МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, КОМПЛЕКСОВ И КОМПЬЮТЕРНЫХ СЕТЕЙ/MATHEMATICAL SOFTWARE FOR COMPUTERS, COMPLEXES AND COMPUTER NETWORKS

DOI: https://doi.org/10.60797/itech.2025.8.2

МЕТОДЫ УПРАВЛЕНИЯ СОСТОЯНИЕМ В МОБИЛЬНЫХ ПРИЛОЖЕНИЯХ НА REACT NATIVE

Научная статья

Рамазанов И.М.[1, *]

[1] Photon Infotech, Лос-Анджелес, Соединенные Штаты Америки

* Корреспондирующий автор (geek6124[at]gmail.com)

Аннотация

В статье рассматриваются методы управления состоянием в мобильных приложениях, разработанных с использованием React Native. Цель работы заключается в анализе существующих методов решения этой задачи, определении инструментов для различных типов проектов.

Рассматриваются инструменты, такие как React Context API, Redux, MobX, Recoil, Zustand. В работе уделяется внимание не только архитектурным особенностям этих решений, но и их влиянию на производительность, удобство разработки. Также анализируются ситуации, когда легкие решения, такие как Zustand, являются подходящими для конкретных задач.

Результаты работы показывают, что выбор метода управления состоянием зависит от факторов: масштаба приложения, сложности взаимодействий между компонентами, требований к быстродействию. Для небольших проектов с ограниченным количеством состояний эффективными будут решения, основанные на контексте или Zustand. В крупных проектах, где требуется высокая гибкость в управлении состоянием, предпочтительнее использовать инструменты, такие как Redux, MobX. Также рассматриваются способы оптимизации работы с состоянием для повышения производительности, предотвращения утечек памяти.

Статья будет полезна разработчикам мобильных приложений на React Native, инженерам, занимающимся оптимизацией производительности существующих проектов. Она интересна преподавателям, исследователям, работающим в области мобильной разработки, изучающим методы управления состоянием.

Ключевые слова: управление состоянием, React Native, Redux, MobX, Recoil, Zustand, производительность, мобильные приложения, контекст API.

STATE MANAGEMENT METHODS IN MOBILE APPLICATIONS BASED ON REACT NATIVE

Research article

Ramazanov I.M.[1, *]

[1] Photon Infotech, Los Angeles, USA

* Corresponding author (geek6124[at]gmail.com)

Abstract

The article examines methods of state management in mobile applications developed using React Native. The aim of the work is to analyse existing methods for solving this problem and to identify tools for different types of projects.

Tools such as React Context API, Redux, MobX, Recoil, and Zustand are reviewed. The paper focuses not only on the architectural features of these solutions, but also on their impact on performance and development convenience. Situations where lightweight solutions such as Zustand are suitable for specific tasks are also analysed.

The results show that the choice of state management method depends on factors such as the scale of the application, the complexity of interactions between components, and performance requirements. For small projects with a limited number of states, context-based or Zustand solutions will be effective. In large projects where high flexibility in state management is required, it is preferable to use tools such as Redux and MobX. Ways to optimise state management to improve performance and prevent memory leaks are also discussed.

This article will be useful for mobile app developers working with React Native and engineers involved in optimising the performance of existing projects. It will also be of interest to teachers and researchers working in the field of mobile development who are studying state management methods.

Keywords: state management, React Native, Redux, MobX, Recoil, Zustand, performance, mobile applications, context API.

Введение

В последние годы мобильная разработка привлекла широкий интерес благодаря распространению фреймворков, таких как React Native. Эти инструменты позволяют создавать кросс-платформенные приложения с использованием одного кода для разных операционных систем, что ускоряет разработку. Важнейшей задачей при создании мобильных приложений является управление состоянием, которое регулирует взаимодействие данных между компонентами интерфейса. Ошибки в управлении состоянием могут привести к снижению производительности, появлению багов, проблемам с масштабируемостью.

Мобильные приложения становятся всё более комплексными, что увеличивает требования к решениям в области управления состоянием. Существует множество инструментов, включая контекст в React, Redux, MobX, Recoil,

Zustand. Каждый из них имеет свои характеристики, которые необходимо учитывать при выборе подхода в зависимости от особенностей проекта.

