


**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

Исследование оптимизации рендеринга компонентов в ReactJS

Обучающийся / Student Агафонова Марина Владимировна
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники
Группа/Group P42081
Направление подготовки/ Subject area 09.04.04 Программная инженерия
Образовательная программа / Educational program Веб-технологии 2021
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Магистр
Руководитель ВКР/ Thesis supervisor Государев Илья Борисович, доцент, кандидат педагогических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан	
Агафонова Марина Владимировна	
29.05.2023	

(эл. подпись/ signature)

Агафонова
Марина
Владимировна

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Государев Илья Борисович	
29.05.2023	

(эл. подпись/ signature)

Государев Илья
Борисович

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Агафонова Марина Владимировна
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники
Группа/Group P42081
Направление подготовки/ Subject area 09.04.04 Программная инженерия
Образовательная программа / Educational program Веб-технологии 2021
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Магистр
Тема ВКР/ Thesis topic Исследование оптимизации рендеринга компонентов в ReactJS
Руководитель ВКР/ Thesis supervisor Государев Илья Борисович, доцент, кандидат педагогических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Исследование подходов к оптимизации рендеринга клиентских приложений, реализованных с помощью фреймворка ReactJS.

Задачи, решаемые в ВКР / Research tasks

- провести анализ зарубежных и отечественных научных источников по теме исследования для формирования представления о вариантах оптимизации рендеринга компонентов в клиентских фреймворках, - провести анализ внутреннего устройства работы библиотеки ReactJS, - определить элементы, которые играют ключевую роль в производительности приложения, времени отрисовки страницы приложения, - разработать клиент-серверное приложение для дальнейшего проведения на нем анализа эффективности оптимизации наиболее упоминаемых методов оптимизации, - провести анализ производительности ReactJS приложения после применения нескольких из самых популярных методов оптимизации, - описать нетривиальные случаи использования компонентов в масштабном коммерческом проекте.

Краткая характеристика полученных результатов / Short summary of results/findings

Проанализированы методы оптимизации рендеринга клиентских приложений, реализованных с помощью ReactJS фреймворка, изучено внутреннее устройство работы библиотеки ReactJS для определения ключевых моментов, которые влияют на время отрисовки страницы приложения, разработано приложение для применения методов оптимизации и сравнения показателей производительности приложения, обобщён опыт

работы с коммерческим проектом в аспекте влияния на производительность клиентского приложения.


Наличие публикаций по теме выпускной работы / Publications on the topic of the thesis

1. Агафонова М.В. Обзор влияния различных библиотек и различной реализации компонента на производительность ReactJS приложения // Научно-технические инновации и веб-технологии -2023. - Т. не указан. - № не указан. - С. не указаны (Статья)
2. Агафонова М.В. Обзор методов оптимизации рендеринга приложения, реализованного с помощью клиентских фреймворков // Современное образование: традиции и инновации -2022. - № 1. - С. 130-135 (Статья; РИНЦ)

Наличие выступлений на конференциях по теме выпускной работы / Conference reports on the topic of the thesis

1. Международная научно-практическая конференция "Современное образование в России: проблемы, решения, перспективы", 21.02.2022 - 21.02.2022 (Конференция, статус - международный)

Обучающийся/Student

Документ подписан	
Агафонова Марина Владимировна	
29.05.2023	

(эл. подпись / signature)

Агафонова
Марина
Владимировна

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Государев Илья Борисович	
29.05.2023	

(эл. подпись / signature)

Государев Илья
Борисович

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Агафонова Марина Владимировна
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники
Группа/Group P42081
Направление подготовки/ Subject area 09.04.04 Программная инженерия
Образовательная программа / Educational program Веб-технологии 2021
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Магистр
Тема ВКР/ Thesis topic Исследование оптимизации рендеринга компонентов в ReactJS
Руководитель ВКР/ Thesis supervisor Государев Илья Борисович, доцент, кандидат педагогических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Техническое задание:

- проанализировать методы оптимизации рендеринга клиентских приложений, реализованных с помощью ReactJS фреймворка,
- изучить внутреннее устройство работы библиотеки React JS для определения ключевых моментов, которые влияют на время отрисовки страницы приложения,
- разработать приложение для последующего применения методов оптимизации и сравнения показателей производительности приложения,
- проанализировать случаи из опыта работы с коммерческим проектом, которые влияли на производительность клиентского приложения.

Цель работы: исследование подходов к оптимизации рендеринга клиентских приложений, реализованных с помощью фреймворка ReactJS.

Задачи работы:

- провести анализ зарубежных и отечественных научных источников по теме исследования для формирования представления о вариантах оптимизации рендеринга компонентов в клиентских фреймворках,
- провести анализ внутреннего устройства работы библиотеки ReactJS,
- определить элементы, которые играют ключевую роль в производительности приложения, времени отрисовки страницы приложения,
- разработать клиент-серверное приложение для дальнейшего проведения на нем анализа эффективности оптимизации наиболее упоминаемых методов оптимизации,
- провести анализ производительности ReactJS приложения после применения нескольких из самых популярных методов оптимизации,

- описать нетривиальные случаи использования компонентов в масштабном коммерческом проекте.

Рекомендуемые материалы и пособия:

1. Князев И.В. Продвинутое кеширование и оптимизация веб-приложений с помощью технологии SWR // Sciences of Europe. 2021. № 73 (1). С. 47-49. URL: <https://cyberleninka.ru/article/n/prodvinutoe-keshirovanie-i-optimizatsiya-veb-prilozheniy-s-pomoschyu-tehnologii-swr>
2. Gowda V., Rangaswamy S. Addressing the Limitations of React JS // International Research Journal of Engineering and Technology (IRJET). 2020. № 7(4). С. 5065-5068. URL: <https://www.irjet.net/archives/V7/i4/IRJET-V7I4966.pdf>
3. Бэнкс А. React: современные шаблоны для разработки приложений / А. Бэнкс, Е. Порселло – СПб.:Питер, 2022 – 320 с.
4. Wieruch R. The Road to React / R. Wieruch. – Leanpub, 2020. – 228 с.
5. Правила хуков [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-rules.html>

Перечень подлежащих разработке вопросов:

ВВЕДЕНИЕ

1 СОВРЕМЕННОЕ СОСТОЯНИЕ ПРОБЛЕМЫ ОПТИМИЗАЦИИ РЕНДЕРИНГА КЛИЕНТСКИХ ПРИЛОЖЕНИЙ

1.1 Определение значимости производительности веб-приложений

1.2 Анализ оптимизации клиентских приложений отечественными и зарубежными учеными

