national Journal of Computer Science Issues. – 2016. – Vol. 13, Issue 6. – P. 32–40. – URL: https://ijcsi.org/papers/IJCSI-13-6-32-40.pdf. – Дата доступа: 09.02.2026.

4. Попеня Н.В., Романенко Д. В. Метод аудиостеганографии для AAC-сжатых аудиосигналов на основе эхо-кодирования и адаптивного кепстрального анализа // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2025. № 2 (296). С. 110–119.

5. Попеня Н.В. Методика подготовки и структура авторских данных для защиты видеофайла методами стеганографии // Труды БГТУ. Сер. 4, Принт- и медиатехнологии. 2025. № 2 (297). С. 71–77.

УДК 004.43

М.Ф. Кудлацкая, доц., канд. техн. наук
(БГТУ, г. Минск)

УПРАВЛЕНИЕ ГЛОБАЛЬНЫМ СОСТОЯНИЕМ В REACT-ПРИЛОЖЕНИЯХ

В течение длительного периода стандартом для управления данными в React-приложениях являлась библиотека Redux. Это сформировало архитектурную парадигму, предполагающую хранение всех типов данных – от бизнес-сущностей до состояния UI-компонентов – в едином глобальном дереве состояния (Store). Однако эволюция React, в частности внедрение Hooks API и Context API, а также рост сложности клиентских интерфейсов, выявили ряд системных ограничений монолитного подхода [1]:

– избыточность шаблонного кода (Boilerplate);

– накладные расходы на производительность;

– нарушение принципа разделения ответственности.

Ключевым трендом современной frontend-разработки стало четкое разграничение двух категорий данных: состояния сервера (Server State) и состояния клиента (Client State).

1. Client State: синхронные, локальные данные, полностью контролируемые клиентским приложением (темы оформления, состояние модальных окон, данные форм).

2. Server State: асинхронные данные, персистентно хранящиеся на удаленном сервере. Клиент владеет лишь их проекцией, которая может устареть в любой момент времени.

Попытки управлять серверным состоянием посредством универсальных менеджеров (например, Redux) требуют ручной реализации механизмов загрузки, кэширования, дедупликации запросов и инвалидации данных.

Внедрение библиотек класса TanStack Query (React Query) позволяет делегировать управление серверным состоянием специализированным инструментам. Данные библиотеки автоматизируют: кэширование и фоновую синхронизацию данных, управление жизненным циклом запроса (loading, error, success), дедупликацию сетевых запросов. Исключение серверных данных из глобального UI-хранилища позволяет сократить объем кода, отвечающего за управление состоянием, на 30–50% [2].

После делегирования работы с серверными данными специализированным утилитам, возникает вопрос выбора инструмента для управления оставшимся UI-состоянием. Ниже приведен сравнительный анализ ключевых решений.

1. React Context API. Нативное решение, встроенное в библиотеку React. Преимуществом является отсутствие внешних зависимостей, интеграция на уровне ядра React. Однако есть проблемы с производительностью при частых обновлениях. Изменение значения контекста провоцирует повторный рендеринг всех компонентов-потребителей [3], даже если им необходима лишь часть данных. Применяется для редко изменяемых данных (темы, локализация, данные авторизованного пользователя). Не рекомендуется для высокочастотных обновлений.

2. Redux Toolkit (RTK). Индустриальный стандарт, адаптированный под современные требования Developer Experience. RTK нивелировал проблему избыточного кода за счет внедрения createSlice и интеграции библиотеки Immer для упрощения работы с иммутабельными данными [2]. К преимуществам относятся предсказуемость потока данных, мощные инструменты отладки, развитая экосистема, наличие RTK Query. Недостатком является высокий порог входа и концептуальная сложность, часто избыточная для небольших и средних проектов.

3. Zustand. Легковесная библиотека, реализующая минималистичный подход к управлению состоянием. Упрощенный менеджер состояния, работающий на базе хуков и не требующий оборачивания приложения в провайдеры контекста. Имеет лаконичный API, полную поддержку TypeScript, возможность использования вне React-компонентов. Оптимальный выбор для большинства проектов средней сложности, где функционал Redux избыточен.