При выборе подходящего инструмента для управления состоянием важно обеспечить баланс между производительностью и архитектурой. Существенным является определение подходящего метода в зависимости от масштаба проекта, сложности взаимодействий между его компонентами.

Цель работы заключается в анализе существующих методов решения этой задачи, определении инструментов для различных типов проектов.

Методы и принципы исследования

Научные работы в области управления состоянием в мобильных приложениях на платформе React Native охватывают несколько направлений: анализ производительности различных методов, оптимизацию взаимодействия данных, архитектурные решения, внедрение технологий синхронизации, распределённого управления состоянием. Важно отметить практическую значимость предложенных решений, направленных на повышение эффективности приложений в реальных условиях.

Научная работа Pronina D., Kyrychenko I. [1], а также статья Olexandr B., Olexandr K. [2] посвящены сравнению различных методов управления состоянием, таких как Redux, React Hooks. Pronina D., Kyrychenko I. [1] изучают эти инструменты в контексте объёма данных, частоты их обновлений, выделяя ключевые особенности каждой технологии. Работа Olexandr B., Olexandr K. [2] ориентированы на кроссплатформенные приложения, где ключевой задачей является снижение количества рендеров, использование мемоизации, что оказывает влияние на производительность в условиях ограниченных ресурсов мобильных устройств.

Вопрос выбора архитектуры для управления состоянием стал предметом внимания ряда научных работ. Например, Qianqian L., Yuxiao D. A [3] рассматривают классические паттерны, такие как MVC, Flux, Redux, их применение в крупных приложениях. Donvir A., Jain A., Saraswathi P. K. [4] расширяют этот анализ, включая локальные, серверные подходы, что актуально для мобильных, веб-приложений. Эти работы помогают понять, какие архитектурные решения обеспечат гибкость, масштабируемость при разработке крупных систем. В статье L.Oleshchenko [8] проводится сравнение библиотек Redux, MobXState-Tree, Recoil, акцентируется внимание на их удобстве, простоте настройки, масштабируемости при управлении состоянием, что делает эти решения подходящими для динамичных проектов.

Практическая направленность также важна для понимания применения теоретических подходов в реальных условиях. Ralph Aran Cañon Cabañero [5] описывают создание системы управления инцидентами, раскрывая особенности реализации функций управления состоянием. Saputro D. F., Gunawan D. [12] изучают разработку POS-приложений, фокусируются на оптимизации состояния, производительности. Эти примеры показывают, как эффективное управление состоянием влияет на отклик системы, что подтверждает значимость правильного подхода для коммерческих приложений.

Задачи синхронизации состояния между клиентом, сервером рассмотрены в работе Tagdiwala V. et al. [7], предлагающих фреймворк для синхронизации, обеспечивающий низкую задержку при взаимодействии с серверными данными. В статье Toka L. et al. [6] изучаются методы минимизации задержек при доступе к данным в облачных приложениях, что важно для обеспечения быстрого отклика систем. В статье Xu Y. et al. [9] описываются платформы FaaS, анализируется возможность распределённого управления состоянием, что обеспечит гибкость архитектуры, взаимодействия компонентов.

Подходы, основанные на реактивном программировании, рассматриваются в статье Levkowitz S. [11]. В ней описываются методы, которые делают управление состоянием более структурированным, предсказуемым. Это позволяет улучшить разработку сложных приложений, где необходимо учитывать постоянные изменения состояния, реагировать на них в режиме реального времени.

Работа авторов Rahman A., Rahman A. S., Hakim M. [10] акцентируеьт внимание на педагогических аспектах создания мобильных приложений. Авторы обсуждают важность выбора подходящей архитектуры на ранних этапах проектирования, что имеет значение для образовательных проектов, где внимание к методическим особенностям разработки важно.

Проблемы масштабирования управления состоянием в распределённых системах, интеграции с серверless-архитектурами остаются малоизученными. Также недостаточно рассмотрены реальные кейсы оптимизации состояния в условиях высокой нагрузки, динамических обновлений данных. Эти вопросы открывают перспективы для дальнейших научных работ, направленных на создание эффективных решений в области управления состоянием мобильных приложений.

Для достижения поставленной цели использована методология, включающая теоретический анализ литературы.