2 ИССЛЕДОВАНИЕ УСТРОЙСТВА РАБОТЫ БИБЛИОТЕКИ REACTJS

2.1 Анализ работы фреймворка ReactJS

2.2 Анализ паттернов проектирования ReactJS приложений

2.3 Выделение ключевых особенностей в работе ReactJS фреймворка

3 ИССЛЕДОВАНИЕ СПОСОБОВ ОПТИМИЗАЦИИ РЕНДЕРИНГА ПРИЛОЖЕНИЯ, РЕАЛИЗОВАННОГО С ПОМОЩЬЮ REACTJS

3.1 Исследование инструментов измерения производительности страницы веб-приложения

3.2 Разработка клиент-серверного приложения с помощью ReactJS и Flask для проведения последующего сравнения методов оптимизации

3.3 Сравнение методов оптимизации для разработанного приложения

4 ИССЛЕДОВАНИЕ СЛУЧАЕВ НА КОММЕРЧЕСКОМ ПРОЕКТЕ, ПОВЛИЯВШИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ КЛИЕНТСКОГО ПРИЛОЖЕНИЯ

4.1 Рефакторинг проекта в связи с изменением подхода к реализации собственного компонента и переходом на иную библиотеку по управлению состоянием приложения

4.2 Сравнение производительности клиентского приложения от изначальной до конечной реализации

4.3 Анализ бесконечного ре-рендеринга компонента из-за неверного применения функции из библиотеки по управлению состоянием компонента

ЗАКЛЮЧЕНИЕ

Форма представления материалов ВКР / Format(s) of thesis materials:

Презентация

Дата выдачи задания / Assignment issued on: 06.02.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 31.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

Название организации-партнера / Name of partner organization: ООО «ДТ АЙТИ РУС»

Тема в области фундаментальных исследований / Subject of fundamental research: нет / not

Тема в области прикладных исследований / Subject of applied research: нет / not

СОГЛАСОВАНО / AGREED:

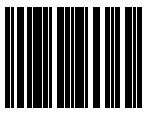
Руководитель ВКР/
Thesis supervisor

Документ подписан	
Государев Илья Борисович	
18.05.2023	

(эл. подпись)

Государев Илья
Борисович

Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Агафонова Марина Владимировна	
19.05.2023	

(эл. подпись)

Агафонова
Марина
Владимировна

Руководитель ОП/ Head
of educational program

Документ подписан	
Государев Илья Борисович	
22.05.2023	

(эл. подпись)

Государев Илья
Борисович

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
1 СОВРЕМЕННОЕ СОСТОЯНИЕ ПРОБЛЕМЫ ОПТИМИЗАЦИИ РЕНДЕРИНГА КЛИЕНТСКИХ ПРИЛОЖЕНИЙ.....	7
1.1 Определение значимости производительности веб-приложений	7
1.2 Анализ оптимизации клиентских приложений отечественными и зарубежными учеными	8
2 ИССЛЕДОВАНИЕ УСТРОЙСТВА РАБОТЫ БИБЛИОТЕКИ REACTJS.....	20
2.1 Анализ работы фреймворка ReactJS.....	20
2.2 Анализ паттернов проектирования ReactJS приложений.....	27
2.3 Выделение ключевых особенностей в работе ReactJS фреймворка...	29
3 ИССЛЕДОВАНИЕ СПОСОБОВ ОПТИМИЗАЦИИ РЕНДЕРИНГА ПРИЛОЖЕНИЯ, РЕАЛИЗОВАННОГО С ПОМОЩЬЮ REACTJS	31
3.1 Исследование инструментов измерения производительности страницы веб-приложения	31
3.2 Разработка клиент-серверного приложения с помощью ReactJS и Flask для проведения последующего сравнения методов оптимизации	35
3.2.1 Постановка задачи по разработке клиент-серверного приложения	35
3.2.2 Проектирование макета.....	39
3.2.3 Проектирование клиентского приложения.....	41
3.2.4 Проектирование серверного приложения.....	43
3.2.5 Сборка проекта в Docker и настройка CI/CD	44
3.3 Сравнение методов оптимизации для разработанного приложения .	49
3.3.1 Кэширование – применение React.memo	50
3.3.2 Кэширование – применение moize.....	52
3.3.3 Lazy loading («ленивая загрузка») изображений	53
3.3.4 Сравнение реализованных способов.....	55

4 ИССЛЕДОВАНИЕ СЛУЧАЕВ НА КОММЕРЧЕСКОМ ПРОЕКТЕ, ПОВЛИЯВШИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ КЛИЕНТСКОГО ПРИЛОЖЕНИЯ	58
4.1 Рефакторинг проекта в связи с изменением подхода к реализации собственного компонента и переходом на иную библиотеку по управлению состоянием приложения.	58
4.2 Сравнение производительности клиентского приложения от изначальной до конечной реализации	68
4.3 Анализ бесконечного ре-рендеринга компонента из-за неверного применения функции из библиотеки по управлению состоянием компонента	73
ЗАКЛЮЧЕНИЕ	75
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	76

ВВЕДЕНИЕ

В настоящее время разработка веб-приложений набирает большую популярность и спрос. Такими приложениями могут быть как сайты-визитки для компаний, так и онлайн-магазины и т.д. Один из ключевых факторов для успешного приложения – это пользовательский интерфейс (UI – user interface). Более того, UI должен быть как хорошо разработан визуально (привлекательный современный, а также интуитивный дизайн), так и профессионально сделан с хорошими показателями по производительности в рендеринге компонентов. Существует много клиентских фреймворков, которые предоставляют возможность реализовать необходимый UI. В качестве примера можно привести самые популярные фреймворки: ReactJS, Angular, Vue.js. В данной работе планируется рассмотрение именно ReactJS.

React является клиентским фреймворком, который гарантирует высокую производительность. Однако не при всех условиях достигается такая производительность. Как показывает практика, при неправильном подходе к разработке оптимизация производительности не работает должным образом.

Самой распространённой ситуацией является то, когда клиентская часть проекта разрабатывается по стандартным методам проектирования для таких типов приложений, но из-за специфики проекта страница веб-сайта может долго загружать или обновлять отображаемые данные. В худшем случае это приводит к тому, что пользователю отображается некорректная информация из-за задержек в рендеринге страницы.

Целью данной работы является исследование подходов к оптимизации рендеринга клиентских приложений, реализованных с помощью фреймворка ReactJS.

Объектом исследования выступают веб-технологии. Предметом исследования является компонент клиентского фреймворка ReactJS.

