

Клиентский стейт-менеджмент против серверного кэширования в React: сравнительный анализ архитектурных паттернов (Flux, Atomic, Singleton)

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлен сравнительный анализ подходов к управлению состоянием в современных React-приложениях: классического паттерна Flux (Redux Toolkit), атомарного стейт-менеджмента (Zustand), паттерна Singleton (кастомный Context) и специализированного серверного кэширования (TanStack Query). В рамках тестового стенда, моделирующего нагрузку типичного дашборда с конкурентными запросами и поллингом, проведены замеры количества перерисовок компонентов, размера итогового бандла, количества сетевых запросов и потребления оперативной памяти. Результаты показывают, что выделение серверного состояния в TanStack Query снижает количество дублирующих сетевых запросов на 66% и оптимизирует перерисовки, тогда как атомарный подход демонстрирует наименьший размер бандла и высокую эффективность для чисто клиентского UI-состояния. Паттерн Singleton признан антипаттерном для реактивного UI из-за проблем с синхронизацией обновлений. Сформулированы практические рекомендации по гибридной архитектуре стейт-менеджмента.

Ключевые слова: React, стейт-менеджмент, серверный кэш, Flux, Atomic, TanStack Query, производительность, перерисовки, архитектурные паттерны.

Client-side state management vs. server caching in React: comparative analysis of architectural patterns (Flux, Atomic, Singleton)

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of state management approaches in modern React applications: the classic Flux pattern (Redux Toolkit), atomic state management (Zustand), the Singleton pattern (custom Context), and specialized server caching (TanStack Query). Within a test bench simulating the load of a typical dashboard with concurrent requests and polling, measurements were taken of component re-renders, final bundle size, number of network requests, and RAM consumption. Results show that extracting server state into TanStack Query reduces duplicate network requests by 66% and optimizes re-renders, while the atomic

approach demonstrates the smallest bundle size and high efficiency for purely client-side UI state. The Singleton pattern is identified as an anti-pattern for reactive UI due to update synchronization issues. Practical recommendations for a hybrid state management architecture are formulated.

Keywords: React, state management, server cache, Flux, Atomic, TanStack Query, performance, re-renders, architectural patterns.

1 Введение

1.1 Актуальность

С развитием экосистемы React и внедрением Concurrent Mode в React 18/19, парадигма управления состоянием претерпела значительные изменения. Исторически сложилось, что глобальные стейт-менеджеры использовались для хранения как UI-состояния (открытые модальные окна, темы), так и серверных данных (списки пользователей, результаты API-запросов). Однако такой монолитный подход приводит к избыточным перерисовкам, усложнению логики инвалидации кэша и дублированию сетевых запросов при конкурентном доступе нескольких компонентов к одним и тем же данным. Современная индустрия движется в сторону разделения ответственности: специализированные библиотеки для серверного кэширования и легковесные решения для клиентского состояния. Количественное сравнение этих подходов необходимо для обоснованного выбора архитектуры, балансирующей между размером бандла, скоростью отклика и сложностью поддержки.

1.2 Обзор исследований

Проблемы оптимизации производительности React-приложений подробно рассматриваются в работе Е.А. Ратушняк, где проводится комплексный анализ современных методов оптимизации, включая мемоизацию, разделение кода и управление состоянием, подчёркивается важность выбора правильных инструментов для повышения скорости отклика [1]. Архитектурные подходы к разработке одностраничных приложений исследуются М.А. Романовым и соавторами, которые проводят сравнительный анализ паттернов SPA, SSR и SSG, демонстрируя влияние выбора архитектуры на производительность и поддержку [2]. С.О. Бондаренко исследует построение пользовательских интерфейсов с использованием React, анализируя компонентную архитектуру и способы организации интерактивных веб-приложений [3]. Вопросы сравнения архитектур управления состоянием рассматриваются в работе А.В. Свищёва и В.А. Кнышова, где проводится детальное сравнение паттернов Flux и Redux для клиентских веб-приложений, подчёркиваются преимущества и недостатки каждого подхода в контексте сложности поддержки и предсказуемости потоков данных [4]. Е.П. Кравцов исследует методы и практики оптимизации высокопроизводительных React-приложений, уделяя особое внимание управлению состоянием, разделению кода и серверному рендерингу как ключевым факторам производительности [5].

Фундаментальные методы построения и кэширования веб-приложений рассматриваются в работе М.В. Булгакова и В.П. Носова, где анализируются различные подходы к серверному и клиентскому кэшированию, демонстрируется влияние выбора стратегии кэширования на общую производительность системы [6].