Научной новизной является предложение нового взгляда на процесс управления состоянием в мобильных приложениях на REACT NATIVE, за счет применения современных инструментов, что, в свою очередь, стало возможным благодаря проведенному анализу литературы.

Впервые в работе проведён сравнительный анализ как классических архитектурных решений (например, Redux, Context API), так и легковесных методов (Zustand, React Hooks) с учётом специфики реальных условий эксплуатации мобильных устройств. Такой междисциплинарный подход позволяет не только выявить узкие места в традиционных схемах обновления состояний, но и предложить новые алгоритмы синхронизации, способствующие снижению затрат вычислительных ресурсов, повышению отзывчивости интерфейса и уменьшению частоты ненужных ререндеров.

Далее, архитектурное решение включает использование промежуточного слоя (middleware), который обеспечивает синхронизацию локальных и глобальных состояний, а также реализацию распределённого управления данными. Применение реактивных паттернов в сочетании с традиционными архитектурными подходами (например, MVC или Flux) позволяет разработчикам оптимально распределить вычислительные ресурсы, снизить задержки при выполнении асинхронных операций и упростить процесс отладки за счёт централизованного мониторинга изменений

состояния. Такой подход не только повышает производительность за счёт уменьшения количества повторных рендеров, но и позволяет оперативно адаптироваться к изменяющимся требованиям проекта, сохраняя целостность и консистентность данных на всех уровнях приложения.

Наконец, практическая значимость предложенных архитектурных решений заключается в их способности создавать устойчивую и легко масштабируемую платформу для разработки мобильных приложений на React Native. Четкая структуризация кода и разделение функциональных обязанностей обеспечивают не только высокую производительность, но и упрощают поддержку и дальнейшую эволюцию приложения. Экспериментальное подтверждение эффективности предложенного подхода, основанное на сравнительном анализе временных характеристик операций и потребления памяти, демонстрирует, что внедрение технологий синхронизации и распределённого управления состоянием способствует созданию более масштабируемых и устойчивых мобильных приложений.

В рамках эксперимента, каждое из трёх действий: добавление, удаление и фильтрация было выполнено при двух объёмах нагрузки (100 и 1000 задач) с проведением повторных измерений (более 10 раз) для каждого сочетания параметров. Тестирование осуществлялось на современных мобильных устройствах на базе Android и iOS с использованием фреймворка React Native. Время отклика пользовательского интерфейса фиксировалось встроенным API performance.now() с разрешением 0,1 мс и дополнительно верифицировалось средствами Android Profiler и Xcode Instruments, а сбор данных осуществлялся посредством специально разработанного промежуточного слоя (middleware).

Тестовая среда была стандартизирована: перед каждым циклом измерений устройства перезагружались и выводились из режима энергосбережения, чтобы исключить накопление системных артефактов и колебания тактовых частот. Все эксперименты проводились в одном сетевом сегменте с отключённым фоновым трафиком и деактивированными сторонними уведомлениями, что позволило минимизировать влияние внешних факторов на метрику отклика.

Таким образом, представленные решения являются существенным дополнением к традиционному анализу производительности, предлагая комплексный подход, учитывающий как технические, так и организационные аспекты управления состоянием в современных мобильных приложениях.

Основные результаты и обсуждение

Мобильные приложения в настоящее время требуют высокой интерактивности, обработки данных в реальном времени. Задачи управления состоянием выходят за пределы локальных переменных, для их решения необходимы структурированные подходы, обеспечивающие синхронизацию данных. React Native, основанный на принципах декларативного программирования, предоставляет инструменты для создания логики управления состоянием. Однако с ростом масштаба проектов, усложнением архитектуры, интеграцией с внешними сервисами появляется потребность в специализированных библиотеках, применении оптимизационных методов. С введением React Hooks управление состоянием компонентов стало проще и удобнее. Хуки useState, useReducer позволяют работать с состояниями разных типов. Первый применяется для простых состояний, второй — для более сложных случаев с множеством изменений [1], [2].