4. Recoil. Решение от Meta, разработанное для высоконагруженных интерфейсов. Состояние моделируется не как дерево, а как ориентированный граф. Узлами графа выступают Атомы (единицы состояния) и Селекторы (чистые функции преобразования). Ключевой осо-

бенностью является автоматический пересчет производных данных. При изменении атома селектор обновляет значение, и перерисовка происходит точечно только в подписанных компонентах. Используется в приложениях со сложной взаимозависимостью данных (графические редакторы, дашборды, электронные таблицы), где критически важна гранулярность обновлений [4].

Для демонстрации различий в подходах (Flux против Atomic) рассмотрим реализацию одной задачи: хранение профиля пользователя и обновление его имени.

1. RTK. Централизованный, строгий подход, основан на действиях и редьюсерах. Требует соблюдения строгой архитектуры (Slice, Dispatch, Selector), что обеспечивает унификацию кода в больших командах.

```
const userSlice = createSlice({
  name: 'user',
  initialState: { name: 'Anonymous', age: 0 },
  reducers: {
    updateName: (state, action) => {
      state.name = action.payload; // Мутация через Immer
    },
  },
});

const UserComponent = () => {
  const name = useSelector((state) => state.user.name);
  const dispatch = useDispatch();
  return <input value={name} onChange={(e) =>
      dispatch(updateName(e.target.value))} />;
};
```

Рисунок 1 – Пример использования RTK

2. Zustand. Децентрализованный, ориентированный на хуки. Минимальный объем кода. Логика обновления инкапсулирована вместе с данными.

```
const useUserStore = create((set) => ({
  user: { name: 'Anonymous', age: 0 },
  updateName: (name) => set((state) => ({
    user: { ...state.user, name }
  })),
}));

// 2. Использование
const UserComponent = () => {
  const { user, updateName } = useUserStore();
  return <input value={user.name} onChange={(e) =>
updateName(e.target.value)} />;
};
```

Рисунок 2 – Пример использования Zustand

3. Recoil. Атомарный, распределенный подход. API схож с useState. Обеспечивает максимальную производительность рендеринга при масштабировании количества полей состояния.

```
export const userNameState = atom({
  key: 'userNameState',
  default: 'Anonymous',
});

// 2. Использование
const UserComponent = () => {
  const [name, setName] = useRe-
coilState(userNameState);
  return <input value={name} onChange={(e) =>
      setName(e.target.value)} />;
};
```

Рисунок 3 – Пример использования Recoil

Таким образом, выбор инструмента управления состоянием должен быть продиктован спецификой конкретных задач: для работы с серверными данными стандартом является TanStack Query, в то время как для реализации глобальной логики в обычных приложениях оптимально подходит Zustand, а в крупных Enterprise-системах – Redux Toolkit.

В то же время Context API остается лучшим решением для простых задач вроде темизации, тогда как высокоинтерактивные интерфейсы со сложными зависимостями, такие как дашборды или редакторы, требуют использования атомарных библиотек Recoil, минимизирующих лишние рендеры за счет своей архитектуры.

ЛИТЕРАТУРА

1. React / Optimizing Performance [Электронный ресурс]. – URL: https://legacy.reactjs.org/docs/optimizing-performance.html. – Дата доступа: 03.02.2026.

2. Redux Toolkit / Usage Guide: Why RTK is Redux Today [Электронный ресурс]. – URL: https://redux-toolkit.js.org/usage/usage-guide. – Дата доступа: 03.02.2026.

3. React / Performance Testing with the Profiler [Электронный ресурс]. – URL: https://react.dev/reference/react/Profiler. – Дата доступа: 03.02.2026.

4. Meta Engineering / Recoil: State Management for Complex Interfaces [Электронный ресурс]. – URL: https://engineering.fb.com/2020/05/14/video/recoil-state-management-for-react/. – Дата доступа: 03.02.2026.