чтобы понять, нужно ли обновлять компонент.

oldState === newState.

Если вы мутируете объект по ссылке, oldState и newState будут указывать на один и тот же адрес в памяти. React посчитает, что изменений не было, и не запустит ре-рендер DOM.

**Правильный подход (Copy-on-Write):**

JavaScript

// ПРАВИЛЬНО: Создание новой ссылки  
setState(prevState => {  
 return {...prevState, hp: 90 }; // Spread operator создает новый объект  
});

Это аналогично работе со строками в C# (которые тоже иммутабельны) или присваиванию новой структуры:

transform.position = new Vector3(x, y, z);

Использование иммутабельности позволяет React избегать дорогостоящих глубоких проверок (Deep Equality Check), которые были бы убийственны для производительности в высокочастотных обновлениях UI. Сравнение указателей — это операция O(1).12

## 4. useEffect: Синхронизация и Жизненный Цикл

Хук useEffect является наиболее сложным для понимания Unity-разработчиками, так как он объединяет в себе концепции Start, Update, FixedUpdate, OnDisable и OnDestroy, но управляется не кадрами, а зависимостями данных.

### 4.1 Ментальная модель: Реактивная подписка

В Unity мы часто пишем код в Update, который проверяет изменение значения:

C#

// Unity: Ручная проверка изменений  
private int \_lastHealth;  
public int health;  
  
void Update() {  
 if (health!= \_lastHealth) {  
 UpdateHealthBar();  
 \_lastHealth = health;  
 }  
}

useEffect автоматизирует этот паттерн. Второй аргумент хука — массив зависимостей (dependency array) — это список значений, которые React должен проверять на изменения (Diffing) между рендерами.

JavaScript

useEffect(() => {  
 UpdateHealthBar();  
}, [health]); // Если health изменился с прошлого рендера -> выполнить эффект

### 4.2 Таблица соответствия жизненных циклов

Для быстрой адаптации используйте следующую таблицу трансляции концепций Unity в React:

| **Концепция Unity (MonoBehaviour)** | **Концепция React (useEffect)** | **Механизм** |
| --- | --- | --- |
| **Start() / Awake()** | useEffect(() => {... },) | Пустой массив зависимостей `` означает, что эффект не зависит ни от каких значений и выполняется только один раз после монтирования (Mount). |
| **OnDestroy()** | return () => {... } (внутри эффекта с ``) | Функция очистки (Cleanup function), возвращаемая из эффекта, вызывается перед размонтированием компонента (Unmount). |
| **Update()** | useEffect(() => {... }) | Отсутствие массива зависимостей означает выполнение после *каждого* рендера. |
| **OnValidate() / Property Setter** | useEffect(() => {... }, [prop]) | Реакция на изменение конкретного свойства или состояния. |
| **OnDisable()** | return () => {... } | Функция очистки вызывается перед *каждым* новым запуском эффекта (если зависимости изменились) или при уничтожении. |

.15

### 4.3 Очистка ресурсов и Утечки памяти

В Unity, если вы запускаете Coroutine в Start, вы обязаны остановить её (обычно в OnDisable или OnDestroy), иначе она продолжит выполняться, даже если объект будет уничтожен (если корутина запущена на другом объекте или через Static).

В React это еще более критично. Поскольку компоненты — это функции, которые могут вызываться сотни раз, создание подписки (например, setInterval или addEventListener) без очистки приведет к созданию сотен параллельных таймеров.

**Аналогия с Coroutine:**

JavaScript

useEffect(() => {  
 // Аналог StartCoroutine  
 const timerId = setInterval(() => {  
 console.log("Tick");  
 }, 1000);  
  
 // Аналог StopCoroutine / OnDisable  
 return () => {  
 clearInterval(timerId);  
 };  
},);

Если забыть return, каждый ре-рендер (если зависимости изменятся или их нет) будет запускать новый таймер, не останавливая старый. Через минуту работы приложения у вас будет 3600 тикающих таймеров, что приведет к утечке памяти и краху производительности, аналогичному спауну тысяч GameObject без их уничтожения.18

### 4.4 Проблема "Stale Closures" (Устаревшие замыкания)

Это самая коварная проблема, с которой сталкиваются C# разработчики, переходящие на JS. Она проистекает из различий в том, как C# и JS обрабатывают захват переменных в лямбда-выражениях.