Для достижения поставленной цели необходимо было решить следующие задачи:

- 1) провести анализ зарубежных и отечественных научных источников по теме исследования для формирования представления о вариантах оптимизации рендеринга компонентов в клиентских фреймворках,
- 2) провести анализ внутреннего устройства работы библиотеки ReactJS,
- 3) определить элементы, которые играют ключевую роль в производительности приложения, времени отрисовки страницы приложения,
- 4) разработать клиент-серверное приложение для дальнейшего проведения на нем анализа эффективности оптимизации наиболее упоминаемых методов оптимизации,
- 5) провести анализ производительности ReactJS приложения после применения нескольких из самых популярных методов оптимизации,
- 6) описать нетривиальные случаи, которые встретились на практике с масштабным коммерческим проектом.

Теоретической основой проводимого исследования выступили работы российских (О.В. Бородин, В.А. Егунов, Р.А. Тороптин, Я.В. Зиновьев, Н.С. Рассказов, М.А. Митрохин, И.В. Князев) и зарубежных (V. Gowda, S. Rangaswamy, M.P. Reddy, S.P. Mishra, S.K. Mukhiya, H.K. Hung) ученых, посвященные оптимизации работы приложений, реализованных с помощью клиентских фреймворков.

Для решения задач, поставленных в рамках исследования, применялся комплекс теоретических методов исследования, таких как сравнительный анализ, обобщение, синтез, индукция и дедукция.

Основной информационной базой проводимого исследования послужили научные электронные библиотеки eLibrary, ResearchGate, Cyberleninka, официальная документация front-end и back-end решений для веб-приложений.

Теоретическая значимость проведенного исследования заключается:

- в сравнении наиболее популярных методов оптимизации рендеринга на одном приложении, выявлении, какой из примененных методов покажет наилучший результат,

- в обобщении и систематизации ключевых факторов, которые могут повлиять на производительность приложения, а именно – рендеринг компонентов.

Практическая значимость проведенного исследования заключается в реализации рассматриваемых методов на одном проекте, что позволит точно проследить, какой именно метод сработал в тех же условиях лучше всего.

Степень достоверности и апробация результатов.

Достоверность теоретических и практических результатов исследования обеспечиваются проведенным анализом области исследования, исследованием комплекса методов, адекватно цели и задачам исследования, его объекту и предмету исследования.

По теме выпускной квалификационной работы опубликованы следующие статьи:

- Агафонова М.В. Обзор методов оптимизации рендеринга приложения, реализованного с помощью клиентских фреймворков // Современное образование: традиции и инновации. – 2022. N. 1 – С. 130-135,

- Агафонова М.В. Обзор влияния различных библиотек и различной реализации компонента на производительность ReactJS приложения // Научно-технические инновации и веб-технологии. – 2023 (принята к публикации).

Было выступление на конференции «Современное образование в России: проблемы, решения, перспективы».

1 СОВРЕМЕННОЕ СОСТОЯНИЕ ПРОБЛЕМЫ ОПТИМИЗАЦИИ РЕНДЕРИНГА КЛИЕНТСКИХ ПРИЛОЖЕНИЙ

1.1 Определение значимости производительности веб-приложений

В настоящее время актуальна тема ускорения работы приложений или отдельных процессов. Даже небольшая разница во времени отработки может дать значительный результат. Для веб-приложений время загрузки страницы или обновление отдельных компонентов имеет критическое значение. Согласно опросу Unbounce, 70% пользователей с меньшей вероятностью будут совершать покупки, если время загрузки сайта превышает ожидания [1]. Кроме того, медленная загрузка веб-сайта может привести к другим проблемам, таким как низкая производительность и отток пользователей. Если рассматривать вопрос ранжирования в поисковых системах, то скорость загрузки сайта является одним из ключевых факторов. Компания Google отметила это еще в 2010 году, и с тех пор значимость показателя "page speed" только увеличивается. Таким образом, скорость загрузки веб-сайта является важным аспектом, который необходимо учитывать при разработке и оптимизации веб-приложений. Если рассматривать то, как быстроедействие влияет на SEO (англ. Search Engine Optimization), то можно выделить следующее:

- 1) поисковые системы напрямую замедляют скорость загрузки, повышая приоритет быстрых сайтов,
- 2) поисковые системы следят за поведением пользователей: если страницы реже посещают и чаще закрывают – рейтинг страницы понижается.

Высокая скорость загрузки стала первоочередной необходимостью, веб-сайты должны открывать почти мгновенно. И для такой производительности веб-приложения необходимо разработать соответствующее техническое решение.

Если рассуждать об оценке скорости загрузки сайта, то нужно учитывать ряд факторов таких, как качество интернет-соединения на стороне пользователя, настроек браузера, типа устройства. В среднем время загрузки

страницы должно составлять 2-3 секунды. Задержка в 5-7 секунд допустима, но все же чревата потерей небольшого процента пользователей. Если страница загружается более 10 секунд, то однозначно нужно предпринимать меры по ускорению загрузки.

На сегодняшний день доступно множество сервисов для проверки скорости веб-сайта, одним из самых популярных является PageSpeed Insights от компании Google. Данный сервис позволяет замерять скорость загрузки на ПК и мобильных устройствах, а также предлагает рекомендации по ускорению работы веб-приложения. Однако, несмотря на его популярность, рекомендации, предлагаемые сервисом, могут оказаться довольно поверхностными и не привести к значительному улучшению быстродействия сайта. Поэтому, для достижения максимальной производительности веб-приложения, рекомендуется использовать несколько сервисов для проверки скорости загрузки и проводить дополнительную оптимизацию кода, основываясь на результате анализа. Такой подход позволит добиться максимальной эффективности и ускорения работы веб-приложения.

1.2 Анализ оптимизации клиентских приложений отечественными и зарубежными учеными

Одной из причин медленного рендеринга страницы ReactJS-приложения может являться неверный выбор архитектурного стиля для приложения. Такой случай рассматривается в работе Р.А. Торопкина, Я.В. Зиновьева, Н.С. Рассказова, М.А. Митрохина в 2020 году [2]. В данной работе рассматриваются существующие методы оптимизации пользовательского интерфейса в веб-приложениях, направленные на уменьшение времени загрузки страницы. Из наиболее популярные методов, которые применяются на рынке, авторы выделили следующие способы оптимизации:

- уменьшение объема загружаемых данных,
- сохранение части информации на устройстве пользователя или кэширования,
- «визуальная» оптимизация веб-страницы.

Но авторы решили рассмотреть наиболее эффективные варианты клиентской оптимизации и их применение на реальном проекте. Если кэширование заключается в сохранении части информации на устройстве пользователя, то авторы рассматривают способ вычислений на стороне клиента. Чем больше сторонних библиотек используется в проекте, тем больше информации нужно передать на пользовательское устройство. Но эту проблему можно решить с помощью инструментов для сборки, таких как `gulp`, `Webpack`. Функционал данных инструментов позволяет в плане оптимизации следующее:

- указывать среду (при компиляции в сборке не будут включаться различные артефакты тестирования и разработки),
- минимизировать исходные JS-файлы, а также различные версии `source map` (исходная карта), размер которой можно уменьшить до двух раз,
- использовать различные плагины для оптимизации сборки, позволяющие формировать файлы формата `«.gz»`, импортировать только отдельные модули из библиотек,
- анализировать размеры используемых зависимостей,
- создавать дерево зависимостей и определять необходимые ресурсы для каждой страницы.