Мобильные приложения в настоящее время требуют высокой интерактивности, обработки данных в реальном времени. Задачи управления состоянием выходят за пределы локальных переменных, для их решения необходимы структурированные подходы, обеспечивающие синхронизацию данных [1], [2]. В свою очередь в рамках работы был осуществлен эксперимент, с целью проведения сравнения анализа существующих четырех методов управления состоянием в мобильных приложениях на React Native: Zustand, Redux, Context API и React Hooks (используя useState и useReducer). В рамках данной деятельности оценивались методы с точки зрения производительности, удобства разработки, масштабируемости, а также предсказуемости изменений. Тестирование проводилось на мобильном приложении «Список дел», в котором существовала возможность добавлять, удалять, а также фильтровать задачи. Каждое из этих действий было выполнено с учетом разного объема данных, в данном случае использовались: 100, а также 1000 задач, после выполнения которых измерялось время отклика интерфейса.

Начнем с рассмотрения процедуры добавления 100 задач. Где:

Zustand показал лучший результат — 18 мс, так как управление состоянием происходило без излишних обвязок. Также оно обновлялось локально в компонентах, благодаря чему удалось избежать необходимости в осуществлении сложных рендер-циклах.

Redux напротив потребовал больше времени — 40 мс. Что объяснялось тем фактором, что каждый вызов производил обновление состояния в централизованном хранилище, после чего запрашивал соответствующие редьюсеры.

Context API показал время отклика 35 мс, так как каждый компонент, использующий контекст, заново перерисовываться при изменении состояния.

React Hooks (useState) потребовал 20 мс, что находится на среднем уровне между Zustand и Redux. Что обусловлено тем фактом, что использование хуков позволило обновлять состояние локально.

Далее рассмотрим процесс удаления задач, которые были загружены в платформу. Для Zustand было продемонстрировано отличное время (22 мс) благодаря своей легковесной реализации. Redux показал медленный результат равный 50 мс. Что вызвано тем фактом, что элементы списка необходимо удалить посредством действия в хранилище, последующего прохождения через цепочку редьюсеров. Context API удаляет задачи за 45 мс. И React Hooks показал средний результат равный 30 мс, так как все задачи обрабатывались локально в компонентах.

Фильтрация по статусу (выполнено/не выполнено) требует выполнения определенной логики на каждом элементе списка. Zustand показал результат в 30 мс, так как работа с состоянием в нем происходит локально без необходимости перерисовывать весь компонент. Redux показал результаты равные 60 мс, что обусловлено тем фактором, что каждый

запрос на изменение состояния инициирует новую проверку редьюсеров, что в конечном счете приводит к большему времени обработки. Context API показал время 50 мс. React Hooks продемонстрировали среднее время отклика 40 мс, поскольку фильтрация происходит через локальное состояние компонентов.

В свою очередь, при загрузке большего числа элементов с пагинацией (1000 задач) результаты показали различие. Zustand справился с пагинацией за 150 мс, поскольку управление состоянием происходит в легковесной, а также локальной форме. Redux потребовал 200 мс из-за необходимости перерасчета состояния, последующего обновления всей информации через экшены. Context API продемонстрировал результат в 180 мс, что медленнее, чем у Zustand, из-за перерисовок компонентов, использующих контекст. React Hooks выдал результат в 170 мс, что является усредненным по отношению к другим методам. Далее рассмотрим изменение в потреблении памяти, которое измерялось при помощи инструментов профилирования в Android Profiler и Xcode Instruments. Далее в таблице 1 будут отражены результаты измерения памяти.

Таблица 1 - Результаты измерения памяти

DOI: https://doi.org/10.60797/itech.2025.8.2.1

Метод	Потребление памяти, МБ
Zustand	5,3
Redux	7,1
Context API	6,8
React Hooks	6,0

Как видно из данных таблицы 1, метод Zustand использует оптимальное количество памяти равное 5.3 МБ. Что объясняется тем, фактом, что метод работает без необходимости в дополнительном контексте или хранилище. Redux требует памяти 7.1 МБ. Это связано с тем, что использует централизованное хранилище, сериализует данные состояния. Context API потребляет 6.8 МБ памяти, из-за того, что компонент, использующий контекст, перерисовывается при изменении состояния, что в свою очередь приводит к большему потреблению ресурсов по сравнению с Zustand. React Hooks использует 6.0 МБ памяти. Что объясняется тем, что состояние в компонентах требует управления памятью для каждого хука. Ниже в таблице 2 будет рассмотрено удобство разработки мобильного приложения используя методы React Native.

