

Архитектурные взаимодействия компонентов React VDOM, обеспечивающие высокую производительность приложений

Вихляев Дмитрий Романович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данном исследовании описывается взаимодействие ключевых компонентов реализации VDOM в инструменте React.js. Приводится функциональное описание способы ускорения рендеринга веб-страниц frontend-приложений. Результатом работы станет архитектурный разбор внутренних процессов VDOM, позволяющих оптимизировать тяжеловесные процедуры на стороне клиента.

Ключевые слова: VDOM, React.js, frontend, рендеринг.

Architectural interactions of React VDOM components that ensure high application performance

Vikhlyayev Dmitry Romanovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This study describes the interaction of the key components of the DOM implementation in the tool. Reac.js . It provides a functional description of ways to speed up rendering of web pages of frontend applications. The result of the work will be an architectural analysis of the internal processes of VDOM, allowing to optimize heavy procedures on the client side.

Keywords: VDOM, Reac.js, frontend, rendering.

1 Введение

1.1 Актуальность

Виртуальный дом (Virtual DOM) и библиотека React широко применяемые инструменты в современной веб-разработке. Virtual DOM представляет собой абстрактное представление реального дерева DOM, которое позволяет эффективно обновлять интерфейс пользователя за счёт минимизации количества прямых манипуляций с DOM-структурой. Это достигается путём создания копии DOM в виде JavaScript-объекта, последующего сравнения его с предыдущим состоянием и внесения лишь необходимых изменений в реальный DOM. Такой подход позволяет значительно повысить производительность пользовательского интерфейса, особенно в условиях частого обновления данных. React, как библиотека для

построения компонентных пользовательских интерфейсов, предоставляет разработчику инструменты для создания модульных, многократно используемых и хорошо масштабируемых компонентов. Одной из ключевых особенностей React является декларативный стиль программирования, который способствует повышению читаемости и сопровождению кода. Применение Virtual DOM в архитектуре React позволяет эффективно реализовывать концепцию однонаправленного потока данных и обновлять только те части интерфейса, которые действительно претерпели изменения, что снижает нагрузку на браузер и улучшает отклик приложения.

1.2 Обзор исследований

В своей статье П.А. Полянская использовала библиотеку React при создании веб-приложения [1]. В.В.Аргасова исследовала web-фреймворк React в разработке информационных систем [2]. Д.А.Мартинovich, С.А.Сатторов описали методы улучшения разработки веб компонентов нативных сайтов с помощью React js [3]. Э.А.Бекирова провела сравнительный анализ производительности и архитектурных подходов в javascript фреймворках: Vue.js и React [4]. Е.В.Ломонов, В.А.Кузьменко исследовали особенности разработки и инструментарий фреймворка React [5]. О.С.Благодельский изучил влияние передовых технологий фронтенд-разработки на улучшение пользовательского интерфейса и повышение уровня взаимодействия с веб-приложениями [6].

1.3 Цель исследования

Цель исследования – описать компоненты и функции, ускоряющие процесс рендеринга веб-страницы на стороне клиента в архитектуре VDOM React.js.

2 Материалы и методы

В работе приводится архитектурное описание исследуемых компонентов. В графическом виде представлены процедурные последовательности. Диаграммы построены с помощью инструмента plantuml.

3 Результаты и обсуждения

Виртуальный DOM является копией реального DOM. Он хранится в оперативной памяти за счёт чего быстрее операции, изменяющие его структура, происходят быстрее. Процесс синхронизации между реальным и виртуальным DOM происходит с помощью механизма согласования. Основная цель использования виртуального DOM заключается в оптимизации процессов обновления пользовательского интерфейса путём минимизации прямых обращений к реальному DOM, которые являются ресурсоёмкими и медленными. Процесс работы с Virtual DOM в React можно условно разделить на два ключевых этапа: рендеринг (rendering) и согласование (reconciliation).

На этапе рендеринга создаётся новое дерево виртуального DOM. Это происходит всякий раз, когда компонент React вызывает метод render() или

используется функциональный компонент, возвращающий JSX-разметку. В результате выполнения данного метода генерируется структура, описывающая текущий внешний вид интерфейса в виде вложенных JavaScript-объектов. Эта структура не взаимодействует напрямую с реальным DOM, а служит его абстрактной моделью. Каждое изменение состояния компонента (state) или его входных параметров (props) инициирует повторный вызов функции рендеринга, что приводит к созданию нового дерева Virtual DOM.

