Куревин Максим Александрович, Магистрант,
Ярославский государственный технический университет,
Ярославль

Научный руководитель:
Маевский Вячеслав Константинович,
доцент кафедры "Информационные системы и технологии", к.т.н.,
Ярославский государственный технический университет,
Ярославль

NEXT.JS И SSR: КАК УСКОРИТЬ ЗАГРУЗКУ СТРАНИЦ И УЛУЧШИТЬ SEO
NEXT.JS AND SSR: HOW TO SPEED UP PAGE LOADING AND IMPROVE SEO

Аннотация: Статья посвящена анализу влияния серверного рендеринга (SSR) в фреймворке Next.js на скорость загрузки и SEO. Рассматриваются подходы SSR, CSR и SSG, их преимущества и ограничения. Подробно описана архитектура SSR в Next.js и её влияние на Core Web Vitals. Приведено сравнение с другими SSR-фреймворками. Сформулированы практические рекомендации по применению SSR для повышения производительности и улучшения индексации сайта.

Abstract: The article analyzes the impact of server-side rendering (SSR) in the Next.js framework on page load speed and SEO. It reviews SSR, CSR, and SSG approaches, outlining their strengths and limitations. The architecture of SSR in Next.js and its effect on Core Web Vitals are described. A comparison with other SSR frameworks is provided. Practical recommendations are proposed for applying SSR to improve performance and search engine indexing.

Ключевые слова: Next.js; SSR; статическая генерация; CSR; скорость загрузки; Core Web Vitals; SEO.

Keywords: Next.js; SSR; static generation; CSR; loading speed; Core Web Vitals; SEO.

Введение

В современной веб-разработке особое внимание уделяется скорости загрузки страниц и их оптимизации для поисковых систем (SEO). Быстро загружающиеся сайты не только обеспечивают лучший пользовательский опыт, но и получают преимущество в ранжировании поисковыми системами [1]. Появление Core Web Vitals (ключевых показателей качества веб-страниц) в качестве факторов ранжирования подчёркивает актуальность производительности сайта для SEO. Эти метрики (например, Largest Contentful Paint, Cumulative Layout Shift и др.) отражают скорость отображения содержимого и стабильность страницы при загрузке и напрямую влияют на видимость сайта в поиске [1].

Одностраничные приложения (SPA) на базе JavaScript-фреймворков (React, Vue, Angular) традиционно используют клиентский рендеринг (CSR), когда весь HTML формируется в браузере. Однако CSR-архитектура может приводить к замедленной первой отрисовке контента и затруднениям при индексировании страниц поисковыми роботами [2]. Чтобы решить эти проблемы, разработчики все чаще обращаются к серверному рендерингу (SSR) и смежным техникам. SSR генерирует HTML-страницы на сервере перед отправкой их пользователю, что позволяет ускорить первоначальную загрузку и предоставить поисковым ботам готовый контент для индексирования [3].

Фреймворк Next.js, основанный на React, стал одним из наиболее популярных решений

для реализации SSR. Next.js предлагает гибридный подход к рендерингу: поддерживает как SSR на каждый запрос, так и статическую генерацию (SSG) во время сборки, а также сочетание методов (ISR – инкрементальная статическая регенерация) в зависимости от задач приложения. Такая универсальность делает Next.js привлекательным для разработчиков, стремящихся улучшить SEO-показатели и время загрузки своих сайтов. В данной работе актуально исследовать, каким образом SSR в Next.js помогает ускорить загрузку страниц и повысить эффективность SEO, а также сравнить Next.js с альтернативными инструментами SSR.

Цель статьи – проанализировать принципы серверного рендеринга в Next.js и его влияние на производительность веб-страниц и SEO, в контексте других методов рендеринга и фреймворков. Для достижения цели последовательно рассматриваются существующие подходы к рендерингу (CSR, SSG, SSR), детали реализации SSR в Next.js, влияние SSR на показатели скорости и SEO (включая Core Web Vitals), а также проводится сравнение Next.js с аналогичными фреймворками (Nuxt.js, Angular Universal, Remix). На основе анализа формулируются выводы о преимуществах и ограничениях SSR и рекомендации по его применению.

Обзор методов рендеринга веб-страниц

Современные веб-приложения могут отображать (рендерить) содержимое на разных этапах: на стороне клиента, на стороне сервера или заранее на этапе сборки. От выбранной стратегии рендеринга зависят скорость загрузки страницы, потребление ресурсов и степень готовности контента для поисковых систем. Рассмотрим три основных подхода: клиентский рендеринг (CSR), статическая генерация (SSG) и серверный рендеринг (SSR).

Клиентский рендеринг (CSR)

При клиентском рендеринге весь HTML формируется в браузере пользователя с помощью JavaScript. Сервер изначально возвращает минимальный HTML-каркас (например, пустой), а затем на клиент загружается JS-бандл приложения, который динамически генерирует DOM-элементы. Такой подход характерен для SPA-приложений. Преимущества CSR – высокая интерактивность и богатый пользовательский опыт после первоначальной загрузки: приложение может обновлять содержимое без перезагрузки страницы, реагируя на действия пользователя в реальном времени. Однако у CSR есть существенные недостатки. Пользователь видит пустую или неполную страницу, пока не загрузится и не выполнится JS, что увеличивает время до первого отображения контента. Метрики вроде First Contentful Paint (FCP) и Largest Contentful Paint (LCP) часто ухудшаются при чистом CSR [4]. Оптимизация ключевых показателей Core Web Vitals в условиях CSR может быть затруднена [5]. Кроме того, поисковым ботам приходится выполнять JS, чтобы получить содержимое страницы, что приводит к задержкам в индексировании. Хотя Google научился рендерить JavaScript, страницы с CSR могут индексироваться в два этапа (сначала базовый HTML, затем полное содержимое после рендеринга), что замедляет появление контента в поиске [1]. Таким образом, при CSR начальная загрузка обычно медленнее, а SEO может страдать из-за задержек в получении контента. На рисунке 1 показана схема процесса загрузки страницы при клиентском рендеринге, где HTML и контент формируются только после загрузки JavaScript.