В C# (Компилятор Roslyn):

Когда вы используете переменную в лямбде, компилятор создает скрытый класс (DisplayClass). Лямбда содержит ссылку на экземпляр этого класса. Если значение переменной меняется снаружи, лямбда видит новое значение, так как обращается к полю класса по ссылке.

В JavaScript (Движок V8):

Замыкание (Closure) захватывает лексическое окружение (Lexical Environment) в момент создания функции. Если переменная была примитивом (число, строка), она "запекается" в замыкании.

**Пример проблемы:**

JavaScript

function Timer() {  
 const [count, setCount] = useState(0);  
  
 useEffect(() => {  
 const id = setInterval(() => {  
 // ОШИБКА: Замыкание "видит" только count из ПЕРВОГО рендера (0)  
 console.log(count);   
 }, 1000);  
 return () => clearInterval(id);  
 },); // Запускаем эффект только один раз  
}

В этом примере, даже если count в стейте станет 100, таймер будет бесконечно печатать 0.

Почему? Потому что функция () => console.log(count) была создана в первом рендере. Она замкнула переменную count из скоупа первого вызова функции Timer, где count был равен 0. Поскольку массив зависимостей пуст ``, React не пересоздает эффект, и таймер продолжает крутить ту самую, "старую" функцию.

**Решение:**

1. Добавить count в зависимости: [count]. Это заставит React перезапускать эффект (сбрасывать старый таймер и создавать новый) каждый раз, когда меняется count. Новая функция замкнет новое значение.
2. Использовать функциональное обновление: setCount(prev => prev + 1). Это позволяет обновлять стейт, не читая его текущее значение из замыкания.21

## 5. useRef: Императивный "Escape Hatch"

В Unity у нас есть прямой доступ к полям класса. Мы можем хранить в них данные, изменение которых не требует перерисовки кадра (например, таймеры перезарядки, буферы данных). В React useState вызывает ре-рендер. Что если нам нужно сохранить данные *между* рендерами, но *не* обновлять UI?

Для этого используется useRef. С технической точки зрения, useRef возвращает мутабельный объект, который персистируется в Fiber Node на протяжении всей жизни компонента.

### 5.1 Аналогия с Private Fields

C#

// Unity C#  
public class Gun : MonoBehaviour {  
 // Влияет на отрисовку (UI) -> Аналог State  
 int \_ammoDisplay;   
   
 // Внутренняя логика, не требует немедленной перерисовки -> Аналог Ref  
 private float \_lastFireTime;   
}

JavaScript

// React JS  
function Gun() {  
 // Изменение вызовет ре-рендер  
 const [ammo, setAmmo] = useState(30);   
   
 // Изменение НЕ вызовет ре-рендер. Значение сохраняется.  
 const lastFireTime = useRef(0);   
  
 const fire = () => {  
 lastFireTime.current = Date.now(); // Прямая мутация!  
 setAmmo(a => a - 1);  
 };  
}

useRef — это ваш способ вернуть "старый добрый OOP" в функциональный мир. Вы можете мутировать ref.current как угодно, React это проигнорирует. Это идеально подходит для хранения ID таймеров, ссылок на WebSocket соединения или экземпляров сторонних библиотек (например, Chart.js).3

### 5.2 Доступ к DOM (аналог GetComponent)

Вторая роль useRef — получение доступа к реальным DOM-элементам. Это прямой аналог GetComponent<T>() или GameObject.Find.

React работает с Virtual DOM. Но иногда вам нужно измерить ширину элемента, установить фокус на input или интегрировать canvas.