Согласование – это процесс сравнения предыдущего и нового дерева Virtual DOM с целью определения минимального набора изменений, необходимых для приведения реального DOM в актуальное состояние. Этот процесс реализуется с помощью алгоритма сравнения, называемого диффингом (diffing). Во время согласования React последовательно сопоставляет элементы обоих деревьев, определяя:

- какие элементы остались без изменений и могут быть повторно использованы;
- какие элементы были добавлены и нуждаются в создании новых узлов DOM;
- какие элементы были удалены и подлежат удалению из DOM;
- какие элементы были изменены и требуют обновления соответствующих DOM-узлов.

После завершения сравнения React формирует список изменений (patches) и применяет его к реальному DOM с использованием оптимизированных операций обновления (рис.1).

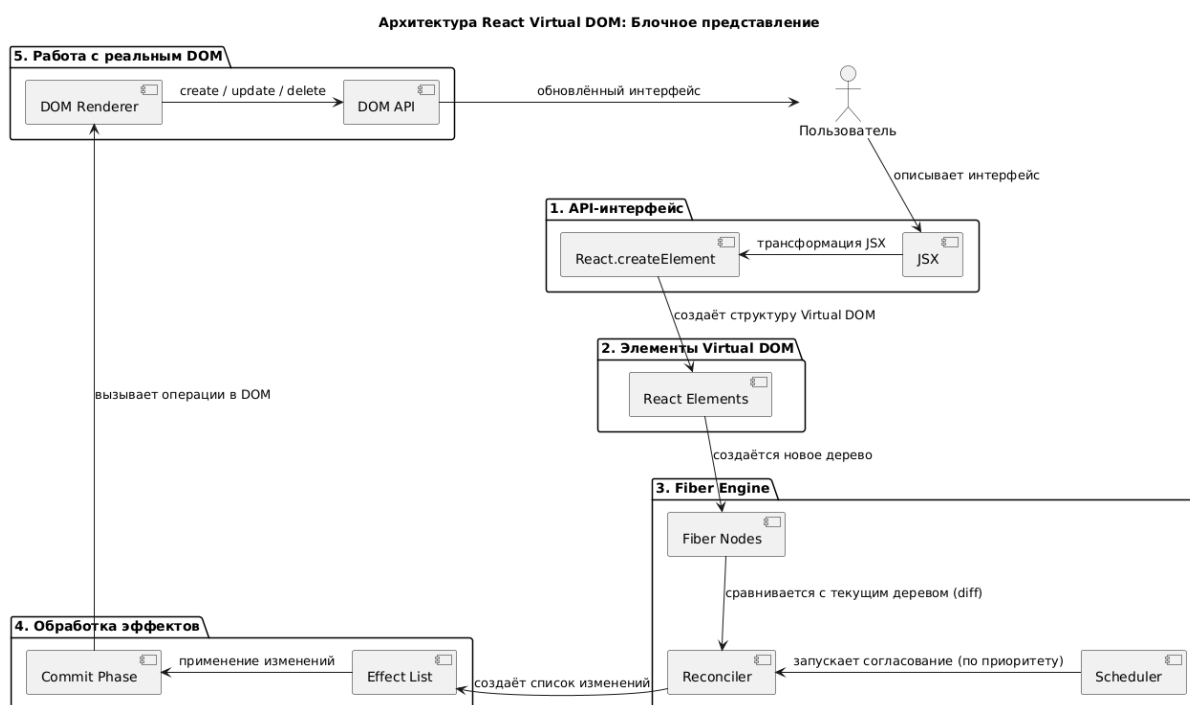


Рис. 1. Архитектура VDOM React.js

Виртуальный дом (Virtual DOM) реализует эффективный механизм обновления пользовательского интерфейса посредством применения

последовательно организованных функциональных методов, обеспечивающих высокую производительность и минимальные затраты на манипуляции с реальным деревом DOM. В основе этого подхода лежит концепция построения абстрактного представления интерфейсной структуры в виде вложенных JavaScript-объектов, которые отражают текущее состояние компонента. На первом этапе осуществляется создание элементов виртуального дерева, реализуемое посредством функции, аналогичной `createElement`, которая формирует объект, содержащий тип элемента, его свойства и вложенные компоненты. Эти объекты служат строительными блоками виртуального представления DOM, не взаимодействующего напрямую с браузерным API, что позволяет исключить дорогостоящие операции непосредственного обновления DOM.