1.3 Цель исследования

Целью исследования является количественное сравнение паттернов Flux (Redux Toolkit), Atomic (Zustand), Singleton (кастомный Context + внешний класс) и Server-cache (TanStack Query v5) по метрикам: количество перерисовок компонентов, размер production-бандла, количество сетевых запросов при конкурентном чтении и потребление памяти, с формулированием рекомендаций по гибридной архитектуре.

2 Материалы и методы

1. Тестовый стенд развернут в среде Docker (образ node:20-alpine). Сборка выполнялась через Vite 5. Аппаратная конфигурация: Intel Core i7-12700H, 32 ГБ ОЗУ. Моделируемый сценарий: дашборд, содержащий 3 независимых виджета, которые одновременно запрашивают одни и те же данные пользователя при монтировании, а также обновляют их каждые 5 секунд (polling). Реализованы четыре подхода с фиксированной бизнес-логикой и одинаковым мок-API сервером с задержкой 150 мс. Измеряемые метрики: количество перерисовок (фиксируется React DevTools Profiler при загрузке и одном цикле обновления), размер бандла (gzipped-артефакт через rollup-plugin-visualizer), количество реальных HTTP-запросов при одновременном монтировании виджетов, изменение потребления памяти (Heap Used) после 10 циклов polling'a.

3 Результаты и обсуждения

Реализация атомарного подхода на базе Zustand отличается минималистичной архитектурой (рис. 1). Стор создается одной функцией, а компоненты подписываются только на конкретные поля через селекторы, что исключает каскадные перерисовки и обеспечивает обновление только зависимых частей дерева компонентов.

```
import { create } from 'zustand';

interface UserState {
  user: { id: number; name: string } | null;
  fetchUser: () => Promise<void>;
}

export const useUserStore = create<UserState>((set) => ({
  user: null,
  fetchUser: async () => {
    const response = await fetch('/api/user');
    const data = await response.json();
    set({ user: data });
  },
}));
```

Рис 1. Фрагмент реализации атомарного стейт-менеджмента на Zustand

Классический паттерн Flux в исполнении Redux Toolkit требует строгой конфигурации провайдеров, слайсов и thunk-функций для асинхронных операций (рис. 2). Это обеспечивает предсказуемый однонаправленный поток данных, но увеличивает объем бойлерплейта и сложность поддержки при работе с сетевыми запросами.

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

export const fetchUser = createAsyncThunk('user/fetch', async () => {
  const response = await fetch('/api/user');
  return response.json();
});

const userSlice = createSlice({
  name: 'user',
  initialState: { data: null, status: 'idle' },
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchUser.fulfilled, (state, action) => {
      state.data = action.payload;
      state.status = 'succeeded';
    });
  },
});

export default userSlice.reducer;
```

Рис 2. Фрагмент реализации паттерна Flux на Redux Toolkit

Пардигма серверного кэширования (TanStack Query) кардинально меняет подход: состояние привязывается не к глобальному объекту, а к уникальному ключу запроса (рис. 3). Библиотека автоматически управляет дедупликацией, stale-таймерами и фоновым обновлением, что устраняет необходимость ручной реализации логики кэширования в клиентском коде.

```
import { useQuery } from '@tanstack/react-query';