Помимо данных инструментов авторы сравнивают SPA и MPA решения, т.к. принцип их работы значительно различается: в случае с MPA отправляются запросы при каждом изменении на странице, что приводит к ее принудительному обновлению, тогда как в SPA используется динамическое обновление DOM-дерева, что позволяет исключить перезагрузку веб-страницы.

Также авторы отмечают, что в последнее время становится популярной «отложенная загрузка» — это оптимизация загрузки медиафайлов и компонентов, некритичных для отображения веб-страницы и взаимодействия с интерфейсом. Такая загрузка больше всего подходит для тех элементов,

которые располагаются за линией видимости пользователя и отображаются только после прокрутки. Подобная загрузка уже реализована во многих клиентских фреймворках, таких как Vue.js, Angular и React. Данные фреймворки позволяют настроить динамическую загрузку маршрутов и компонентов.

Важно помнить и о формате файлов, т.к. более экономичные форматы позволяют сократить объем передаваемого трафика. Авторы приводят в качестве примера формат WebP, который предоставляет возможность экономить до 34% при конвертировании картинки из формата JPG и 45% для формата PNG [3]. Однако, стоит учитывать, что только у 79,2% пользователей присутствует поддержка данного формата.

Помимо этого, авторы рассматривают способ, который позволяет уменьшить объем загружаемых файлов, снизить нагрузку на устройство пользователя при рендеринге страницы. Одним из таких способов является серверный рендеринг. Его суть заключается в том, что вся нагрузка по отображению страницы ложится на сервер, а ресурсы клиентского устройства освобождаются для иных процессов [4].

Одним из самых популярных способов для оптимизации работы приложения является кеширование, и авторы статьи описывают одну из вариаций – использование «сервис-воркеров». Реализация такого способа заключается в том, что сайт разделяется на две части: со статическим и динамическим контентом. При повторной загрузке скачивается только часть с уникальным контентом и объединяется со статическим [5].

Более продвинутое кеширование и оптимизация веб-приложения с помощью технологии SWR описано в работе И.В. Князева. Автор отмечает, что в современных веб-приложениях для кеширования данных используется стратегия инвалидации HTTP-кеша (RFC-5861). Этот протокол описывает то, как должны обновляться закешированные данные: они загружаются сразу же, а обновляются «на лету».

Существуют веб-приложения, в которых кэш-память может кэшировать редко используемые данные, что приводит к ситуации, когда обновление контента может произойти только после повторной загрузки данных на клиент в случае ошибки. Бывают ситуации, когда кеширование не имеет особого смысла. Автор статьи приводит пример, когда кэш содержит данные о пользователях и их действиях, а время выполнения запроса в браузере составляет несколько секунд. Некоторые же веб-сайты отказываются от кеширования в целях безопасности.

В свою очередь, технология SWR предоставляет возможность с минимальным количеством кода обрабатывать уже обновленный контент. Это является хорошим компромиссом между эффективностью веб-приложения и пользовательским опытом. Автор статьи подробно описывает и демонстрирует использование SWR: импорт функции `useSWR` из библиотеки SWR, создание пользовательского кэша с помощью `createCache`, использование `mutate` функции для проверки ключей из кэша и создание отдельных функций для получения регулярного выражения в качестве ключа, синхронизации своих кэшированных состояний.

Как отмечает автор, технология SWR в большой степени упрощает и ускоряет разработку. Один из преимуществ является то, что запрос кэшируется и отправляется на сервер всего лишь один раз, что положительно влияет на эффективность и производительность приложения [6].

В другой работе И.В. Князева, которая посвящена анализу работы приложения с использованием `server-side rendering`, описывается такой способ оптимизации, как серверный рендеринг, который упоминался в работе Р.А. Торопкина, Я.В. Зиновьева, Н.С. Рассказова, М.А. Митрохина, но не был применен для рассматриваемого проекта. И.В. Князев в своей статье проанализировал продвинутые функции NextJS, реализацию самого кода, миграцию, настройку и развертывание приложения с использованием современного `server side rendering` (рендеринга на стороне сервера).

Как отмечает автор, в JS-фреймворках (например, React и Angular) в основном используют рендеринг на стороне клиента (CSR – client side rendering). В таком случае сервер отправляет на страницу почти пустой файл HTML, за которым следуют все JS-файлы в одном пакете, который позже должен быть обработан в браузере пользователя для рендеринга DOM-дерева. На таких страницах начальная скорость загрузки низкая, поэтому пользователь видит пустой экран, пока не будет выполнен весь код JavaScript и не завершатся запросы к API. Но такой сценарий касается только первоначальной загрузки страницы. Последующие скорости загрузки будут быстрыми, т.к. дальнейшие изменения потребуют только обновления соответствующих разделов DOM-дерева.

SSR подход решает проблему начальной скорости загрузки в CSR, т.к. сервер извлекает информацию из базы данных и отправляет клиенту подготовленный файл HTML. Страница становится интерактивной после выполнения кода JavaScript. Сервер же отвечает только за выборку и обработку данных, имея более высокую начальную скорость загрузки.

Вторым преимуществом SSR перед CSR является возможность более эффективной поисковой оптимизации (SEO). Поскольку клиенту приходит уже готовая страница с соответствующими метаданными, алгоритмы поисковой системы, такие как Google, могут легко классифицировать эти страницы, что приводит к более высокому рейтингу веб-сайта в поисковой выдаче. В случае CSR для заполнения метаданных необходимо выполнить код JavaScript, что может занять более 300-400 мс. Это приводит к задержке и обработке пустой страницы алгоритмом, что негативно сказывается на SEO-показателях. Однако, использование только одного из методов рендеринга страниц не является оптимальным решением для всех приложений. Более эффективным подходом может быть использование гибридного приложения, которое объединяет преимущества обоих методов для оптимизации доступности и взаимодействия с пользователем. Для создания такого гибрида

можно использовать Next.js, что предоставляет возможность создания высокопроизводительных приложений с оптимальной SEO-оптимизацией.

Автор показал, как осуществить автоматическую настройку проекта, а затем – ручную настройку. В результате получилось приложение, которое обладает следующими свойствами:

- 1) автоматическая компиляция с webpack и babel,
- 2) быстрое реагирование на обновления,
- 3) статическая генерация и рендеринг на стороне сервера,
- 4) раздача статических файлов.