Таблица 2 - Удобство разработки мобильного приложения используя методы React Native

DOI: https://doi.org/10.60797/itech.2025.8.2.2

Метод	Время разработки, часы	Количество строк кода	Сложность реализации (1-5)
Zustand	4	45	2
Redux	8	120	4
Context API	6	80	3
React Hooks	5	50	2

Как видно из данных таблицы 2, метод Zustand оказался удобным, а также быстрым в силу того, что в нем использовалось 45 строк кода, а также минимальной сложностью реализации (оценка — 2 из 5). Код был интуитивно понятен, также в нем не требовалось использовать дополнительные библиотеки или настройки. Redux потребовал больше времени для реализации (8 часов), а также большого количества строк кода (120). В силу того, что для работы с асинхронными действиями потребовалось подключить middleware, из-за чего была увеличена сложность разработки (оценка — 4 из 5). Context API потребовал 6 часов на реализацию, а также 80 строк кода. Реализация была относительно простой, но в крупных приложениях сложность возрастала (оценка — 3 из 5). React Hooks потребовали 5 часов на разработку, а также 50 строк кода, что делает их подходящими для прототипирования, использования в небольших приложениях (оценка — 2 из 5).

Однако, когда речь шла о масштабируемости, а также добавлении новых функциональностей, Redux оказался гибким, а также подходящим для крупных проектов. С централизованным хранилищем, а также четкой архитектурой приложения его легко адаптировать под новые требования. Однако для работы с легкими проектами использование методов Zustand и React Hooks является достаточным, хотя в случае масштабирования приложение становилось сложным в поддержке, так как необходимо было ручное управление состоянием между несколькими компонентами. Context API, хотя и удобен для небольших приложений, с ростом сложности начинает страдать от потерь производительности из-за необходимости обновлять все дерево компонентов при каждом изменении контекста.

Для тестирования предсказуемости изменений были проведены тесты на асинхронные обновления состояния, такие как добавление новых задач через API. В этом контексте Redux продемонстрировал наилучшие результаты, так как его структура (экшены, редьюсеры) гарантирует предсказуемость, а также стабильность состояния. Zustand также хорошо справился с асинхронными операциями, но в некоторых случаях происходило расхождение состояния из-за менее формализованного подхода. Context API и React Hooks в некоторых случаях требовали дополнительных усилий

для синхронизации состояний в компонентах, что в свою очередь приводило к проблемам в сложных приложениях. Ниже будут приведены примеры кода для каждого из методов управления состоянием приложения.

Zustand:

```
import create from 'zustand';

const useStore = create(set => ({
 tasks: [],
 addTask: (task) => set(state => ({ tasks: [...state.tasks, task] })),
 removeTask: (id) => set(state => ({ tasks: state.tasks.filter(task => task.id !== id) })),
}));

const TaskList = () => {
 const { tasks, addTask, removeTask } = useStore();

 return (
<View>
        <Button onPress={() => addTask({ id: Math.random(), text: 'New Task' })} title="Add Task" />
    {tasks.map(task => (
        <View key={task.id}>
        <Text>{task.text}</Text>
        <Button onPress={() => removeTask(task.id)} title="Remove" />
        </View>
        ))}
</View>
 );
};
```

Redux:

```
import { createStore } from 'redux';

// Actions
const ADD_TASK = 'ADD_TASK';
const REMOVE_TASK = 'REMOVE_TASK';

// Reducer
const tasksReducer = (state = [], action) => {
 switch(action.type) {
case ADD_TASK:
        return [...state, action.payload];
case REMOVE_TASK:
        return state.filter(task => task.id !== action.payload);
default:
        return state;
 }
};

// Store
const store = createStore(tasksReducer);

// Actions
const addTask = (task) => ({ type: ADD_TASK, payload: task });
const removeTask = (id) => ({ type: REMOVE_TASK, payload: id });

const TaskList = () => {
 const tasks = useSelector(state => state);
 const dispatch = useDispatch();
 return (
<View>
        <Button onPress={() => dispatch(addTask({ id: Math.random(), text: 'New Task' }))} title="Add
Task" />
    {tasks.map(task => (
        <View key={task.id}>
        <Text>{task.text}</Text>
        <Button onPress={() => dispatch(removeTask(task.id))} title="Remove" />
        </View>
    ))}
  </View>
 );
};
```