На следующем этапе выполняется процесс рендеринга, в ходе которого происходит генерация нового виртуального дерева в результате вызова функции рендеринга компонента. Это может быть метод `render` в классовых компонентах либо возврат JSX в функциональных компонентах. Построенное дерево представляет актуальное состояние пользовательского интерфейса, определяемое текущими значениями состояния и входных параметров. В случае изменения этих данных инициируется повторный рендеринг компонента и формируется новая структура виртуального дерева.

После построения нового дерева запускается процедура сравнения или диффинга, направленная на выявление различий между текущим и предыдущим состоянием виртуального DOM. Сравнение осуществляется с использованием оптимизированного алгоритма сопоставления, в рамках которого анализируются типы элементов, их ключи и иерархическая структура. На основании выявленных различий формируется набор инструкций, описывающих минимальные необходимые изменения, необходимые для синхронизации виртуального представления с реальным DOM.

Формирование этих инструкций, или патчей, осуществляется на этапе согласования, и представляет собой логическую запись операций вставки, удаления, перемещения или модификации узлов. Эти операции не применяются немедленно, а группируются и оптимизируются с целью минимизации количества обращений к реальному DOM. Завершающий этап – это коммит-фаза, в ходе которой React либо другой механизм Virtual DOM применяет подготовленные изменения к реальному DOM, используя базовые методы манипуляции деревом, такие как `appendChild`, `removeChild`, `setAttribute` (рис.2).

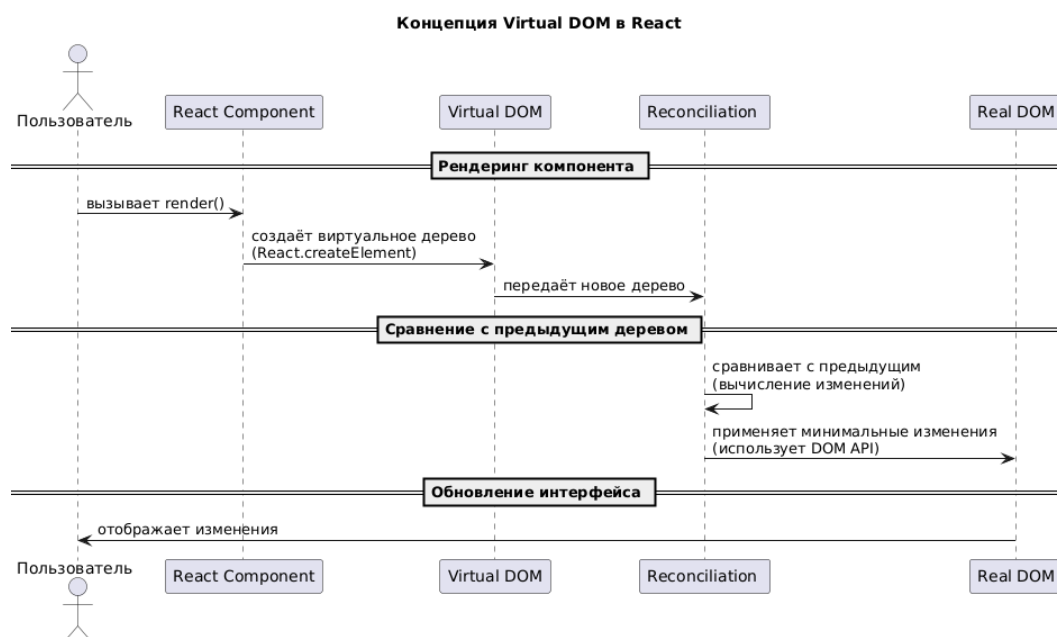


Рис. 2. Диаграмма последовательности исполнения рендеринга в React.js

Fiber – это новая архитектура механизма согласования (reconciliation) в React, введённая с версии 16. Она была создана для решения проблем старого одношагового подхода к обновлению Virtual DOM и обеспечила более гибкое, прерываемое и приоритетное обновление интерфейса. Ниже представлено последовательное и подробное описание работы Fiber в научно-ориентированном стиле.