Затем автор даёт следующие рекомендации перед развертыванием приложения:

- 1) использование кеширования (по возможности),
- 2) отложенная загрузка пакетов JavaScript большого объема до тех пор, пока не будут нужны,
- 3) настройка логирования,
- 4) настройка обработки ошибок,
- 5) настройка страницы с ошибками 404 (Not found – не найдено) и 500 (ошибка сервера),
- 6) настройка измерения производительности,
- 7) проверка в Lighthouse производительности, передового опыта, доступности и SEO,
- 8) ознакомление с поддерживаемыми браузерами.

Для повышения производительности автор предлагает использовать:

- next/image и автоматическую оптимизацию изображений,
- автоматическую оптимизацию шрифтов,
- оптимизацию скриптов.

При обсуждении фреймворка Next.js, автор отмечает, что его можно внедрять постепенно, продолжая использовать уже существующий код и добавляя столько элементов React, сколько необходимо. Для перехода на Next.js были выделены две стратегии. Первая стратегия заключается в том,

чтобы настроить сервер или прокси таким образом, чтобы все в определенном подпути указывало на приложение Next.js. Вторая стратегия предполагает создание нового приложения Next.js, указывающего на корневой URL-адрес необходимого домена. Для проксирования некоторых подпутей в уже существующее приложение можно использовать `rewrites` внутри `next.config.js`. Такой подход позволяет постепенно переходить на Next.js, не нарушая уже существующую функциональность и обеспечивая плавный переход на новый фреймворк.

Автор рассматривает использование Next.js для микро-фронтендов с монорепозитариями и субдоменами. Данный фреймворк позволяет использовать поддомены для постепенного внедрения новых приложений. Автор выделяет следующие преимущества микроинтерфейсов:

- меньшие, более сплоченные и удобные в обслуживании кодовые базы,
- более масштабируемые организации с разделенными автономными командами,
- возможность обновлять или даже переписывать части интерфейса более инкрементально [7].

В работе О.В. Бородина и В.А. Егунова рассматривает способ оптимизации, который заключается в многопоточности с использованием API Web-workers. Особенность WebWorkers API состоит в том, что веб-воркеры – это потоки, принадлежащие браузеру, которые можно использовать для выполнения JS-кода без блокировки цикла событий [8]. Использование воркеров наиболее подходит для следующих вариантов использования:

- 1) работа с файлами,
- 2) шифрование,
- 3) предварительная загрузка данных,
- 4) проверка правописания,
- 5) рендеринг трёхмерных сцен.

Авторы демонстрируют применение данного метода оптимизации на примере обработки изображения. Важно отметить, что клиентская среда современных веб-приложений, как правило, представлена языком JavaScript, который в свою очередь не определяет параллельную модель выполнения – код JS выполняется в одном потоке. Веб-воркеры же позволяют выполнять длительные в вычислительном плане задачи без блокировки потока пользовательского интерфейса. Существуют следующие типы веб-воркеров:

- 1) выделенные воркеры (dedicated workers),
- 2) разделяемые воркеры (shared workers),
- 3) сервис-воркеры (service workers).

Авторы отмечают, что воркеры имеют свои ограничения, т.к. внутри воркер-скрипта доступен следующий набор полей:

- 1) объект navigator,
- 2) объект location (только чтение),
- 3) XMLHttpRequest,
- 4) setTimeout()/clearTimeout() и setInterval()/clearInterval(),
- 5) кэш приложений,
- 6) импорт внешних скриптов с использованием метода importScripts(),
- 7) создание других объектов web worker.

В то же время у воркер-скрипта нет доступа к следующим элементам:

- 1) модель DOM,
- 2) объект window,
- 3) объект document,
- 4) объект parent.

Авторы приводят измерения по времени выполнения операции по обработке изображения с помощью различного количества потоков. Если рассматривать ситуацию с 500 изображениями, то было достигнуто ускорение в три раза при использовании четырех потоков (воркеров) вместе с основным потоком, по сравнению с использованием только основного потока. На основе

полученных результатов авторы выделили следующие достоинства и недостатки технологии.

Достоинства web workers:

1) Поток пользовательского интерфейса свободен и может выполнять отрисовку и служебные задачи вовремя. Это повышает отзывчивость приложения и не вынуждает пользователя покидать страницу с мыслью о том, что процесс полностью «завис».

2) Скорость выполнения вычислений увеличивается, но только для тех случаев, которые можно выполнять параллельно.

3) Несмотря на некоторые ограничения, технология имеет хорошую поддержку и удобный интерфейс взаимодействия, кроссбраузерность и кроссплатформенность, не нуждается в настройке и установке.

Недостатки web workers:

1) накладные расходы на пересылку и сериализацию данных, невозможность передачи данных воркеру по ссылке, т.к. речь идёт о разных контекстах выполнения,

2) некоторые алгоритмы могут потребовать некоторого видоизменения для учета всех тонкостей,

3) некоторые данные могут потребовать конвертацию из-за потери контекста и отсутствия доступа к некоторым нужным API [9].

Как и в работе Р.А. Торопкина, Я.В. Зиновьева, Н.С. Рассказова, М.А. Митрохина, в статье зарубежных ученых V. Gowda и S. Rangaswamy было отмечено, что неоптимальные архитектурные решения для конкретного проекта могут повлиять на эффективность работы веб-приложения [10]. В их работе, посвященной устранению ограничений ReactJS, отмечается такая проблематичная ситуация, когда при каждом клике мыши на строку таблицы, вся таблицы перерисовывалась, что приводило в итоге к низким показателям производительности. Если же таблица содержит большой объем данных, то перерисовка всей таблицы при каждом клике была ещё более непродуктивной. Слишком большие задержки в отображении контента могут повлечь за собой

как плохое впечатление пользователей от работы приложения (в следствие чего произойдет отток пользователей), так и отображение некорректных данных, если контент должен обновляться с определенной периодичностью (сайты фондовых бирж и т.д.). Авторы отмечают, что данную проблему со строками таблицы можно решить изоляцией каждой строки или столбцы таблицы в качестве независимого компонента. В результате эти компоненты будут отслеживать и реагировать на события независимо друг от друга. Таким образом, каждый раз, когда происходит событие, необходимо повторно отобразить только одну строку или столбец вместо повторного отображения всей таблицы.

Данный подход касался разделения главного компонента на множество индивидуальных, но существуют и другие способы оптимизации рендеринга страницы. Так альтернативным методом к предыдущему примеру выступает параллельное выполнение различных операций. Подобный метод уже описывался в ранее рассмотренных работах, и назывался он как многопоточность. Как и русскоязычные ученые, авторы статьи рассматривают веб-воркеры в качестве инструмента для реализации выполнения операций на множестве потоков. V. Gowda и S. Rangaswamy предлагают использовать веб-воркеры, которые были представлены компанией Google и Mozilla. Чаще всего многопоточными делают следующие операции:

- 1) кеширование,
- 2) кодирование,
- 3) отрисовка canvas,
- 4) фоновые операции ввода/вывода,
- 5) фильтрация больших данных.