JavaScript

const inputRef = useRef(null);  
  
useEffect(() => {  
 // Прямой доступ к API браузера (аналог native code)  
 if (inputRef.current) {  
 inputRef.current.focus();  
 }  
},);  
  
return <input ref={inputRef} />;

Здесь свойство ref в JSX говорит React: "Когда создашь реальный DOM-узлел для этого input, положи ссылку на него в inputRef.current".26

## 6. Продвинутая Архитектура: Custom Hooks

Сила Unity заключается в Компонентном подходе (Composition over Inheritance). Вы собираете Игрока из скриптов Health, Mover, InputHandler.

В React классовые компоненты часто страдали от сложности переиспользования логики (HOCs, Render Props). Хуки решают эту проблему элегантно.

**Custom Hook** — это просто JS-функция, имя которой начинается с use, и которая внутри себя вызывает другие хуки. Это позволяет инкапсулировать сложную логику (state + effects) в переиспользуемый модуль.

### 6.1 Аналогия со ScriptableObject и Components

Представьте, что вы пишете систему управления инвентарем.

В Unity вы бы создали класс InventorySystem (возможно, ScriptableObject или чистый C# класс), который управляет списком предметов.

В React вы создаете хук useInventory.

JavaScript

// useInventory.js (Логический компонент)  
function useInventory(initialItems) {  
 const [items, setItems] = useState(initialItems);  
  
 const addItem = (item) => setItems(prev => [...prev, item]);  
 const removeItem = (id) => setItems(prev => prev.filter(i => i.id!== id));  
  
 return { items, addItem, removeItem };  
}  
  
// Player.js (Визуальный компонент)  
function Player() {  
 // Инъекция логики - аналог GetComponent<Inventory>()  
 const { items, addItem } = useInventory();   
   
 return (  
 <div>  
 {items.map(i => <ItemView key={i.id} item={i} />)}  
 <button onClick={() => addItem({ id: 1, name: 'Sword' })}>Add Sword</button>  
 </div>  
 );  
}

Это позволяет полностью отделить **Логику** (State + Effects) от **Представления** (JSX). Вы можете использовать один и тот же хук useInventory в компоненте Player, Chest (Сундук) или Shop (Магазин). Это чистейшая реализация принципа единственной ответственности и повторного использования кода, к которой стремятся архитекторы Unity.28

## 7. Производительность и Мемоизация

Unity-разработчики одержимы производительностью: Garbage Collection, Draw Calls, FPS. В React аналогами этих проблем являются лишние ре-рендеры (Wasted Renders) и пересоздание объектов.

### 7.1 useMemo: Кэширование вычислений

В Unity мы избегаем тяжелых вычислений в Update. Мы делаем их в Start или кэшируем результат.

В React, так как тело функции выполняется при каждом рендере, тяжелые вычисления (сортировка большого массива, фильтрация данных) нужно мемоизировать.

JavaScript

// Аналог кэширования в Start()  
const sortedData = useMemo(() => {  
 return heavySortAlgorithm(data);  
}, [data]); // Пересчитать только если изменились data

Если data не изменилась (ссылочная целостность), useMemo вернет закэшированный результат, не запуская функцию сортировки. Это похоже на проверку "Dirty Flag" перед пересчетом путей навигации.30

### 7.2 useCallback: Стабильные делегаты

В C# создание new Action(MyMethod) аллоцирует память. Если делать это в Update, будет нагрузка на GC.

В JS объявление функции const handleClick = () =>... внутри компонента создает новый объект функции при каждом рендере.

Если этот handleClick передается в дочерний компонент (который обернут в React.memo для оптимизации), то дочерний компонент будет "думать", что пропсы изменились (так как ссылка на функцию новая), и будет перерендериваться зря.

useCallback сохраняет ссылку на функцию стабильной между рендерами.

