Воронцов Глеб Игоревич, бакалавр
Уфимский университет науки и технологий
Vorontsov Gleb Igorevich
Ufa University of Science and Technology

СРАВНИТЕЛЬНЫЙ АНАЛИЗ КЛИЕНТСКОГО И СЕРВЕРНОГО РЕНДЕРИНГА COMPARATIVE ANALYSIS OF CLIENT-SIDE AND SERVER-SIDE RENDERING IN MODERN WEB APPLICATIONS

Аннотация. Рассмотрены клиентский, серверный, статический и гибридный подходы к формированию веб-интерфейсов. Сопоставлены их влияние на начальную загрузку, интерактивность, кэширование и нагрузку на инфраструктуру. Предложены критерии выбора архитектуры с учетом назначения приложения и условий эксплуатации.

Abstract. Client-side, server-side, static, and hybrid approaches to web interface rendering are reviewed. Their impact on initial loading, interactivity, caching, and infrastructure load is compared. Criteria for selecting an architecture according to application purpose and operating conditions are proposed.

Ключевые слова: Веб-приложение, клиентский рендеринг, серверный рендеринг, статическая генерация, гидратация, производительность.

Keywords: Web application, client-side rendering, server-side rendering, static generation, hydration, performance.

Введение

Современное веб-приложение объединяет представление данных, маршрутизацию, интерактивные элементы и обмен с серверными интерфейсами. Поэтому место, в котором формируется начальная разметка страницы, является не частной деталью реализации, а архитектурным решением. Оно влияет на объем передаваемого JavaScript-кода, время появления содержимого, требования к серверу, возможности кэширования и поведение приложения на устройствах различной производительности [1].

Цель статьи состоит в сравнении клиентского рендеринга, серверного рендеринга, статической генерации и гибридных схем. Анализ выполнен по документации современных платформ, стандартам HTTP и пользовательским метрикам производительности. Рассматриваются не отдельные фреймворки, а общие свойства подходов, позволяющие соотнести архитектуру с характером содержимого, частотой его обновления и требуемым уровнем интерактивности.

Основные модели формирования интерфейса

При клиентском рендеринге браузер получает базовый HTML-документ, загружает сценарии, запрашивает данные и формирует интерфейс посредством изменения DOM. Статические файлы приложения удобно размещать в распределенном хранилище, а переходы после запуска могут выполняться без полной перезагрузки страницы. Однако начальное отображение зависит от загрузки, разбора и выполнения JavaScript, вследствие чего медленная сеть или маломощный процессор увеличивают задержку до появления полезного содержимого [1].

Серверный рендеринг формирует HTML для каждого запроса и отправляет браузеру уже наполненную разметку. Такой вариант способен сократить путь до первого содержательного отображения и предоставляет содержимое до завершения выполнения клиентских сценариев. Одновременно возрастает роль времени ответа сервера: получение данных, построение дерева компонентов и сериализация результата входят в критический путь. Динамическая генерация также требует контроля масштабирования и стратегии кэширования [1, 4].

Статическая генерация переносит построение HTML на этап сборки. Полученные документы можно раздавать через сеть доставки содержимого без вычислений на каждый запрос. Модель особенно эффективна для документации, справочных разделов и материалов, изменяющихся реже, чем их запрашивают. Ограничением становится актуальность: изменение данных требует повторной генерации всей страницы либо ее части. Корректные правила HTTP-кэширования должны отделять повторно используемые ответы от персонализированных [5, 7].

Гибридная архитектура сочетает предварительно сформированный HTML с клиентской интерактивностью. В React процесс присоединения обработчиков к серверной разметке называется гидратацией; исходное дерево на клиенте должно соответствовать серверному результату [3]. Современные платформы позволяют оставлять получение данных и крупные зависимости на сервере, а интерактивные границы выполнять в браузере [2]. Преимущество такого подхода проявляется только при осознанном разделении компонентов, иначе пользователь получает и стоимость серверного построения, и избыточный клиентский пакет.

Критерии сравнительного анализа

Производительность нельзя описать одной длительностью загрузки. Core Web Vitals разделяют восприятие качества на скорость отображения крупнейшего элемента LCP, отзывчивость INP и визуальную стабильность CLS [6]. Дополнительные показатели TTFB и FCP помогают определить источник задержки. Серверный рендеринг может улучшить раннее отображение, но медленный TTFB ухудшит LCP; клиентская оптимизация может ускорить переходы, но длинные задачи JavaScript отрицательно влияют на INP.

Второй критерий – распределение вычислительной нагрузки. Клиентская схема переносит построение интерфейса на устройство пользователя и делает результат чувствительным к скорости процессора и объему памяти. Серверная схема обеспечивает более предсказуемую среду вычислений, но расходует ресурсы инфраструктуры на запросы. Гидратация не устраняет клиентскую работу полностью: браузер загружает код интерактивных компонентов, восстанавливает состояние и связывает обработчики, поэтому объем сценариев необходимо измерять отдельно от размера HTML.