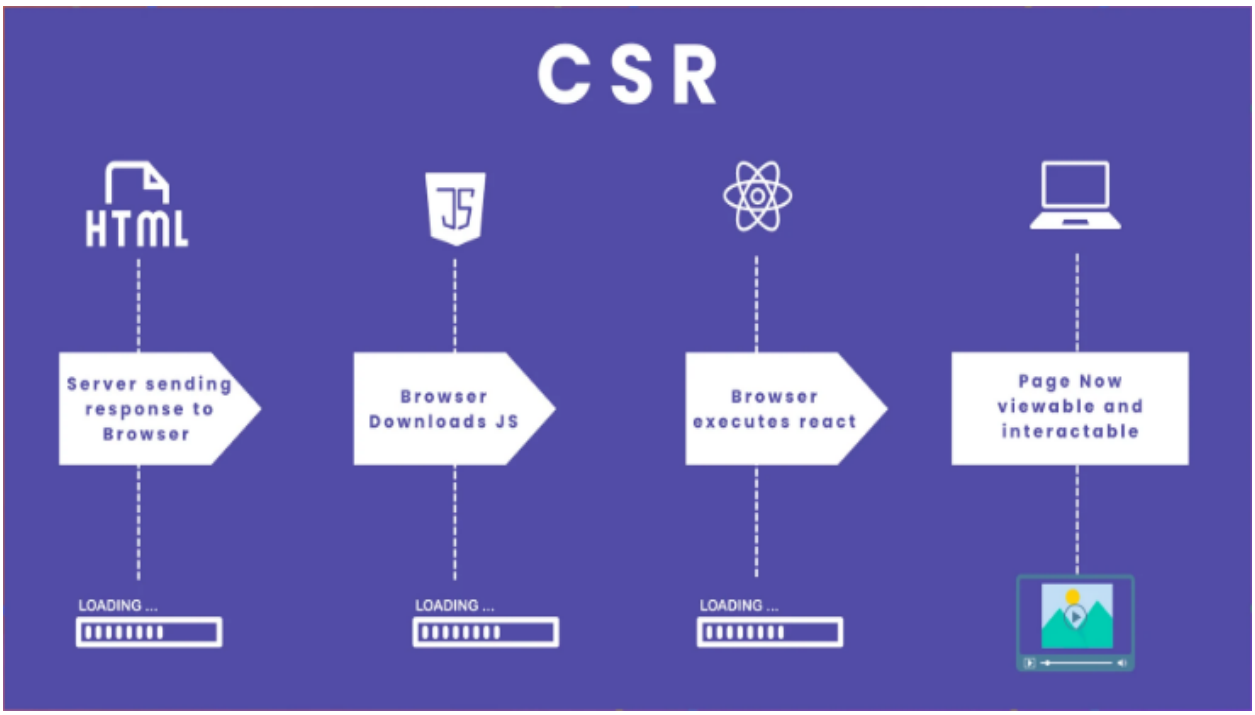

Рис. 1 Этапы формирования интерфейса при клиентском рендеринге (CSR)

Статическая генерация (SSG)

Статическая генерация предусматривает формирование HTML-страниц заранее, на этапе сборки (build). В этом случае при деплое приложения генерируется статический HTML для каждого маршрута (страницы) на основе текущих данных. При запросе от клиента сервер просто отдает готовый HTML-файл (возможно, с минимальным JS для интерактивности). Преимущество SSG – максимально быстрая отдача контента: время до отображения страницы минимально, так как нет задержки на серверное выполнение логики. По сути, SSG обеспечивает производительность традиционных статических сайтов – быструю начальную загрузку и отличный пользовательский опыт. Соответственно, Core Web Vitals метрики (LCP, CLS и пр.) у статических страниц обычно наилучшие [5]. SEO-показатели также выигрывают: поисковые системы получают полностью сформированный контент, который легко индексировать. Недостатки SSG проявляются при частом обновлении данных: поскольку страницы статичны, изменения контента требуют повторной генерации и деплоя сайта. Для сайтов с очень большим числом страниц SSG может привести к длительной сборке приложения [5]. Кроме того, чисто статические сайты не подходят для персонализированного контента или данных реального времени. Новейшие подходы, такие как инкрементальная статическая регенерация (ISR), частично решают эту проблему, позволяя обновлять отдельные страницы по расписанию или по запросу, но ISR является расширением SSG. В целом, SSG идеален для контента, который обновляется не слишком часто: он обеспечивает лучшую скорость загрузки и показатели SEO, однако менее гибок для динамических данных. Рисунок 2 иллюстрирует жизненный цикл SSG: от генерации HTML на этапе сборки до доставки пользователю через CDN.

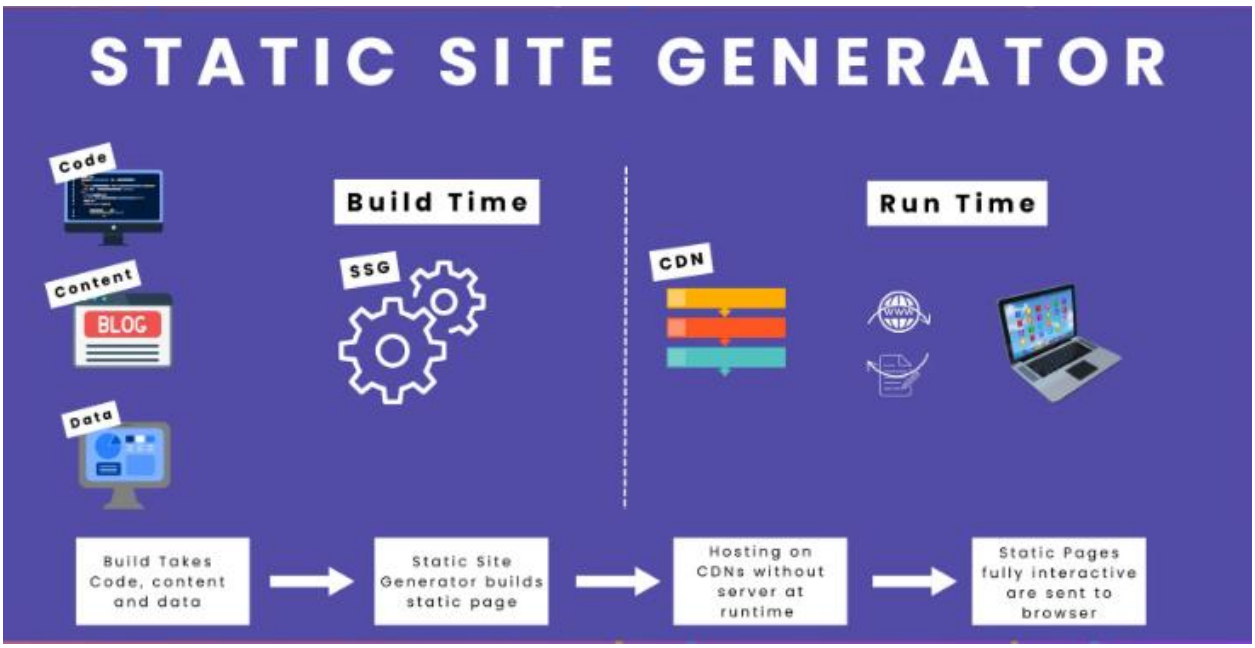

Рис. 2 Этапы генерации и доставки контента
при использовании статической генерации (SSG)