В подобных ситуациях несколько веб-воркеров могут быть созданы и определенные части данных будут отнесены к конкретному воркеру. Как только работа воркеров завершится, результат будет объединен. Так же важно убедиться, что не создано слишком много воркеров, т.к. все воркеры существуют до тех пор, пока не завершится основной воркер.

Но не всегда проблема бывает именно при разработке кода проекта. В работе М.Р. Reddy и S.P. Mishra, которая посвящена анализу компонентных библиотек для React JS, рассматривается проблема, когда пользователю нужно скачать пакет большого размера, из-за чего происходит задержка в загрузке страницы [11]. Поэтому при выборе библиотек для проекта важно обращать внимание на размер пакета. Но может быть довольно-таки трудно выбрать между производительностью приложения и библиотекой, которая предоставляет необходимые функции для реализации всех требований технического задания проекта. Из-за этого такой вариант оптимизации нельзя назвать наилучшим и подходящим для каждого проекта.

Как и в других составляющих проекта (back-end, база данных и т.д.), важно правильно выбрать паттерн проектирования для клиентской части приложения или даже разработать собственный подход, если существующие паттерны не подходят для конкретного проекта. Довольно глобальным способом оптимизации является архитектурный стиль приложения. S.K. Mukhiya и H.K. Hung упоминают в своей статье работу J. Nielsen, который описывал сервисно-ориентированную архитектуру (SOA – service-oriented architecture) [12]. Данная архитектура помогает достичь масштабируемости, улучшить пользовательский отклик, сократить задержки и организовать модель программирования. Но такой архитектурный стиль требует, чтобы все стандарты веб-сервисов перенимали одни и те же технические стандарты. Такую архитектуру можно совмещать с другими методами оптимизации, что положительно скажется на производительности приложения.

По результату анализа упомянутых источников можно выделить следующие тренды:

- 1) Одним из самых популярных способов оптимизации является кеширование. Существует множество вариантов реализации данного способа: использование «сервис-воркеров», более продвинутое кеширование с помощью технологии SWR.

2) Вторым из часто упоминаемых способов оптимизации является многопоточность. В рассмотренных материалах многопоточность реализовывалась с помощью веб-воркеров.

3) Важным аспектом является архитектурный стиль проекта. Неверно подобранный паттерн может сильно сказаться на производительности приложения, необходимо понимать принципы работы SPA и MPA [13].

Анализ ведущих трендов технологического развития показал, что проблема оптимизации рендеринга компонентов в клиентских фреймворках является актуальной и востребованной, т.к. пока что существуют только общие способы по оптимизации работы приложения в целом, но не рассматриваются способы улучшения именно компонентов клиентских фреймворков, который могут быть реализованы как в виде класса, так и в виде функции (если рассматривать React JS).

Проведенный анализ зарубежных научных источников по теме исследования «Исследование оптимизации рендеринга компонентов в клиентских фреймворках» позволил выявить актуальное состояние способов оптимизации работы веб-приложений, реализованных с помощью клиентских фреймворков.

2 ИССЛЕДОВАНИЕ УСТРОЙСТВА РАБОТЫ БИБЛИОТЕКИ REACTJS

2.1 Анализ работы фреймворка ReactJS

React JS – это фреймворк, JavaScript-библиотека с открытым исходным кодом для разработки клиентских приложений. Первый выпуск данной библиотеки был сделан 29 мая 2013, а последней версией, которая имеется на данный момент, является версия 18.2.0, выпущенная 14 июня 2022 года. React JS используют для создания интерфейсов одностраничных и многостраничных приложений, т.е. данный фреймворк предназначен для:

- создания функциональных интерактивных интерфейсов веб-приложений, при работе с которыми нет необходимости специально обновлять страницу для того, чтобы увидеть актуальные данные,
- удобной разработки отдельных компонентов или полноценных страниц, React JS компоненты легко используются повторно,
- легкого масштабирования проекта, внесения новой функциональности с любым стеком технологий,
- работы с серверной частью системы (получение данных, их обработка) [14].

Данный фреймворк пользуется популярностью уже много лет, его часто применяют в коммерческих решениях. Это можно объяснить тем, что React JS имеет множество особенностей, которые делают его гибким и мощным инструментом для разработки клиентских приложений.

Первой такой особенностью является декларативность. Декларативный стиль означает, что разработчику достаточно один раз описать, как будут выглядеть результаты работы кода – элементы в разных состояниях. React JS автоматически обновляет состояние элементов в зависимости от условий.

Если говорить об устройстве веб-интерфейса в целом, то любой такой интерфейс основан на HTML-документе, CSS-стилях и подключенном JavaScript коде. Модель HTML-документа называется DOM-деревом

(Document Object Mode, объектная модель документа). Это древовидная модель, в которой в иерархическом виде собраны все используемые на странице элементы. В данном случае особенностью React JS является то, что он создает и хранит в кэше виртуальное DOM-дерево – это копия обычного DOM-дерева, которая изменяется быстрее, чем реальная структура. Такой подход необходим для того, чтобы быстро обновлять состояние страницы. Если пользователь выполнит определенное действие через интерфейс приложения или произойдет какое-либо событие элементов интерфейса, DOM-дерево должно измениться. Но реальная объектная модель может быть огромной, что может привести к медленному ее обновлению. Из-за этого React JS работает не с ней, а с виртуальной копией в кэше, которая занимает гораздо меньше места в памяти. Когда происходит событие, из-за которого состояние объекта должно обновиться, и данное изменение быстро отображается в виртуальном DOM-дереве. После этого обновляется реальное DOM-дерево. С точки зрения пользователя изменения на странице будут отображаться быстро, а не после долгой загрузки.

Для обеспечения быстродействия React JS обновляет DOM по частям. В памяти хранятся две копии: актуальная и предыдущая. Если происходят какие-то обновления, то фреймворк сравнивает эти две копии между собой и изменяет только ту часть DOM-дерева, которая действительно поменялась. Такая схема работы необходима для того, чтобы не перезагружать DOM-дерево полностью, что может значительно замедлить работы страницы [15].

В старых версиях библиотеки управлять состояниями данных или компонента можно было с помощью классов. Сейчас в React JS существуют хуки – функции, которые могут реагировать на состояние элемента. Помимо встроенных в React JS хуков можно создавать собственные. Хук эффекта `useEffect` работает по тому же принципу, что и методы `componentDidMount`, `componentDidUpdate` и `componentWillUnmount` в классовых компонентах ReactJS. Пример использования хука `useEffect` представлен на рисунок 1.