Архитектура Fiber представляет собой фундаментальную перестройку внутреннего механизма согласования в библиотеке React, направленную на решение задач, связанных с производительностью, управлением приоритетами и возможностью прерывать вычисления. В отличие от предыдущей модели, основанной на рекурсивном обходе дерева компонентов и синхронном обновлении виртуального представления пользовательского интерфейса, архитектура Fiber реализует пошаговый подход к построению и согласованию дерева компонентов, в основе которого лежит представление каждого узла в виде специализированного объекта Fiber с чётко определённой структурой и связями.

Каждый узел Fiber содержит тип компонента, свойства, состояние, указатели на дочерние, родительские и соседние элементы, а также ссылку на предыдущее состояние, что обеспечивает возможность двойной буферизации и сравнения различных версий интерфейса. Благодаря этим характеристикам React получает возможность пошагового обхода дерева, с сохранением и возобновлением промежуточных результатов вычислений, что критически важно для обеспечения отзывчивости интерфейса при выполнении тяжёлых или долгосрочных операций.

Работа Fiber-архитектуры условно разделяется на две фазы: фазу рендеринга и фазу коммита. В ходе первой осуществляется построение нового дерева компонентов, представляющего собой абстрактную модель

будущего интерфейса. Этот процесс является прерываемым и реализуется как цикл единичных операций (unit of work), каждая из которых обрабатывает отдельный узел Fiber и подготавливает его к последующему применению. Использование прерываемой модели обеспечивает возможность отклика на высокоприоритетные события, такие как пользовательское взаимодействие, даже при активной переработке дерева компонентов.

Фаза коммита, в отличие от рендеринга, является непрерываемой и отвечает за непосредственное применение всех накопленных изменений к реальному дереву DOM. В рамках данной фазы выполняются операции вставки, удаления и модификации DOM-элементов, а также вызовы побочных эффектов, ассоциированных с жизненным циклом компонентов. Отделение фазы вычислений от фазы побочных действий позволяет строго контролировать момент воздействия на DOM и исключить нежелательные эффекты от незавершённых операций согласования.

Архитектура Fiber также закладывает основы для реализации конкурентного рендеринга, в рамках которого React может выполнять несколько независимых операций обновления параллельно, управлять их приоритетами и реагировать на появление более актуальных задач. Модель обеспечивает высокую масштабируемость интерфейса и гарантирует минимальное время отклика вне зависимости от сложности структуры пользовательского интерфейса (рис.3).