JavaScript

// Стабильная ссылка на функцию (аналог кэшированного делегата)  
const handleClick = useCallback(() => {  
 console.log("Clicked");  
},);

Это критически важно при интеграции с библиотеками, которые зависят от ссылочной целостности колбэков, или для оптимизации списков (Virtual Lists).30

## 8. Архитектурная Таблица: Unity vs React (Сводка Дня 3)

| **Концепция C# / Unity** | **Концепция React Hooks** | **Комментарий по миграции** |
| --- | --- | --- |
| **Сериализованное поле** (``) | useState | Сохраняет данные между кадрами (рендерами). Вызывает обновление UI при изменении. |
| **Приватное поле** (private var) | useRef | Сохраняет данные без вызова обновления UI. "Escape hatch" к мутабельности. |
| **Start() / Awake()** | useEffect(...,) | Инициализация. Не забывайте про cleanup, если создаете подписки! |
| **OnDestroy()** | useEffect return func | Очистка ресурсов. Критично для таймеров и EventListeners. |
| **Update()** (реактивная часть) | useEffect(..., [deps]) | Выполняется при изменении зависимостей. Заменяет polling (опрос) на push-реактивность. |
| **Coroutine** | useEffect + Async/Await | Асинхронные операции. Осторожно с Race Conditions! |
| **ScriptableObject** (как контейнер логики) | **Custom Hook** | Вынос логики из View. Главный инструмент архитектора. |
| **Object Pooling** | useMemo / React.memo | Предотвращение лишних аллокаций и рендеров. |
| **GetComponent()** | useRef (для DOM) | Прямой доступ к нативным элементам. |

## 9. Заключение

Переход к использованию Hooks требует от инженера Unity принятия новой ментальной модели: **Компонент как Функция во времени**. Вместо статичного объекта, который *имеет* состояние, мы работаем с потоком вычислений, который *читает* состояние из временной шкалы (Fiber).

Понимание того, что хуки — это упорядоченный список, обрабатываемый курсором, устраняет "магию" и позволяет применять строгие инженерные практики. Правила хуков становятся не догмой, а логическим следствием архитектуры linked list. Проблема "Stale Closures" превращается из загадки в понятную особенность работы лексических областей видимости в долгоживущих процессах.

В практической части Дня 3 мы применим эти знания для создания "Ядра Календаря", используя useState для хранения даты, useEffect для загрузки событий и useMemo для оптимизации вычисления сетки дней. Эти навыки станут фундаментом для построения сложных Enterprise-приложений, где управление состоянием и жизненным циклом требует точности, сравнимой с разработкой физического движка игры.