Серверный рендеринг (SSR)

При серверном рендеринге HTML-страница генерируется приложением на сервере на каждый входящий запрос. То есть, когда пользователь запрашивает страницу, сервер выполняет код приложения (например, React-компоненты) и формирует финальный HTML, уже содержащий актуальные данные, после чего отправляет его клиенту. Браузер пользователя получает полностью сформированную страницу и может сразу отобразить контент. Преимущества SSR состоят в том, что пользователь видит готовую страницу быстрее, чем при CSR, поскольку не нужно ждать загрузки всех данных через AJAX – основное содержимое уже встроено в HTML [4]. Для часто обновляемых данных SSR гарантирует, что пользователь и поисковый бот всегда получают свежий, актуальный контент [5]. С точки зрения SEO, SSR существенно упрощает индексацию: роботы сразу видят полный HTML со всем текстом и метаданными. Согласно документации Google, серверный или предварительный рендеринг устраняет проблемы индексации JavaScript и избавляет от медленной двухволновой индексации, обеспечивая более плавное и быстрое попадание контента в поисковый индекс [1]. Кроме того, SSR улучшает важные показатели загрузки: за счет отдачи готового HTML снижается время до первого отрисованного контента (FCP) и LCP [4], а также уменьшается вероятность крупных сдвигов макета (CLS), так как основной контент известен браузеру сразу [1]. Недостатки SSR связаны с возросшей нагрузкой на сервер. Поскольку каждое отображение требует вычислений на сервере, время до первого байта (TTFB) может увеличиваться [5]. Например, при высокой нагрузке на сайт генерация страниц на лету способна замедлить отклик сервера. SSR также требователен к инфраструктуре: для отрисовки большого числа страниц нужны производительные серверы или масштабирование на несколько узлов, что повышает стоимость обслуживания [1]. Еще один нюанс – время до интерактивности: хотя пользователь сразу видит содержимое, JS-бандл все равно должен загрузиться и "гидратировать" страницу (привязать обработчики событий). До завершения гидратации пользовательские действия могут не обрабатываться, что создает иллюзию задержки отклика [1]. Таким образом, SSR представляет собой компромисс: он сочетает относительную свежесть данных и SEO-дружелюбность, близкие к SSG, с более

высокой стоимостью генерации контента, присущей динамическим приложениям. На практике оптимальные результаты достигаются комбинированием SSR с другими подходами (например, кэшированием или частичной статической генерацией) в зависимости от характера данных. На рисунке 2 показан процесс серверного рендеринга: HTML формируется на сервере, затем происходит клиентская гидратация и взаимодействие. Таким образом, SSR позволяет значительно ускорить отображение первичного контента и улучшить SEO, особенно на медленных сетях. Однако его архитектура предполагает участие сервера при каждом запросе, в отличие от SSG, где страницы формируются заранее. На рисунке 4 приведено наглядное сравнение архитектурных подходов SSR и SSG. SSR включает в себя формирование HTML на сервере с возможным обращением к базе данных, тогда как при SSG все необходимые ресурсы уже собраны на этапе сборки и отдаются как статические файлы.

Рис. 3 Этапы формирования интерфейса при серверном рендеринге (SSR)

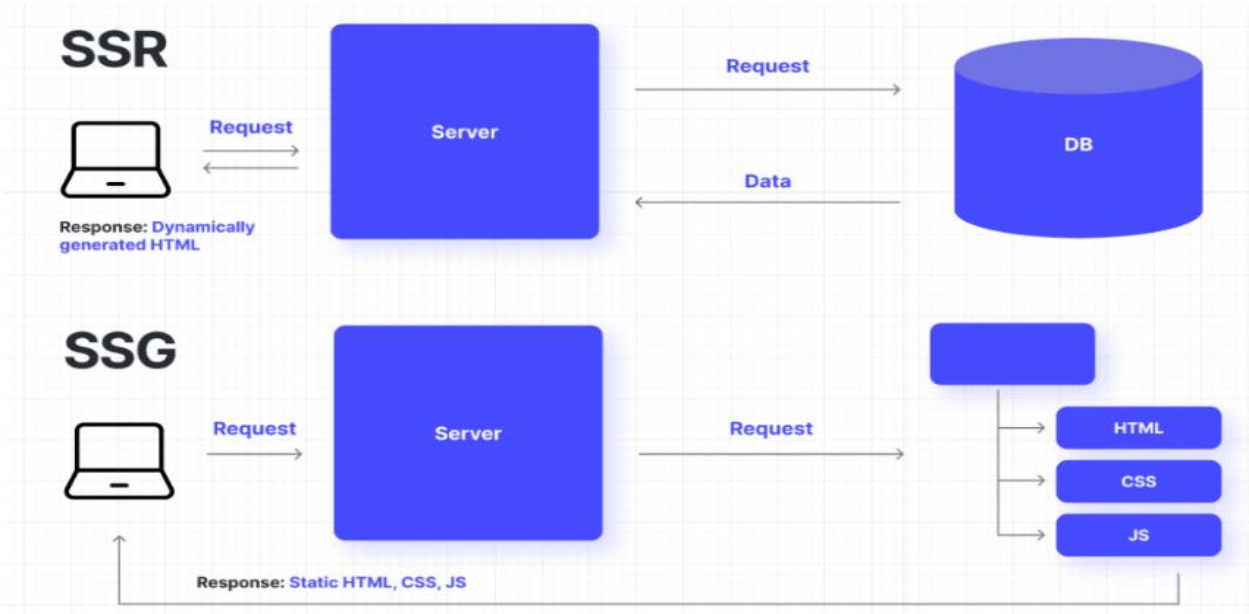

Рис. 4 Сравнение архитектурных схем SSR и SSG

SSR в Next.js: принципы и архитектура

Next.js – фреймворк на основе React, предлагающий разработчикам удобные инструменты для реализации серверного рендеринга и гибридного подхода к генерации страниц. Архитектура Next.js изначально спроектирована как универсальная (isomorphic) – один и тот же код React-компонентов может выполняться как на стороне сервера, так и в браузере. Это достигается за счет встроенного механизма, который решает, где рендерить каждый компонент, и механизма гидратации, связывающего серверный HTML с клиентским React-приложением.