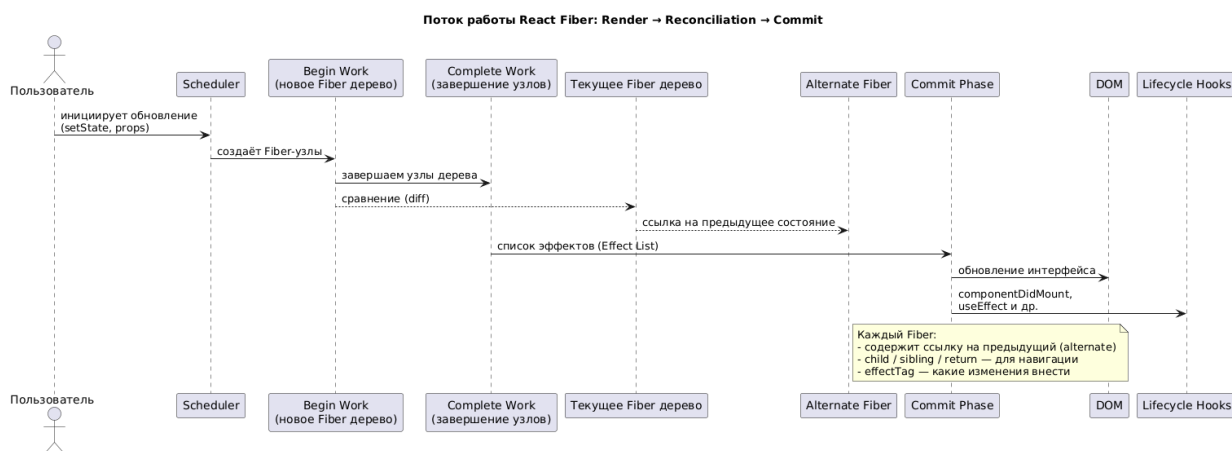


Рис. 3. Диаграмма последовательности потока работы Fiber

Алгоритм diffing (или алгоритм различий) в контексте библиотеки React представляет собой метод оптимизированного сравнения двух версий виртуального DOM-дерева с целью определения минимального набора изменений, которые необходимо внести в реальное дерево DOM. Проблема, решаемая данным алгоритмом, заключается в высокой вычислительной сложности полного сравнения произвольных деревьев, которая теоретически составляет $O(n^3)$, где n — количество узлов. React применяет ряд эвристических упрощений, благодаря которым сложность задачи снижается до линейной $O(n)$, обеспечивая высокую производительность при обновлении пользовательского интерфейса.

Основная идея алгоритма заключается в том, что при повторном рендеринге компонента создаётся новая версия виртуального DOM-дерева. Алгоритм сравнивает его с предыдущей версией и, используя набор упрощённых правил, определяет, какие части дерева остались неизменными, какие были обновлены, добавлены или удалены. В результате этого сравнения формируется набор операций, который затем применяется к реальному DOM минимально необходимым образом.

Ключевые эвристики, лежащие в основе алгоритма diffing, включают определённые положения.

Различие по типу узлов: если тип двух сравниваемых узлов отличается (например, `<div>` и `<span>`), считается, что соответствующий DOM-элемент должен быть полностью удалён и создан заново. Это же правило применяется, если типы компонентов отличаются, включая классовые и функциональные компоненты.

Сравнение свойств: если тип узла одинаков, производится побайтовое сравнение его свойств. Только те свойства, которые изменились, подлежат обновлению в реальном DOM.

Рекурсивное сравнение дочерних элементов: если узлы имеют одинаковый тип, выполняется сравнение их дочерних элементов. При этом React предполагает, что порядок дочерних узлов остаётся стабильным между рейдерами.

Идентификация по ключу (key): при наличии списка однотипных элементов (например, `<li>` в `<ul>`) для повышения точности сравнения используется специальный атрибут `key`. Он позволяет избежать ненужного удаления и повторного создания элементов, а также минимизировать количество перемещений. В случае отсутствия ключей React использует индекс элемента, что может привести к неэффективному поведению при изменении порядка элементов.

В процессе работы алгоритма сравниваются не только элементы, но и вложенные компоненты, для которых может быть повторно вызван метод `render`, если их свойства или состояние были изменены. В противном случае повторный рендер компонента может быть пропущен, если включена оптимизация типа `PureComponent` или используется метод `shouldComponentUpdate`.

Алгоритм diffing реализуется внутри архитектуры Fiber и работает во время фазы рендеринга. Он позволяет React эффективно определять, какие именно участки интерфейса нуждаются в обновлении, тем самым достигая высокой производительности и отзывчивости при изменении состояния приложения. Таким образом, диффинг служит основой виртуальной модели обновления пользовательского интерфейса, позволяя React минимизировать прямое взаимодействие с DOM и обеспечивать плавную работу даже в условиях сложных и динамически изменяющихся структур (рис.4,5).