**Следующий шаг:** Переход к практической реализации useCalendar (Модуль 3.3).

#### Источники

1. Интенсив React для Backend Разработчика неделя 10
2. Inside React's useState: A Deep Dive into the Hook's Internals | by Rupal Singhal | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@rupalsinghal/inside-reacts-usestate-a-deep-dive-into-the-hook-s-internals-be1b47f0999d>
3. Weird behavior with a BinaryReader - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/9232357/weird-behavior-with-a-binaryreader>
4. A deep dive into React Fiber - LogRocket Blog, дата последнего обращения: декабря 6, 2025, <https://blog.logrocket.com/deep-dive-react-fiber/>
5. Linked Lists: The Hidden Power Behind React Hooks - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/talissoncosta/linked-lists-the-hidden-power-behind-react-hooks-4kfg>
6. Understanding the React Hooks API: A Behind-the-Scenes Look - FullStack Labs, дата последнего обращения: декабря 6, 2025, <https://www.fullstack.com/labs/resources/blog/behind-the-scenes-react-hooks-api>
7. React hooks: not magic, just arrays | by Rudi Yardley - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e>
8. Why Do React Hooks Rely on Call Order? - Overreacted.io, дата последнего обращения: декабря 6, 2025, <https://overreacted.io/why-do-hooks-rely-on-call-order/>
9. Breaking the Rules of React Hooks: Behind the Scenes | by Crystal Chang | Flatiron Labs, дата последнего обращения: декабря 6, 2025, <https://medium.com/flatiron-labs/breaking-the-rules-of-react-hooks-9e892636641e>
10. c# - Get Encoding of BinaryReader/Writer? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/29435814/get-encoding-of-binaryreader-writer>
11. c# - POST string to ASP.NET Web Api application - returns null - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/13771032/post-string-to-asp-net-web-api-application-returns-null>
12. c# - Immutable class vs struct - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/14147340/immutable-class-vs-struct>
13. The Importance of Immutability in React: A Guide with Examples | by aysunitai | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@aysunitai/the-importance-of-immutability-in-react-a-guide-with-examples-c5dc253947d2>
14. Understanding Mutable vs. Immutable Data in React and Their Impact on Rendering, дата последнего обращения: декабря 6, 2025, <https://dev.to/muthuraja_r/understanding-mutable-vs-immutable-data-in-react-and-their-impact-on-rendering-ldc>
15. Lifecycle of Reactive Effects – React, дата последнего обращения: декабря 6, 2025, <https://react.dev/learn/lifecycle-of-reactive-effects>
16. When Should You Use useEffect Instead of React Lifecycle Methods? - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@conboys111/when-should-you-use-useeffect-instead-of-react-lifecycle-methods-866de1037e38>
17. Deep Dive into useEffect: The Hook That Handles React Lifecycle Like a Pro [Part-24] | by Jaswanth Kumar | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@jaswanth_270602/deep-dive-into-useeffect-the-hook-that-handles-react-lifecycle-like-a-pro-dc1566c2020b>
18. Why is the cleanup function from `useEffect` called on every render? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/57023074/why-is-the-cleanup-function-from-useeffect-called-on-every-render>
19. Understanding React's useEffect cleanup function - LogRocket Blog, дата последнего обращения: декабря 6, 2025, <https://blog.logrocket.com/understanding-react-useeffect-cleanup-function/>
20. Scripting API: MonoBehaviour.StopCoroutine - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/MonoBehaviour.StopCoroutine.html>
21. What is the difference between a 'closure' and a 'lambda'? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/220658/what-is-the-difference-between-a-closure-and-a-lambda>
22. Demystifying Closures in JavaScript: Understanding Memory Allocation and Lexical Scope | by Pallavi Ganpat Babar | Women in Technology | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/womenintechnology/demystifying-closures-in-javascript-understanding-memory-allocation-and-lexical-scope-13f39db11dc0>
23. C# Delegates and Memory Allocations: Lambdas - Matt Gibson, дата последнего обращения: декабря 6, 2025, <https://www.mattgibson.dev/blog/csharp-delegates-memory-lambdas>
24. Closing over the loop variable considered harmful, part two | Fabulous adventures in coding, дата последнего обращения: декабря 6, 2025, <https://ericlippert.com/2009/11/16/closing-over-the-loop-variable-considered-harmful-part-two/>
25. React Hooks in Simple Terms — With Real-Life Analogies | by Pallavi | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@emailmepallavi/react-hooks-in-simple-terms-with-real-life-analogies-cf7962c5426b>
26. Built-in React Hooks, дата последнего обращения: декабря 6, 2025, <https://react.dev/reference/react/hooks>
27. BEE-spoke-data/code-tutorials-en · Datasets at Hugging Face, дата последнего обращения: декабря 6, 2025, <https://huggingface.co/datasets/BEE-spoke-data/code-tutorials-en/viewer/default/train>
28. A Guide to React Custom Hooks - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/rasaf_ibrahim/a-guide-to-react-custom-hooks-2b4h>
29. Item System: ScriptableObjects vs Custom Classes? : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/a5em35/item_system_scriptableobjects_vs_custom_classes/>
30. Top 7 React Hooks you must know - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/vishnusatheesh/top-7-react-hooks-you-must-know-3k7g>
31. Advanced C# Tips: Prefer Structs for Immutable Data | Coding and Beyond, дата последнего обращения: декабря 6, 2025, <https://www.codingandbeyond.com/2024/06/27/advanced-c-tips-prefer-structs-for-immutable-data/>
32. I heard react save hooks state in list and use key as index. Is it true? : r/reactjs - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/reactjs/comments/1fy8zpb/i_heard_react_save_hooks_state_in_list_and_use/>