В классической модели Next.js (до версии 13) разработчик определяет страницы в директории pages/. Для каждой страницы можно указать способ получения данных: например, экспортировать функцию getServerSideProps для SSR или getStaticProps для SSG. Если страница использует SSR, Next.js на каждый запрос вызывает функцию getServerSideProps на сервере, которая получает необходимые данные (например, из базы или внешнего API) и возвращает их в виде props для React-компонента страницы [6]. Затем Next.js выполняет рендеринг React-компонента страницы с этими данными на сервере (с помощью движка Node.js) и формирует HTML-код. Этот HTML отправляется пользователю вместе с ссылками на сгенерированные JS-скрипты. В результате браузер быстро отображает готовый HTML-контент, а затем загружает и выполняет React-бандл, который "оживляет" страницу (привязывает события и делает интерфейс интерактивным).

Архитектура Next.js включает в себя ряд механизмов, направленных на повышение эффективности серверного рендеринга. Одним из таких механизмов является файловая маршрутизация: каждая страница автоматически становится маршрутом, а благодаря встроенному разделению кода (code splitting), пользователю передаётся только JavaScript, необходимый для конкретной страницы. Это уменьшает объём начального бандла и ускоряет процесс загрузки и гидратации.

Важную роль играет система кэширования. При повторных запросах одних и тех же страниц Next.js может использовать заранее сформированный HTML, сократив затраты на рендеринг. Кроме того, применяется режим ISR (Incremental Static Regeneration), при котором страницы создаются как статические, но периодически обновляются на сервере. Это позволяет снизить нагрузку и сохранить актуальность данных без полной генерации при каждом обращении.

Next.js также предоставляет встроенные возможности для реализации серверной логики – так называемые API-маршруты. Они позволяют выполнять запросы к базе данных и другим источникам прямо в рамках самого приложения, исключая необходимость в отдельном backend-сервере. Такая архитектура делает SSR более компактным и менее зависимым от внешних сетевых запросов.

С выходом новых версий фреймворк получил поддержку React 18 и технологий стримингового рендеринга. HTML может отдаваться частями по мере готовности, а не целиком после полной генерации страницы, что позволяет пользователю начать взаимодействие с контентом быстрее. Подобная схема реализует принципы частичного пререндеринга и объединяет достоинства SSR и SSG. Несмотря на то, что эти возможности всё ещё развиваются, они уже сейчас позволяют уменьшить задержку отображения первого контента и повысить отзывчивость интерфейса.

В целом, работа архитектуры SSR в Next.js строится следующим образом: браузер инициирует запрос, сервер формирует HTML с использованием необходимых данных и отправляет его клиенту. Затем React восстанавливает состояние и активирует интерфейсные события. Такой подход позволяет объединить высокую скорость начального рендеринга с интерактивностью, характерной для SPA-приложений. Стоит отметить, что Next.js не

ограничивается SSR: разработчик может выбирать подход для каждой страницы. Например, публичные страницы с постоянным контентом можно собрать статически (SSG) для максимальной скорости, а страницы с часто меняющимися данными или персонализацией – рендерить через SSR при каждом запросе. В гибридном приложении Next.js одни страницы работают в режиме SSR, другие – SSG или даже чистого CSR, что дает большую гибкость. Именно благодаря этой архитектуре Next.js стал «де-факто» стандартом SSR в экосистеме React, позволяя реализовать SEO-дружественные и быстрые веб-приложения с минимальными усилиями.

Влияние SSR на скорость загрузки и SEO

Внедрение SSR существенно сказывается на показатели скорости веб-страниц и на эффективность их поискового продвижения. Рассмотрим подробнее, как серверный рендеринг влияет на ключевые метрики производительности (в том числе Core Web Vitals) и на различные аспекты SEO.

Ускорение начальной загрузки.

Главный выигрыш SSR – сокращение времени, которое пользователь ждет появления контента на экране. За счет того, что HTML формируется сервером, браузер может начать рендеринг разметки сразу по получении первых данных от сервера, не дожидаясь загрузки и исполнения большого JS-бандла. Это заметно улучшает перцептивную производительность: пользователь видит структуру страницы и основной контент значительно быстрее, чем в архитектуре CSR. В терминах метрик, SSR снижает First Contentful Paint (FCP) – время до появления первого текстового или графического элемента – и ускоряет Largest Contentful Paint (LCP) – время до отображения самого крупного по размеру элемента (например, hero-изображения или заголовка) [4]. Согласно документации Angular Universal, SSR может существенно улучшить показатели LCP по сравнению с клиентским рендерингом, особенно на медленных сетях или устройствах [4]. Это подтверждают практические замеры: например, в независимом тестировании SSR-фреймворков Next.js, Nuxt.js и Remix показали время LCP порядка ~1,2 с, тогда как чисто клиентские приложения обычно имеют больший LCP [7]. Чем быстрее LCP, тем выше вероятность удержать внимание пользователя и тем лучше Core Web Vitals скоринг страницы.

Помимо LCP, SSR положительно влияет на Cumulative Layout Shift (CLS) – метрику стабильности макета. Поскольку значительная часть содержимого и стилей загружается сразу с серверным HTML, уменьшается количество "прыжков" интерфейса при дозагрузке ресурсов. В SSR-страницах размеры изображений и блоков чаще всего известны сразу, и контент не появляется внезапно позднее, вытесняя другие элементы. В результате суммарное смещение макета (CLS) ниже, чем в сценариях CSR, где контент вставляется динамически. Компания Google отмечает, что страницы с SSR обычно демонстрируют лучшие значения LCP и CLS – двух из трех метрик Core Web Vitals [1]. Улучшение этих показателей не только делает работу сайта комфортнее для пользователя, но и служит фактором ранжирования: начиная с 2021 года Google учитывает Core Web Vitals при определении позиции сайта в поисковой выдаче.

Однако не все метрики однозначно выигрывают от SSR. За получение готового HTML на сервере приходится платить увеличением задержки перед началом передачи страницы. Time to First Byte (TTFB) – время до первого байта ответа сервера – при SSR может быть больше, чем при отдаче статического файла. Генерация страницы на сервере (особенно при сложной логике или медленных запросах к базе) добавляет сотни миллисекунд к TTFB [5]. Например, сравнение разных SSR-фреймворков показало, что TTFB у оптимизированного Next.js приложения может составлять ~63 мс (благодаря эффективному рендерингу и кэшированию), тогда как у аналогичного приложения на Nuxt без оптимизации TTFB достигал 438 мс [7]. Такой разброс свидетельствует, что производительность SSR сильно зависит от

реализации и инфраструктуры. Увеличенный TTFB может негативно сказаться на восприятии быстродействия и даже слегка затормозить LCP, если сервер очень медленно отвечает. Поэтому важно оптимизировать серверный код (использовать CDN, кешировать результаты SSR где возможно, минимизировать внешние вызовы во время рендеринга).