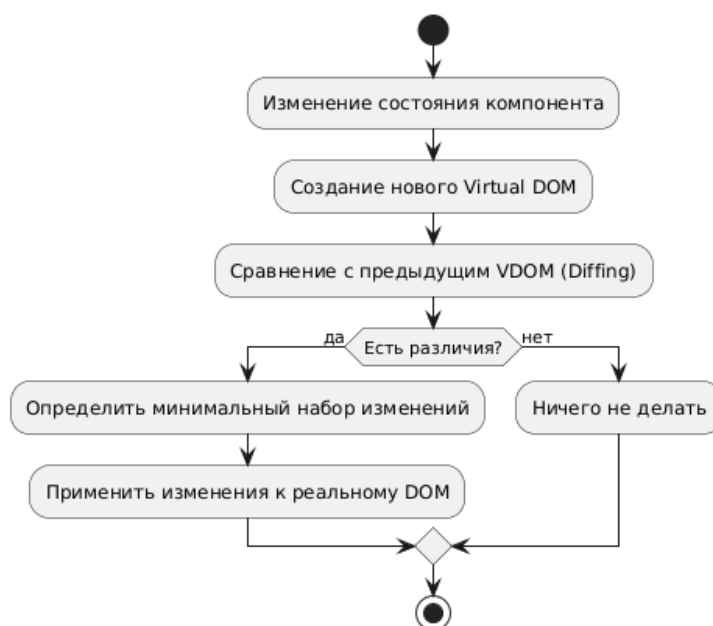


Рис. 4. Последовательность действий при изменении состоянии компонента

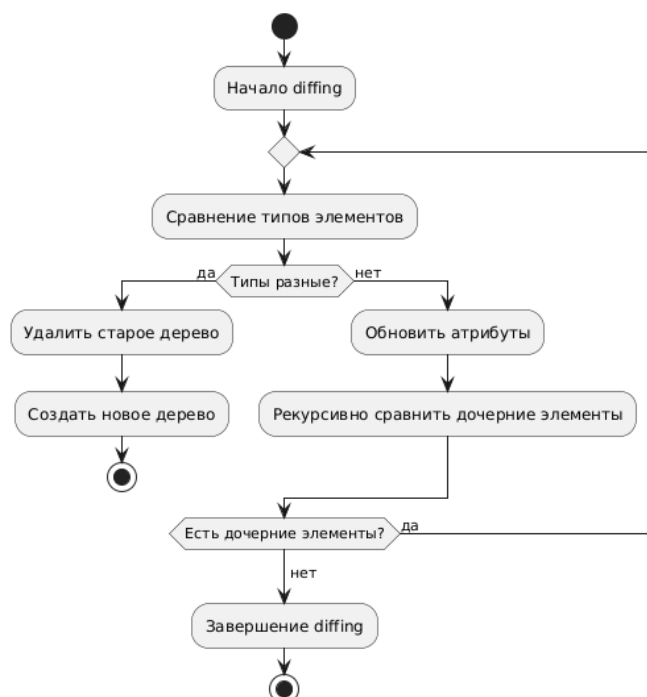


Рис. 5. Последовательность действий в алгоритме различий

Архитектура виртуального DOM, будучи абстрактной моделью интерфейса, обеспечивает значительные преимущества в плане производительности, но не лишена определённых ограничений. Её применение основано на концепции промежуточного дерева, в котором хранятся текущие и предыдущие состояния интерфейса, позволяя вычислять минимальный набор изменений перед внесением их в реальный DOM. Это, в свою очередь, снижает количество прямых операций с DOM – одного из самых ресурсоёмких узлов в браузере, – и минимизирует время, затрачиваемое на перерисовку.

С точки зрения фактической производительности, исследование, проведённое Google и независимыми разработчиками в рамках проекта js-framework-benchmark [9], показывает, что VDOM-движки, такие как React Fiber, Inferno и Preact, обеспечивают обновление интерфейса в среднем за 1-3мс. при простых изменениях, и до 10-20мс. при работе со сложными вложенными деревьями. Это значительно быстрее, чем традиционный подход, при котором перерисовывается весь интерфейс, независимо от характера изменений.

Тем не менее, архитектура VDOM имеет узкие места. Дополнительные вычислительные издержки, связанные с построением и сравнением виртуального дерева. Каждое обновление требует социализации состояния интерфейса, выполнения алгоритма diffing и генерации исправления. В случае большого количества компонентов и частых рендеров это может приводить к увеличению нагрузки на основной поток JavaScript, особенно в старых устройствах и мобильных браузерах.