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

function Example() {
  const [count, setCount] = useState(0);

  // По принципу componentDidMount и componentDidUpdate:
  useEffect(() => {
    // Обновляем заголовок документа, используя API браузера
    document.title = `Вы нажали ${count} раз`;
  });

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      <button onClick={() => setCount(count + 1)}>
        Нажми на меня
      </button>
    </div>
  );
}
```

Рисунок 1 – Пример использования хука useEffect

Когда вызывается `useEffect`, React сначала отправляет изменения в DOM, а затем запускает функцию с «эффектом». Благодаря тому, что эффекты объявляются внутри компонента, они могут иметь доступ к его пропсам и состоянию. По умолчанию, React запускает эффекты после каждого рендера, включая первый. Если нужно, из эффекта можно вернуть функцию, которая указывает, как выполнить за собой «сброс». Хуки позволяют организовать побочные эффекты в компоненте по связанным частям вместо того, чтобы разделять все на методы жизненного цикла. [16].

В плане применения самих хуков следует придерживаться следующих двух правил:

- 1) Вызывать хуки только на главном уровне. Не следует вызывать хуки внутри циклов, условий или вложенных функций.
- 2) Использовать хуки только в функциональных компонентах React JS. Не рекомендуется вызывать хуки в обычных JavaScript-функциях. Однако

есть одно исключение — это пользовательские хуки, которые можно использовать в любых функциях [17].

Если речь идет о создании пользовательских хуков, то такая потребность может возникнуть в случае, когда одна и та же логика состояния должна повторно использоваться в разных компонентах. Для решения этой задачи традиционно используется два подхода: компоненты высшего порядка и рендер-пропсы. Однако, при помощи пользовательских хуков можно решить эту задачу, не добавляя лишних компонентов. Пример хука по реализации процесса подписки на статус друга в сети показан на рисунке 2.

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID,
      handleStatusChange);
    };
  });

  return isOnline;
}
```

Рисунок 2 – Пример пользовательского хука useFriendStatus

Данный хук принимает в качестве аргумента friendID и возвращает переменную, указывающую, находится ли друг в сети или нет. На рисунке 3 продемонстрировано применение данного хука в двух разных компонентах.

Состояния каждого компонента не связаны между собой. Хуки позволяют повторно использовать логику состояния, а не само состояние. Каждый вызов хука создает абсолютно независимое состояние. Даже если

использовать один и тот же хук несколько раз в одном компоненте, это не повлияет на состояние других компонентов.

Создание пользовательских хуков – это скорее соглашение, чем дополнение. Если функция начинается с префикса "use" и использует другие хуки, то она считается пользовательским хуком[18].

```
function FriendStatus(props) {  
  const isOnline = useFriendStatus(props.friend.id);  
  
  if (isOnline === null) {  
    return 'Загрузка...';  
  }  
  return isOnline ? 'В сети' : 'Не в сети';  
}  
  
function FriendListItem(props) {  
  const isOnline = useFriendStatus(props.friend.id);  
  
  return (  
    <li style={{ color: isOnline ? 'green' : 'black' }}>  
      {props.friend.name}  
    </li>  
  );  
}
```

Рисунок 3 – Пример применения пользовательского хука в нескольких компонентах

Помимо хука `useEffect`, наиболее часто используется такой хук как `useState`. Вызов `useState` объявляет «переменную состояния» и возвращает пару значений: текущее состояние и функцию, обновляющую состояние. `useState` – это новый способ использовать те же возможности, что даёт `this.state` в классах. Обычно переменные «исчезают» при выходе из функции. К переменным состояния это не относится, потому что их сохраняет React. Единственный аргумент `useState` – это исходное состояние. В отличие от случая с классами, состояние может быть и не объектом, а строкой или числом [19]. Пример применения данного хука приведен на рисунке 4.

```

1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>Вы кликнули {count} раз(a)</p>
9:       <button onClick={() => setCount(count + 1)}>
10:        Нажми на меня
11:       </button>
12:     </div>
13:   );
14: }

```

Рисунок 4 – Пример применения хука useState

Чтобы передать какие-то данные в компонент, можно использовать свойства компонента (props). Но есть и альтернативный способ – хук useContext. Контекст позволяет передавать данные от родительского компонента к дочернему, минуя промежуточные. Хук useContext является важным инструментом библиотеки React и предназначен для получения текущего значения контекста, связанного с определенным объектом контекста. Этот объект получается в результате вызова метода React.createContext(). При вызове хука useContext происходит анализ свойства "value" в ближайшем компоненте <MyContext.Provider>, находящемся выше вызывающего компонента в дереве. Именно это свойство определяет текущее значение контекста, связанного с объектом контекста.

Предположим, что ближайший компонент <MyContext.Provider>, находящийся над вызывающим компонентом в дереве, обновляется. В этом случае, хук useContext вызовет повторный рендер компонента с последним значением контекста, связанным с данным провайдером MyContext. Важно отметить, что даже если родительский компонент использует React.memo или реализует shouldComponentUpdate, будет выполнен повторный рендер начиная с компонента, который использует useContext. Таким образом, компонент, вызывающий useContext, всегда будет перерендериваться при

изменении значения контекста. Если повторный рендер компонента является затратным, то можно использовать мемоизацию для его оптимизации [20]. Пример применения хука useContext представлен на рисунке 5.