Другой сложный момент – время до интерактивности. Метрики вроде First Input Delay (FID) или современного показателя Interaction to Next Paint (INP) отражают, насколько быстро страница реагирует на первое действие пользователя. При SSR пользователь получает статический HTML, но полноценная работа интерфейса начинается только после загрузки и выполнения JS. В этот промежуток нажатия на кнопки могут игнорироваться. Исследования показывают, что SSR иногда увеличивает суммарную задержку взаимодействия, так как браузеру приходится обрабатывать большой объем скриптов для гидратации уже отрисованного содержимого [3]. Другими словами, страница кажется загруженной, но, если пользователь слишком быстро попытается взаимодействовать, приложение еще "не готово". Это может ухудшить показатель INP. Таким образом, SSR ускоряет метрики отображения (относится к loading в Core Web Vitals), но может негативно влиять на метрики взаимодействия (interactivity), если приложение тяжеловесное. Для смягчения этого эффекта современные подходы предлагают разбивать JavaScript-бандлы, откладывать выполнение не критичного кода и использовать приёмы частичной гидратации. Next.js продолжает развиваться в этом направлении (React 18 Streaming, выборочные клиентские компоненты и пр.), чтобы уменьшить влияние SSR на время до интерактивности.

Практическая реализация SSR и сопутствующих техник была проведена в рамках выпускной квалификационной работы Куревина М.А. В эксперименте сравнивались метрики производительности до и после внедрения SSR (в сочетании с lazy loading, динамическими импортами и адаптивной логикой) [11].

Измерения проводились на главной странице веб-приложения с помощью Google Lighthouse в режиме Slow 4G, с расчётом медианных значений по 7 прогонам. Результаты приведены в таблице 1.

Таблица 1

Сравнение производительности до и после внедрения SSR
и дополнительных оптимизаций (главная страница)

Метрика	До оптимизации	После оптимизации	Изменение
FCP (сек.)	2.5	2.7	+ 0.2 (почти без изменения)
LCP (сек.)	8.0	5.5	− 2.5 (31%)
TBT (мс)	1130	440	− 690 (61%)
Speed Index (сек.)	6.0	5.2	− 0.8 (13%)
CLS	0	0	без изменений
Performance score	44	59	+ 15 (34%)
SEO score	91	91	без изменений

SEO и индексация.

Одной из основных причин популярности SSR стало улучшение видимости приложений в поисковых системах. Когда SSR-страница загружается, она уже содержит весь необходимый контент: текст, ссылки, заголовки, мета-теги, данные Open Graph для социальных сетей и т.д. Поисковый робот (например, Googlebot) при обращении к странице, отрендеренной на сервере, сразу получает полноценный HTML-документ, содержащий основной контент и метаинформацию. Это обеспечивает мгновенную доступность содержимого для индексации, в отличие от клиентского рендеринга, где робот сталкивается

лишь с шаблонным HTML и ссылками на JavaScript-бандлы, требующие дополнительного выполнения. SSR устраняет необходимость в «двухэтапной» индексации, когда поисковой системе приходится сначала обработать JS, чтобы извлечь контент, а затем повторно возвращаться к странице. Благодаря этому новые и обновлённые страницы быстрее попадают в поисковую выдачу, что особенно важно для сайтов с часто обновляемым контентом.

Кроме того, серверный рендеринг повышает универсальность и достоверность представления сайта для различных поисковых и социальных роботов. Хотя современные версии Googlebot хорошо обрабатывают JavaScript, менее продвинутые сканеры, включая те, что используются в социальных сетях (например, Facebook Crawler или Twitterbot), могут не выполнять JS вообще. SSR обеспечивает, чтобы такие системы получали ключевые элементы страницы и корректно отображали превью, извлекая Open Graph-теги напрямую из HTML. Это повышает качество сниппетов при распространении ссылок и способствует росту охвата в мессенджерах и социальных платформах.

Наконец, SSR улучшает полноту индексирования контента. Во многих SPA-приложениях данные подгружаются частями, например, при прокрутке страницы или взаимодействии пользователя с элементами интерфейса. Поисковый бот может не активировать эти действия и, соответственно, не получить доступ ко всем данным. При SSR большая часть контента, доступного пользователю при первом открытии страницы, уже включена в HTML-ответ сервера, что гарантирует более полное сканирование и индексацию ресурса.Кроме технической доступности контента, SSR влияет на поведенческие факторы, которые косвенно связаны с SEO. Быстрая загрузка и рендеринг снижают показатель отказов (bounce rate) – меньше пользователей закрывают страницу, не дождавшись загрузки [1]. Пользователи дольше остаются на сайте, больше просматривают страниц, что поисковые алгоритмы могут трактовать как признак качества и релевантности. Исследования указывают на прямую зависимость: ускорение сайта на несколько секунд способно заметно повысить органический трафик за счет улучшения позиций в выдаче и увеличения конверсии пользователей в посетителей, которые просматривают больше страниц. В этом смысле SSR выступает как инструмент улучшения UX, который в свою очередь приводит к улучшению SEO-метрик.

Важно отметить, что SSR не является единственным способом обеспечить SEO-дружественность. Статическая генерация (SSG) зачастую дает еще лучшие результаты по скорости и ничем не уступает SSR с точки зрения доступности контента для поисковиков. Однако SSG применима не всегда – когда контент слишком динамичный или зависит от запроса (например, персональные данные, результаты поисковых фильтров), SSR остается более подходящим решением. Некоторые проекты используют комбинацию: критически важные для SEO страницы (например, главная, разделы с продуктами) генерируются статически, а второстепенные или часто меняющиеся страницы рендерятся на сервере по запросу [7]. Такой подход рекомендуют и эксперты: комбинировать стратегии рендеринга в зависимости от типа контента, чтобы достичь баланса между скоростью и актуальностью данных [5].

Подводя итог, серверный рендеринг в целом положительно влияет на SEO благодаря устранению препятствий для индексирования и ускорению сайта. Next.js и аналогичные фреймворки, реализующие SSR, доказали на практике свою эффективность: переход от чистого React (CSR) к Next.js (SSR) способен существенно повысить поисковый трафик и рейтинг страниц [2,8]. Тем не менее, максимальный эффект достигается при грамотной реализации SSR: оптимизации серверного кода, использовании кеширования, мониторинге Core Web Vitals и учете потенциальных задержек гидратации. В следующем разделе мы сравним Next.js с другими фреймворками, чтобы понять, насколько эффективно и удобно в каждом из них реализован SSR и какие результаты они демонстрируют.