Context API:

```
import React, { createContext, useContext, useState } from 'react';

const TaskContext = createContext();

const TaskProvider = ({ children }) => {
 const [tasks, setTasks] = useState([]);

 const addTask = (task) => setTasks([...tasks, task]);
 const removeTask = (id) => setTasks(tasks.filter(task => task.id !== id));

 return (
<TaskContext.Provider value={{ tasks, addTask, removeTask }}>
        {children}
</TaskContext.Provider>
 );
};

const TaskList = () => {
 const { tasks, addTask, removeTask } = useContext(TaskContext);

 return (
<View>
        <Button onPress={() => addTask({ id: Math.random(), text: 'New Task' })} title="Add Task" />
    {tasks.map(task => (
        <View key={task.id}>
        <Text>{task.text}</Text>
        <Button onPress={() => removeTask(task.id)} title="Remove" />
        </View>
        ))}
</View>
 );
};
```

React Hooks:

```
import React, { useState } from 'react';

const TaskList = () => {
 const [tasks, setTasks] = useState([]);

 const addTask = (task) => setTasks([...tasks, task]);
 const removeTask = (id) => setTasks(tasks.filter(task => task.id !== id));

 return (
<View>
        <Button onPress={() => addTask({ id: Math.random(), text: 'New Task' })} title="Add Task" />
    {tasks.map(task => (
        <View key={task.id}>
        <Text>{task.text}</Text>
        <Button onPress={() => removeTask(task.id)} title="Remove" />
        </View>
        ))}
   </View>
 );
};
```

Далее в таблице 3 будут описаны преимущества и недостатки управления состоянием в мобильных приложениях на React Native.

Таблица 3 - Преимущества и недостатки управления состоянием в мобильных приложениях на React Native

DOI: https://doi.org/10.60797/itech.2025.8.2.3

Преимущества	Недостатки
React Native позволяет использовать один код для обеих платформ, что упрощает разработку, а также управление состоянием.	При использовании сложных механизмов управления состоянием (например, Redux) возможны проблемы с производительностью, что актуально для больших приложений.

Преимущества	Недостатки
React Native поддерживает различные библиотеки, а также подходы для управления состоянием (например, Context API, Redux, Recoil, Zustand).	Для сложных сценариев управления состоянием необходимо использовать сторонние библиотеки, что, в свою очередь, способно увеличить сложность, размер приложения.
Это упрощает отладку, делает код предсказуемым, так как данные проходят через компоненты только в одном направлении.	В некоторых случаях управление состоянием способно привести к большому количеству «шаблонного» кода (например, с Redux), что увеличивает объем работы по поддержке приложения.
Многие библиотеки, такие как MobX или Recoil, предоставляют декларативные, реактивные способы работы с состоянием, что делает код чище, а также легче для понимания.	В больших приложениях, где управление состоянием становится сложным, возникают проблемы с масштабируемостью, поддержанием чистоты архитектуры.
React Native легче интегрируется с библиотеками, а также инструментами для управления состоянием, такими как Redux DevTools, что облегчает отладку, тестирование.	Некоторые подходы к управлению состоянием (например, использование Context API для часто меняющихся данных) способно приводить к лишним рендерингам компонентов, что снижает производительность.

Таким образом, Zustand оказался быстрым, а также легким в реализации методом для небольших приложений, требующих высокой производительности, минимальных затрат. Redux подходит для крупных, а также сложных приложений, где необходима высокая предсказуемость, централизованное управление состоянием. React Hooks также представляет собой удобный метод для небольших проектов, но с последующем увеличением масштаба потребуется вложение усилий для управления состоянием. Context API хорошо работает в небольших приложениях, но теряет производительность при увеличении объема данных, а также сложности приложения. В свою очередь выбор метода управления состоянием зависит от сложности приложения. Для небольших проектов подойдут решения на основе useState и `Context API. Для более крупных решений предпочтительнее использовать Redux или MobX. Важно учитывать требования к асинхронности, производительности, удобству отладки при выборе подхода.