export function useUser() {
  return useQuery({
    queryKey: ['user'],
    queryFn: async () => {
      const response = await fetch('/api/user');
      return response.json();
    },
    refetchInterval: 5000,
    staleTime: 4000,
  });
}
```

Рис 3. Фрагмент реализации серверного кэширования на TanStack Query

Результаты измерений сведены в таблицу (табл. 1).

Анализ результатов выявляет четкое разделение зон ответственности между клиентскими стейт-менеджерами и специализированными серверными кэшами. TanStack Query показал абсолютное превосходство в работе с серверными данными. Благодаря механизму дедупликации, при одновременном монтировании трех виджетов был выполнен только один сетевой запрос, в то время как остальные подходы инициировали три

независимых запроса. Кроме того, количество перерисовок при обновлении данных составило всего 1, так как библиотека оптимизирует распространение обновлений только на компоненты, активно использующие данный queryKey. Потребление памяти умеренное, что объясняется хранением метаданных кэша и состояний загрузки.

Таблица 1 – Сравнительные характеристики подходов к управлению состоянием

Паттерн / Библиотека	Размер бандла (gzipped), КБ	Сетевые запросы (3 виджета)	Перерисовок на обновление	Потребление памяти (Δ), МБ
Flux (Redux Toolkit)	45.2	3	4	12.5
Atomic (Zustand)	3.2	3	2	4.1
Singleton (Context)	0.8	3	5	3.8
Server-cache (React Query)	14.7	1	1	5.2

Zustand продемонстрировал наилучший баланс для чисто клиентского UI-состояния. Размер бандла пренебрежимо мал, а механизм селекторов гарантирует, что перерисовываются только зависимые компоненты. Однако для работы с серверными данными Zustand требует ручной реализации логики кэширования, дедупликации и обработки состояний загрузки, что значительно увеличивает объем кода и риск ошибок.

Redux Toolkit показал ожидаемо высокий размер бандла и избыточное количество перерисовок. Это связано с тем, что при обновлении стора React вынужден проверять всех подписчиков, что создает определенный оверхед даже при использовании useSelector. Потребление памяти также наибольшее, что объясняется внутренней структурой стора и возможностью сохранения истории действий. Тем не менее, этот паттерн остается эталоном для сложных сценариев с множеством синхронных трансформаций состояния, где предсказуемость архитектуры важнее производительности.

Singleton показал наименьший размер бандла, но наихудшие результаты по перерисовкам. Это связано с тем, что обновление внешнего класса не интегрировано нативно в жизненный цикл React, что вынуждает использовать принудительные триггеры или подписки на кастомные события, которые часто срабатывают избыточно и приводят к рассинхронизации UI. Таким образом, попытка использовать клиентские стейт-менеджеры для хранения серверных данных является антипаттерном, так как они не обладают встроенными механизмами дедупликации,

инвалидации по времени и фокусной ревалидации, что напрямую ведет к перегрузке сети и деградации пользовательского опыта.

4 Выводы

1. Серверное состояние должно управляться специализированными инструментами. TanStack Query снижает количество дублирующих сетевых запросов на 66% и минимизирует перерисовки за счет привязки состояния к ключам запросов.

2. Атомарный подход является оптимальным выбором для клиентского UI-состояния, обеспечивая наименьший размер бандла и гранулярный контроль над перерисовками без избыточного бойлерплейта.

3. Паттерн Flux оправдан только в крупных проектах со сложной бизнес-логикой, требующей строгого аудита действий и предсказуемости потоков данных.

4. Паттерн Singleton в связке с React Context является антипаттерном для реактивного управления состоянием из-за риска рассинхронизации внешнего экземпляра класса с циклом рендеринга и избыточных перерисовок.

5. Рекомендуемая гибридная архитектура: использование TanStack Query для всех данных, получаемых по сети, и Zustand для локального UI-состояния, не требующего персистентности на сервере.

Библиографический список

1. Ратушняк Е.А. Анализ современных методов оптимизации в React // Программные системы и вычислительные методы. 2025. № 3. С. 1-9. URL: <https://cyberleninka.ru/article/n/analiz-sovremennyh-metodov-optimizatsii-v-react> (дата обращения: 14.06.2026).
2. Романов М.А., Бадалин А.О., Рыбалка Д.В., Баженов А.Э. Анализ и сравнительное исследование архитектурных подходов к разработке одностраничных приложений (SPA, SSR, SSG) // Программные системы и вычислительные методы. 2025. № 4. С. 108-117. URL: <https://cyberleninka.ru/article/n/analiz-i-sravnitelnoe-issledovanie-arhitekturnyh-podhodov-k-razrabotke-odnostranichnyh-prilozheniy-spa-ssr-ssg> (дата обращения: 14.06.2026).
3. Бондаренко С.О. Современные интерактивные веб-приложения - построение пользовательского интерфейса с React // Вестник науки и образования. 2018. № 5 (41). С. 46-50. URL: <https://cyberleninka.ru/article/n/sovremennye-interaktivnye-veb-prilozheniya-postroenie-polzovatel'skogo-interfeysa-s-react> (дата обращения: 14.06.2026).
4. Свищёв А.В., Кнышов В.А. Сравнение Flux и Redux архитектуры клиентских веб-приложений // Международный журнал гуманитарных и естественных наук. 2022. № 11-2. С. 121-124. URL: <https://cyberleninka.ru/article/n/sravnenie-flux-i-redux-arhitektury-klientskih-veb-prilozheniy> (дата обращения: 14.06.2026).

5. Кравцов Е.П. Разработка высокопроизводительных React-приложений: методы и практики оптимизации // European science. 2024. № 1 (69). С. 53-58. URL: <https://cyberleninka.ru/article/n/razrabotka-vysokoproizvoditelnyh-react-prilozheniy-metody-i-praktiki-optimizatsii> (дата обращения: 14.06.2026).
6. Булгаков М.В., Носов В.П. Исследование и разработка методов построения и кэширования веб-приложений // Открытое образование. 2008. № 2. С. 28-40. URL: <https://cyberleninka.ru/article/n/issledovanie-i-razrabotka-metodov-postroeniya-i-keshirovaniya-veb-prilozheniy> (дата обращения: 14.06.2026).