Сравнение Next.js с другими фреймворками (Nuxt.js, Angular Universal, Remix)

Фреймворк Next.js стал популярным представителем SSR-решений в мире React, но у него есть аналоги и в других экосистемах. Рассмотрим, как реализован серверный рендеринг и решены сопутствующие задачи в Nuxt.js (Vue.js), Angular Universal (Angular) и Remix (еще один React-фреймворк), а также сравним их с Next.js с точки зрения производительности и SEO.

Nuxt.js (Vue.js) – это фреймворк, вдохновленный Next.js, но для экосистемы Vue.js. Nuxt.js изначально позиционируется как инструмент для универсальных приложений на Vue.js: по умолчанию используется SSR (т.н. режим "universal"), с возможностью статической генерации страниц. Архитектурно Nuxt очень похож на Next.js: файловая маршрутизация, автоматический код-сплиттинг, встроенные методы для SSR и SSG. Преимущество Nuxt – интеграция в Vue-стек: разработчики Vue получают аналогичные возможности по SEO и скорости, что и React-разработчики в Next. Nuxt обеспечивает быструю начальную загрузку и улучшенное SEO за счет серверного рендеринга и генерации контента заранее [8]. Согласно документации, SSR в Nuxt "позволяет отдавать страницы в виде полностью сформированного HTML, что приводит к более быстрым начальным загрузкам и улучшению SEO, поскольку поисковые системы могут легко индексировать предрендеренный контент" [8]. В бенчмарках производительности Nuxt 3 показывает результаты, сопоставимые с Next.js: оба фреймворка достигают близких значений LCP (~1 с с небольшим) и других метрик на тестовых проектах [7]. Один из тестов выявил, что Next.js и Nuxt 3 имели схожие показатели производительности, тогда как, например, Angular заметно отставал [9]. Это говорит о высоком уровне оптимизации Nuxt: несмотря на различие в используемом фреймворке UI (Vue vs React), итоговая скорость SSR-страниц и их UX примерно равны Next.js. С точки зрения разработчика, выбор между Next и Nuxt часто сводится к предпочтению React или Vue, так как по возможностям SSR и улучшению SEO они оба эффективны. В Nuxt также присутствуют возможности SSG (через команду nuxt generate) и частичного рендеринга. Таким образом, Nuxt.js представляет достойную альтернативу Next.js для тех, кто работает с Vue: он обладает аналогичным набором функций (SSR, SSG, код-сплиттинг, маршрутизация) и столь же активно улучшает показатели производительности и поисковой оптимизации приложения [8].

Angular Universal – это технология для добавления SSR в приложения на Angular. В отличие от React и Vue, где SSR реализуется внешними фреймворками, Angular Universal интегрируется непосредственно в Angular (начиная с Angular 4+). Angular Universal позволяет рендерить Angular-приложение на Node.js (или Deno) сервере, создавая HTML, который затем отправляется клиенту. В контексте SEO Angular Universal был ответом на проблему "невидимости" классических Angular SPA для поисковиков. Использование Angular Universal даёт улучшение SEO за счет отдачи контента поисковым ботам и ускорение загрузки для пользователя. Официальная документация Angular отмечает, что SSR повышает SEO, облегчая сканирование и индексирование содержимого приложений на Angular [4]. Также подчеркивается улучшение Core Web Vitals: уменьшение FCP/LCP и CLS при использовании SSR [4]. Однако на практике Angular Universal не всегда так же быстр, как Next или Nuxt. Angular – более тяжелый фреймворк, и даже после серверного рендеринга размер бандла и сложность приложения влияют на время интерактивности. По результатам независимого тестирования, Angular SSR продемонстрировал более низкие показатели производительности по сравнению с Next.js и Nuxt.js [9]. В тесте Pau Sanchez (2024) Angular существенно уступил как Svelte (лидеру теста), так и Next/Nuxt, по времени обработки запросов на сервере и метрикам скорости на клиенте [9]. Это может объясняться тем, что Angular выполняет больше работы и на сервере, и при гидратации. Тем не менее, Angular Universal широко используется в корпоративных приложениях, где SEO важен, а экосистема уже построена вокруг Angular. С точки зрения возможностей, Angular Universal поддерживает и прямой SSR на каждый запрос,

и предварительный рендеринг (аналог SSG) для определенных маршрутов. Инструментарий Angular CLI позволяет разработчику довольно легко добавить SSR в проект (ng add @angular/ssr). Таким образом, Angular Universal закрывает разрыв между Angular-приложениями и поисковыми системами, хотя в плане чистой скорости рендеринга React/Vue решения могут быть эффективнее. Выбор здесь обычно зависит от исходного стека: миграция с Angular на другой фреймворк – трудоемкий процесс, поэтому проекты на Angular предпочитают включить Universal, чтобы достичь приемлемого SEO и ускорения, даже если абсолютные показатели чуть ниже, чем у Next.js.

Remix – относительно новый фреймворк (релиз в 2021 г.) на базе React, который сделан с упором на серверное исполнение логики. В отличие от Next.js, Remix не предлагает статической генерации – он разработан как SSR-by-default. В Remix каждый запрос обрабатывается на сервере (Node.js, Cloudflare Workers или др. окружении), где выполняются так называемые Loaders (функции получения данных) и затем страница рендерится с этими данными. Remix использует концепцию вложенных маршрутов и поддерживает потоковую отдачу HTML (стриминг), что позволяет отображать контент по частям. С точки зрения SEO и производительности, Remix демонстрирует себя с лучшей стороны: благодаря тотальному SSR страницы Remix имеют быстрый initial load и хорошо индексируются. В обзорах отмечается, что Remix обеспечивает очень быстрые первые загрузки и отличный SEO, отчасти за счет продвинутых возможностей стриминга HTML и эффективного управления кэшированием данных [10]. В сравнении с Next.js, Remix убирает из уравнения выбор между SSG/SSR – разработчик просто пишет серверную логику для каждого маршрута, а фреймворк сам решает, как оптимально доставить контент. Бенчмарки производительности показывают, что Remix сопоставим с Next.js и Nuxt. Например, в тестах Enterspeed Remix имел LCP около 1,2 с – на уровне Next и Nuxt [7], а TTFB в районе 136 мс (несколько выше, чем у Next, но вполне конкурентный) [7]. Эти данные говорят о том, что Remix достигает целей SSR – быстрой отрисовки и готовности к SEO – не хуже более зрелых решений. Отличия Remix проявляются больше в модели разработки: Remix стремится использовать возможностей Edge-вычислений (распределенного запуска ближе к пользователю), имеет встроенную маршрутизацию на основе React Router с вложенными маршрутами, что удобно для сложных приложений. Но отсутствие SSG делает Remix менее подходящим для контента, который мог бы кешироваться долгое время; в таких случаях Next.js с ISR имеет преимущество.