VDOM требует хранения предыдущего состояния дерева, что ведёт к увеличению потребления памяти. В React каждый узел Fiber содержит ссылки на текущее и старое состояния, родительские и дочерние элементы, что, в случае больших приложений, может занимать десятки мегабайт оперативной памяти. Это особенно критично для устройств с ограниченными ресурсами и может влиять на время запуска (Time-To-Interactive) и стабильность интерфейса.

Для решения этих проблем были предложены методы оптимизации:

- shouldComponentUpdate / React.memo / PureComponent – предотвращение лишних рендеров;
- key-пропы – для точной идентификации элементов в списках;
- lazy-loading компонентов и фрагментов дерева – уменьшение объёма работы на каждой итерации;
- использование хуков с мемоизацией (useMemo, useCallback) – для избежания повторных вычислений;
- разделение ответственности (component splitting) – декомпозиция дерева на логически изолированные части;
- использование Concurrent Mode / startTransition – приоритезация и прерываемость рендеров.

С точки зрения плюсов и минусов для клиента, VDOM:

- Преимущества: высокая отзывчивость интерфейса; снижение количества операций с DOM; предсказуемость и удобство отладки; кросс-браузерная совместимость; унификация UI-логики.

- Недостатки: рост потребления оперативной памяти; увеличение загрузки CPU; сложность контроля над производительностью при глубоком дереве; возможные утечки памяти при ошибках в управлении состоянием.

Для сервера, особенно в контексте серверного рендеринга (SSR), VDOM также упирается в два противоположных направления:

– Преимущества: возможность рендеринга в виде сериализованного дерева, независимого от DOM API браузера; предсказуемость шаблонов рендеринга; удобство организации изоморфных приложений.

– Недостатки: высокие накладные расходы на сериализацию дерева и выполнение diff-алгоритма на сервере; необходимость эмуляции среды браузера (например, через jsdom); задержки на этапе начального рендера (TTFB) при больших деревьях.

4 Выводы

Таким образом, виртуальный DOM представляет собой компромисс между удобством декларативной разработки и низкоуровневой производительностью. Он оправдывает себя в условиях динамических интерфейсов с высокой частотой изменений и сложной структурой состояния, но требует осознанного подхода к оптимизации, особенно при разработке тяжёлых клиентских приложений и систем с ограниченными вычислительными ресурсами.

В результате исследования была рассмотрена архитектурная составляющая функционала VDOM в React.js. Рассмотрены ключевые компоненты обеспечивающие процесс рендеринга. Приведены подходы реализующие оптимизацию и ускорение работы тяжеловесных процессов веб-приложения на стороне клиента. Описаны основные преимущества, и недостатки системного подхода и трудности при разработке.

Библиографический список

1. Полянская П.А. Использование библиотеки React при создании веб-приложений // Студенческий. 2024. № 1-2 (255). С. 60-61.
2. Аргасова В.В. Применение web-фреймворка React при разработке информационных систем // Аллея науки. 2018. Т. 2. № 7 (23). С. 526-528.
3. Мартинович Д.А., Саторов С.А. Методы улучшения разработки веб-компонентов нативных сайтов с помощью React js // Славянский форум. 2024. № 2 (44). С. 338-344.
4. Бекирова Э.А. Сравнительный анализ производительности и архитектурных подходов в javascript фреймворках: Vue.js и React // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2024. № 3 (45). С. 18-23.
5. Ломонов Е.В., Кузьменко В.А. Особенности разработки и инструментарий фреймворка React // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2023. № 1 (39). С. 26-30.
6. Благодельский О.С. Влияние передовых технологий фронтенд-разработки на улучшение пользовательского интерфейса и повышение уровня взаимодействия с веб-приложениями // Актуальные исследования. 2024. № 20-1 (202). С. 73-79.
7. Чепурной М.П., Дяченко А.С. Сравнительный анализ фреймворков для

фронтенд-разработки // Международный студенческий научный вестник. 2025. № 1. С. 12.

8. Шадров И.А., Гаев Л.В. React в современной веб-разработке: ключевые задачи и решения // Инновационная наука. 2025. № 5-2. С. 88-89.
9. Официальные результаты бенчмарка js web frameworks URL: <https://krausest.github.io/js-framework-benchmark/index.html> (дата обращения: 24.06.2025).