Третий критерий связан с повторным использованием ответов. Стандарт HTTP определяет кэш как механизм сохранения семантики при сокращении повторных передач и задает директивы Cache-Control для срока свежести и областей хранения [7]. Статические документы обычно допускают длительное общее кэширование. Для серверных страниц решение зависит от персонализации и прав доступа, а ошибочная публикация пользовательского ответа в общем кэше создает риск раскрытия данных. При клиентском рендеринге отдельно настраиваются кэш оболочки и кэш API-ответов.

Выбор архитектуры и практическая проверка

Для открытого каталога, новостной страницы или документации важны раннее появление текста, индексируемая разметка и эффективная раздача через CDN. Здесь основой обычно служат статическая генерация либо серверный рендеринг. Для кабинета оператора, внутренней панели и графического редактора приоритетом являются длительная интерактивная сессия и частые локальные изменения состояния; после загрузки такой интерфейс может рационально работать преимущественно на клиенте. Тип приложения важнее формального выбора популярного фреймворка.

Практичная гибридная схема принимает решение на уровне маршрута и компонента. Публичная оболочка и данные, необходимые для первого экрана, формируются заранее или на сервере. В браузер передаются только те компоненты, которым требуются состояние, события либо доступ к Web API. Документация Next.js прямо связывает серверные компоненты с получением данных и сокращением клиентского JavaScript, а клиентские – с обработчиками, эффектами и браузерными возможностями [2]. Такое разделение уменьшает интерактивную границу без отказа от динамического поведения.

Сравнение вариантов следует проводить на функционально одинаковых прототипах. Необходимо фиксировать объем HTML, CSS и JavaScript, число запросов, TTFB, LCP, CLS, лабораторный TBT и серверное время формирования ответа. Измерения выполняются для холодного и прогретого кэша, нескольких профилей сети и устройств. Лабораторные результаты дополняются полевыми данными: для Core Web Vitals рекомендуется оценивать 75-й процентиль отдельно для мобильных и настольных загрузок [6].

Распространенная ошибка состоит в объявлении одного подхода безусловно быстрым. Предварительный HTML не гарантирует хорошую отзывчивость, если после него загружается крупный пакет сценариев; клиентская оболочка не гарантирует масштабируемость, если API формирует тяжелые ответы; статическая генерация теряет преимущества при неуправляемой инвалидации. Архитектурное решение должно фиксировать ожидаемый профиль пользователей, допустимую давность данных, модель кэширования и измеримые бюджеты производительности. После выпуска эти предположения проверяются мониторингом реальных загрузок.

Заключение

Клиентский рендеринг удобен для насыщенных интерактивных сессий и разгружает сервер от построения разметки, но повышает зависимость начального запуска от JavaScript и мощности устройства. Серверный рендеринг ускоряет доступ к исходному содержимому, однако требует контролировать TTFB, вычислительную стоимость и персонализированное кэширование. Статическая генерация обеспечивает минимальную стоимость раздачи для редко изменяемых материалов, а гибридная модель позволяет применять разные решения к отдельным маршрутам и компонентам.

Таким образом, универсального способа формирования веб-интерфейса не существует. Обоснованный выбор строится на свойствах содержимого, характере интерактивности, возможностях инфраструктуры и результатах измерений. Наиболее устойчивой является архитектура, в которой серверная и клиентская работа ограничены реальными обязанностями, кэширование соответствует семантике данных, а качество подтверждается совокупностью пользовательских и системных показателей.

Список литературы:

1. Osmani A., Miller J. Rendering on the Web // web.dev. 2019. Updated 05.01.2026. URL: https://web.dev/articles/rendering-on-the-web (дата обращения: 11.08.2026).

2. Next.js. Server and Client Components // Next.js Documentation. URL: https://nextjs.org/docs/app/getting-started/server-and-client-components (дата обращения: 11.08.2026).

3. React. hydrateRoot // React Documentation. URL: https://react.dev/reference/react-dom/client/hydrateRoot (дата обращения: 11.08.2026).

4. Next.js. Server-side Rendering (SSR) // Next.js Documentation. URL: https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering (дата обращения: 11.08.2026).

5. Next.js. Static Site Generation (SSG) // Next.js Documentation. URL: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation (дата обращения: 11.08.2026).

6. Google. Web Vitals // web.dev. URL: https://web.dev/articles/vitals (дата обращения: 11.08.2026).

7. Fielding R., Nottingham M., Reschke J. HTTP Caching. RFC 9111. IETF, 2022. DOI: 10.17487/RFC9111