Next.js vs конкуренты: если обобщить, все перечисленные фреймворки (Next, Nuxt, Angular Universal, Remix) решают схожие задачи – улучшают загрузку страниц и SEO через SSR – но с разным акцентом:

- Производительность. По показателям скорости рендеринга Next.js, Nuxt.js и Remix находятся примерно на одном уровне высоких результатов [7]. Angular Universal несколько отстает по эффективности, хотя тоже значительно ускоряет Angular-приложения по сравнению с CSR. Некоторые новейшие фреймворки, такие как SvelteKit или Astro (не рассматриваются подробно в статье), могут превосходить перечисленные по скорости, но они менее распространены. В реальных условиях конечная производительность часто зависит не столько от выбора между Next/Nuxt/Remix, сколько от качества реализации конкретного проекта и настройки инфраструктуры.
- SEO. Все рассматриваемые решения доказали способность улучшать SEO. Практически в каждом есть подтверждение: Next.js/Remix – высокие рейтинги и успешные кейсы миграции с чистого React, Nuxt – позиционирование как "SEO-friendly" фреймворка для Vue [8], Angular Universal – официальная поддержка от Google для решения SEO-проблем Angular [4]. Таким образом, с точки зрения поискового продвижения любой из этих инструментов при правильном использовании обеспечит индексируемость и высокое качество

страниц. Детали могут заключаться в удобстве управления мета-тегами: Next.js и Nuxt имеют встроенные компоненты для <head> (NextHead/Vue Meta), Remix позволяет управлять заголовками через специальные хелперы, Angular Universal требует ручного включения мета-сервисов. Но все они решают задачу выдачи нужных метаданных ботам.

- Удобство и экосистема. Next.js выигрывает за счет зрелой экосистемы и гибридности: он позволяет комбинировать SSR и SSG, имеет богатую документацию и сообщество. Nuxt.js – лучшее решение для тех, кто предпочитает Vue; он также достаточно зрелый (первая версия вышла в 2017 г., актуальная Nuxt 3 в 2022 г.). Angular Universal – скорее дополнение к большому фреймворку, его выбирают, когда проект уже на Angular. Remix – новаторский подход, еще набирающий популярность; его выбирают энтузиасты React, желающие более современный взгляд на SSR, или для использования на edge-платформах. Next.js и Nuxt.js также предлагают множество дополнительных возможностей (Image Optimization, встроенные API, плагины и модули и т.п.), тогда как Remix фокусируется на минимализме ядра (но расширяем через утилиты).
- Результаты в особых сценариях. Стоит упомянуть, что масштабируемость SSR – важный фактор. SSR нагружает сервер, и при всплесках трафика может быть узким местом. В этом плане Next.js и Nuxt.js предлагают стратегии ISR и кеширования, Angular Universal можно запустить на кластеризованных Node-серверах, Remix легко деплоится на бессерверные функции (например, Vercel, Cloudflare Workers) для горизонтального масштабирования. Каждый фреймворк имеет решения для высокой нагрузки, но подходы разные. Например, Next.js на Vercel автоматически распределяет SSR-запросы по региональным серверам, Remix тоже хорошо работает на распределенных платформах. Angular приложения могут быть развернуты на Vercel через адаптер. В итоге, все современные SSR-фреймворки могут обеспечить достаточную производительность на продакшене, но из коробки Next и Nuxt могут быть несколько более оптимизированы (судя по бенчмаркам).

В целом, Next.js держит лидерство в своём сегменте благодаря балансу возможностей: он гибридный, относительно простой в освоении и активно развивается Vercel'ом. Nuxt.js предоставляет аналогичный функционал для Vue с отличной производительностью и SEO, Angular Universal – узкоспециализированное решение для Angular-мира, а Remix – перспективный конкурент Next.js в React-мире, фокусирующийся на более современном подходе к SSR. Выбор конкретного инструмента зависит от стека и требований проекта, но с точки зрения конечной цели (ускорение загрузки и улучшение SEO) каждый из них при правильном применении дает существенный положительный эффект.

Заключение

Серверный рендеринг (SSR) зарекомендовал себя как эффективный подход для ускорения загрузки веб-страниц и повышения их SEO-показателей. Проведенный анализ показал, что по сравнению с чисто клиентским рендерингом (CSR) использование SSR позволяет сократить время до отображения основного содержимого (метрики FCP/LCP) и обеспечить стабильность макета (низкий CLS) за счет отдачи полностью сформированного HTML. В рамках фреймворка Next.js SSR реализован максимально прозрачно для разработчика и сочетается с другими техниками (SSG, ISR), что дает гибкость в выборе стратегии рендеринга для разных страниц приложения. Next.js с SSR позволяет преодолеть ограничения традиционных SPA: страницы загружаются быстрее и индексируются поисковыми системами без дополнительных усилий, что подтверждается как теоретическими сведениями, так и практическими кейсами миграции проектов на Next.js [2]. Аналогичные возможности предоставляют Nuxt.js (для Vue) и Remix (для React), а технология Angular Universal приносит преимущества SSR в экосистему Angular. Сравнение фреймворков показывает, что Next.js и родственные ему решения обеспечивают схожий высокий уровень

производительности (вплоть до LCP ~1 с на типичных страницах) и успешно решают задачи SEO-оптимизации, формируя на стороне сервера контент, готовый к индексированию [1,7].

SSR не лишен издержек: увеличение нагрузки на сервер, возможный рост TTFB и задержки до полного интерактивного состояния страницы. Тем не менее, при грамотной архитектуре эти риски минимизируются. Next.js предоставляет инструменты для кеширования результатов SSR, разделения кода и даже потокового рендеринга, уменьшая влияние SSR на время отклика. Таким образом, преимущества (ускорение сайта и улучшение SEO) заметно перевешивают недостатки, особенно для сайтов, где скорость и поисковая видимость критически важны (контентные проекты, e-commerce, маркетинговые лендинги и пр.).