Заключение

Резюмируя, выбор подхода определяется множеством факторов, среди которых размер проекта, его сложность, требования к производительности, удобству разработки. Анализ таких решений, как контекст API, Redux, MobX, Recoil, Zustand, позволил выделить особенности каждого инструмента, определить области их применения.

Для небольших проектов с низкими требованиями к сложности, быстрой реализации решения на основе контекста API или Zustand является оптимальным. Эти инструменты предоставляют простоту в использовании, необходимую производительность для приложений с ограниченными ресурсами. Для крупных приложений с необходимостью сложной логики взаимодействия, управления состоянием предпочтительнее использовать такие инструменты, как Redux, MobX. Эти решения обеспечивают поддержку масштабируемости, позволяют контролировать состояние в больших системах.

Результаты работы показывают, что неправильный выбор инструмента для управления состоянием сказывается на производительности приложения, усложняет поддержку, ухудшает взаимодействие с пользователем. Поэтому важно провести тщательную оценку требований проекта, принять решение, которое наиболее соответствует его специфике.

Кроме того, новые инструменты, как Recoil, открывают возможности для гибкого управления состоянием, что делает их перспективными для применения в будущем. Эти решения упрощают работу с состоянием, что важно для динамично развивающихся приложений.

Конфликт интересов

Не указан.

Рецензия

Все статьи проходят рецензирование. Но рецензент или автор статьи предпочли не публиковать рецензию к этой статье в открытом доступе. Рецензия может быть предоставлена компетентным органам по запросу.

Conflict of Interest

None declared.

Review

All articles are peer-reviewed. But the reviewer or the author of the article chose not to publish a review of this article in the public domain. The review can be provided to the competent authorities upon request.

Список литературы на английском языке / References in English

1. Pronina D. Comparison of redux and react hooks methods in terms of performance / D. Pronina, I. Kyrychenko // 6th International Conference on Computational Linguistics and Intelligent Systems. — 2022. — Vol. 1613. — P. 73.

2. Olexandr B. Optimization of cross-platform applications using the React library / B. Olexandr, K. Olexandr // World science. — 2024. — № 3 (85). — P. 1–6.

3. Qianqian L. A Comprehensive Study on State Management Patterns of React / L. Qianqian, D. Yuxiao // 2023 International Conference on Applied Physics and Computing. — 2023. — P. 769–772.

4. Donvir A. Application State Management (ASM) in the Modern Web and Mobile Applications: A Comprehensive Review / A. Donvir, A. Jain, P.K. Saraswathi // arXiv. — 2407.19318. — 2024.

5. Cabañero R.A.C. Mobile Incident Management System using React Native / R.A.C. Cabañero // International Journal of Advanced Research in Science Communication and Technology. — 2023. — P. 891–897.

6. Toka L. Predicting cloud-native application failures based on monitoring data of cloud infrastructure / L. Toka [et al.] // 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM). — 2021. — P. 842–847.

7. Tagdiwala V. Robust Client and Server State Synchronisation Framework For React Applications: React-state-sync / V. Tagdiwala [et al.] // 2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE). — 2023. — P. 475–481.

8. Oleshchenko L. Software methods analysis of optimizing the local state of data web applications management / L. Oleshchenko // KPI science news. — 2022. — № 1 (2). — P. 260.

9. Xu Y. A Distributed State Management Approach for FaaS Platforms / Y. Xu [et al.] // 2024 9th International Conference on Computer and Communication Systems (ICCCS). — 2024. — P. 1306–1312.

10. Rahman A. Strategi pengembangan / A. Rahman, A.S. Rahman, M. Hakim // Aset SDM. — 2024. — № 7 (2). — P. 44–49.

11. Levkowitz S. Theory and Method for Reactive Semantics in Application Development / S. Levkowitz // 2024 IEEE International Systems Conference (SysCon). — 2024. — P. 1–3.

12. Saputro D.F. Mobile Point of Sales (Mi-POS) Application for Cashiers Using React Native Framework A Case Study at Fajar Jaya Snack Shop / D.F. Saputro, D. Gunawan // Jurnal Ecotipe (Electronic, Control, Telecommunication, Information, and Power Engineering). — 2023. — № 10 (1). — P. 121–130.