На основе проведённого обзора можно сформулировать ряд практических рекомендаций для разработчиков, ориентирующихся на повышение производительности и SEO своих веб-приложений. Прежде всего, выбор стратегии рендеринга должен зависеть от характера контента. Для преимущественно статических страниц наиболее подходящей является статическая генерация (SSG), поскольку она обеспечивает максимальную скорость загрузки и высокую устойчивость к нагрузке. В то же время, для страниц, содержащих динамический или персонализированный контент, целесообразно использовать серверный рендеринг (SSR), особенно в случаях, когда важна актуальность информации и её доступность для поисковых систем. Современные фреймворки, такие как Next.js, позволяют гибко сочетать обе стратегии в рамках одного приложения, реализуя гибридную архитектуру.

В процессе использования SSR важно уделять внимание оптимизации серверного кода. Рекомендуется минимизировать время выполнения функций получения данных, таких как getServerSideProps или Loader, за счёт кэширования ответов от API, оптимизации запросов к базе данных и возврата только необходимого объёма информации. Для уменьшения нагрузки на сервер при повторных запросах следует использовать механизмы кеширования, включая CDN или возможности хостинговой платформы, такие как ISR в Vercel.

Однако даже при применении SSR необходимо контролировать показатели Core Web Vitals. Внедрение серверного рендеринга не гарантирует автоматического улучшения всех пользовательских метрик. Например, слишком тяжёлая гидратация может негативно повлиять на такие параметры, как FID или INP. Поэтому важно регулярно отслеживать ключевые показатели (LCP, CLS, INP) с помощью инструментов Lighthouse, WebPageTest и других аналитических платформ, а также внедрять техники разделения бандла и постепенной гидратации компонентов.

Также необходимо учитывать базовые SEO-практики. Наличие серверного рендеринга ещё не означает полноценной оптимизации. Критически важно заполнять мета-теги, описания, Open Graph-теги и микроразметку Schema.org, а также следить за тем, чтобы сгенерированные страницы содержали всю необходимую информацию для поисковых систем. Регулярные проверки с использованием Google Search Console или SEO-аудиторов позволяют убедиться, что страницы индексируются корректно и соответствуют требованиям современных алгоритмов ранжирования.

При выборе фреймворка на этапе проектирования следует учитывать поддержку SSR «из коробки». Для React-проектов разумным решением станет Next.js, для Vue – Nuxt.js, а для Angular – Angular Universal. В случае необходимости использования современных подходов SSR с React стоит обратить внимание на Remix. Сравнительный анализ показывает, что все упомянутые фреймворки предоставляют сходные возможности в части производительности и SEO, поэтому окончательное решение может зависеть от опыта команды, экосистемы и бизнес-требований.

Для проектов, где полная миграция на SSR-фреймворк невозможна, можно применять технику прогрессивного улучшения. Одним из решений является использование пререндеринга критических страниц при помощи сторонних сервисов, таких как Prerender.io.

Это позволяет частично компенсировать ограничения SPA с точки зрения индексации. Тем не менее, в долгосрочной перспективе переход на архитектуру с нативной поддержкой SSR обеспечивает лучшую масштабируемость и управляемость проекта.

В заключение, развитие технологий веб-рендеринга свидетельствует о стремлении объединить лучшие стороны различных подходов. SSR в сочетании с SSG и новыми методами (ISR, стриминг, частичная гидратация) позволяет создавать веб-приложения, одновременно быстрые и динамичные. Фреймворк Next.js стал одним из лидеров этой области, предоставляя удобный инструментарий для реализации таких приложений. Применение SSR в Next.js и аналогичных платформах доказало свою эффективность: страницы загружаются быстрее, удерживая пользователей, а поисковые системы получают качественный контент, повышая рейтинг сайта. Следование рассмотренным рекомендациям поможет максимально реализовать потенциал SSR, улучшив как пользовательский опыт, так и показатели SEO, что особенно важно в условиях высокой конкуренции в цифровой среде.

Список литературы:

1. Prerender.io. What Is Server-Side Rendering (SSR), and How It Affects SEO [Электронный ресурс] // Prerender Blog. – 2022. – URL: https://prerender.io/blog/what-is-srr-and-why-do-you-need-to-know/ (дата обращения: 18.04.2025).

2. Brian Hulela. Improving SEO When Migrating from React to Next.js with SSR [Электронный ресурс] // Medium.com. – 13.03.2025. – URL: https://medium.com/@brianhulela/improving-seo-when-migrating-from-react-to-next-js-with-ssr-e3008b917db2 (дата обращения: 18.04.2025).

3. Monus A. The Ultimate Guide To Server-Side Rendering (SSR) [Электронный ресурс] // DebugBear Blog. – 25.04.2022 (обновлено 29.03.2025). – URL: https://www.debugbear.com/blog/server-side-rendering (дата обращения: 18.04.2025).

4. Angular Developers. Server-side rendering (Angular Universal) [Электронный ресурс] // Angular Official Documentation. – 2023. – URL: https://angular.dev/guide/ssr (дата обращения: 18.04.2025).

5. Frog Proger. 5 стратегий рендеринга веб-страниц: как выжать максимум из вашего сайта [Электронный ресурс] // Proglib.io. – 2024. – URL: https://proglib.io/p/5-strategiy-renderinga-veb-stranic-kak-vyzhat-maksimum-iz-vashego-sayta-2024-08-08 (дата обращения: 18.04.2025).

6. Next.js Documentation. Server-side Rendering (SSR) [Электронный ресурс] // Next.js Official Docs. – URL: https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering (дата обращения: 18.04.2025).

7. Andreassen K. We measured the SSR performance of 6 JS frameworks – here’s what we found [Электронный ресурс] // Enterspeed Blog. – 2023. – URL: https://www.enterspeed.com/blog/we-measured-the-ssr-performance-of-6-js-frameworks-heres-what-we-found (дата обращения: 18.04.2025).

8. Naturally Team. Nuxt vs Next – Choosing The Best JS Framework [Электронный ресурс] // Naturally Tech Blog. – 2023. – URL: https://naturally.com/blog/nuxt-vs-next (дата обращения: 18.04.2025).

9. Sanchez P. Frontend SSR frameworks benchmarked: Angular, Nuxt, NextJs and SvelteKit [Электронный ресурс] // PauSanchez Blog. – 15.06.2024. – URL: https://www.pausanchez.com/en/articles/frontend-ssr-frameworks-benchmarked-angular-nuxt-nextjs-and-sveltekit/ (дата обращения: 18.04.2025).

10. Hygraph. Remix vs. Next.js: A side-by-side comparison [Электронный ресурс] // Hygraph Blog. – 2023. – URL: https://hygraph.com/blog/remix-vs-next (дата обращения: 18.04.2025).

11. Куревин М.А. Исследование эффективности современных методов веб-оптимизации производительности в разработке в условиях ограниченной пропускной способности: ВКР (магистр). Ярославль: ЯГТУ, 2025.