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color:
theme.foreground }}>
      Я стилизован темой из контекста!
    </button>
  );
}
```

Рисунок 5 – Пример применения хука useContext

2.2 Анализ паттернов проектирования ReactJS приложений

Перед началом разработки приложения многие задумываются о качестве кода. Особенно это важно, если кодом будут пользоваться другие разработчики. В этом случае особенно важным становится вопрос контроля и расширяемости.

В таком случае компонент React должен:

- предоставлять простой и понятный API,
- иметь несколько модификаций и вариантов использования, легко настраивается при необходимости,
- позволять расширять и контролировать свое поведение.

Для создания такого компонента сообщество React разработало несколько паттернов. Все они так или иначе позволяют разработчику вмешиваться в работу компонента и настраивать или модифицировать его под свои нужды.

Один из таких паттернов является паттерн «Составные компоненты». Этот шаблон разработки позволяет создавать понятные декларативные компоненты без многоуровневой передачи свойств компонента. Его основное достоинство – разделение ответственности между несколькими элементами. Составные компоненты проще настраивать, и API у них максимально простой [21].

Больше нет необходимости передавать все параметры в один большой родительский компонент и затем пробрасывать их до дочерних элементов интерфейса. Теперь каждое свойство сразу прикрепляется к своему подкомпоненту – это выглядит проще и логичнее. На рисунке 6 продемонстрирована разница между обычной реализацией и паттерном составного компонента.

Так как все элементы пользовательского интерфейса вынесены в отдельные подкомпоненты, разработчик может их перегруппировать или даже убрать по своему усмотрению. Таким образом реализуется модифицируемость компонента.



Рисунок 6 – Преобразования обычного компонента к паттерну составного компонента

Основная логика содержится в базовом компоненте счетчика, а затем используется `React.Context` для совместного использования состояния и обработки событий в дочерних элементах. В итоге получается четкое разделение ответственности внутри компонента.

Следующим паттерном является шаблон «Управление свойствами». Этот шаблон предназначен для создания управляемых компонентов (<https://reactjs.org/docs/forms.html#controlled-components>). При этом внешнее состояние используется как “единственный источник истины”, и пользователь может изменять дефолтное поведение компонента, добавляя собственную логику. Пользователь полностью контролирует состояние компонента и напрямую влияет на его поведение [22]. Реализация паттерна продемонстрирована на рисунке 7.



Рисунок 7 – Пример реализации паттерна управления свойствами

Существует такой вид архитектуры приложений на React JS как Flux. Если вспомнить про паттерн MVC (model-view-controller, модель-представление-контроллер), то сам фреймворк React JS в данном паттерне играет только представления (view слой). Но при этом Flux аналогичен модели

в паттерне MVC. Это обусловлено тем, что проекты по этому шаблону будут отвечать за данные в приложение, т.е. по такой архитектуре можно сделать серверное приложение. Особенность паттерна Flux заключается в однонаправленной передаче данных. Такая концепция упрощает поиск ошибок, к примеру. Также Flux дополняет View составляющие в React.

Flux имеет 4 главных компонента:

- диспетчер (Dispatcher),
- хранилище (Stores),
- представления (Views) (React компонент),
- действие (Action).

В отличие от React, работа которого основа на использовании виртуального DOM, во Flux происходят операции, которые провоцируются действиями пользователя. В результате этих операций изменяются данные приложения. Это помогает сделать код предсказуемым, в отличие от MVC паттерна. [23].

Через диспетчера (Dispatcher) представление (View) вызывает действие (Action). Стоит отметить, что это отправится в различные хранилища. Сами хранилища (Stores) представляют именно бизнес-логику приложения. Данные хранилища изменяют зависимые представления [24].

Это доказывает, что шаблон проектирования Flux следует за однонаправленным потоком данных. Action, Dispatcher, Store и View – независимые узлы с конкретными входными и выходными данными. Данные проходят через Dispatcher, центральный хаб, который в свою очередь управляет всеми данными.

2.3 Выделение ключевых особенностей в работе ReactJS фреймворка

На основе описанного анализа было выявлено, что ключевым моментом в устройстве работы фреймворка React JS, который влияет на производительность приложения, является обновление виртуального DOM-

дерева, а именно – тот элемент интерфейса, состояние которого изменилось из-за действий пользователя или обновления данных. Из этого можно сделать вывод, что при разработке приложения на React JS следует выделять в отдельный компонент какие-либо части интерфейса исходя из их работы с данными. К примеру, в виде отдельного компонента может быть представлена форма для заполнения информации, формирование списка объектов (например, список пользователей), где представление для каждого объекта будет тоже реализовано в виде отдельного компонента.

Так же важно обращать внимание на те особенности фреймворка React JS, которые приводят к ререндерингу страницы: свойства компонента, использование хуков. Чтобы избежать излишней отрисовки компонента, можно использовать мемоизацию. Помимо этого, стоит обращать внимание на то, как данные передаются из родительского компонента в дочерние компоненты. Если данные не нужны в промежуточных дочерних компонентах, то можно воспользоваться хуком `useContext`.

Проведенный анализ позволил определить, что в работе фреймворка React JS играет ключевую роль в производительности приложения, времени отрисовки страницы приложения.

3 ИССЛЕДОВАНИЕ СПОСОБОВ ОПТИМИЗАЦИИ РЕНДЕРИНГА ПРИЛОЖЕНИЯ, РЕАЛИЗОВАННОГО С ПОМОЩЬЮ REACTJS

3.1 Исследование инструментов измерения производительности страницы веб-приложения

Для оценки производительности клиентского приложения чаще всего используются средства, которые уже встроены во многих браузерах. Таким инструментом является «DevTools» (development tools) – это программа, позволяющая создавать, тестировать, отлаживать программное обеспечение. В данном инструменте можно посмотреть исходный код сайта, отладить HTML верстку сайта, CSS стили, скрипты на JavaScript. Можно проверить сетевой трафик, который потребляется сайтом, контролировать скорости сети (к примеру, установить быстрый 3G, медленный 3G, режим офлайн или сделать собственную настройку с необходимыми параметрами), оценить быстродействие и многое другое.

В данном инструменте имеется вкладка «Performance», в которой представлен интерфейс для оценки производительности сайта. Здесь отображается нагрузка, которую создаёт приложение на компьютере пользователя. Панель разработчика браузера предоставляет различные инструменты для анализа производительности страницы, такие как показатели FPS (frames per second, кадровая частота), загрузка CPU и сетевые запросы, а также необходимые данные и инструменты для повышения производительности. На этой панели также имеется таймлайн использования сети, выполнения JavaScript и загрузки памяти. После первого построения таймлайнов, можно получить данные о всем жизненном цикле страницы и выполнении кода. Кроме того, можно оценить время исполнения отдельных частей кода и выбрать конкретный период на шкале, чтобы увидеть, какие процессы происходили в выбранный интервал. Все эти возможности позволяют провести анализ каждого события, которое происходило в момент загрузки страницы или во время взаимодействия пользователя с ней.

Ниже таймлайна можно посмотреть суммарную информацию о загрузке, выполнению скриптов JavaScript, рендерингу, отрисовке и другим процессам. Всё это представлено в виде кольцевой диаграммы, справа от которой находится сводное описание, сколько времени в миллисекундах занял тот или иной процесс.

Помимо вкладки «Performance» в DevTools присутствует такой инструмент как Lighthouse. Он не только тестирует сайт и показывает оценку производительности, но и даёт конкретные рекомендации: что можно улучшить, чтобы сделать загрузку сайта быстрее. Для запуска проверки достаточно выбрать необходимые параметры и нажать кнопку «Generate report». Lighthouse анализирует четыре показателя:

- 1) производительность,
- 2) доступность,
- 3) SEO,
- 4) Best Practices (лучшие практики).

Касательно производительности, данный инструмент анализирует скорость загрузки сайта. На данную оценку влияют такие факторы как время блокировки, отрисовка стилей, загрузка интерактивных элементов, шрифтов и контента. Так же стоит отметить, что для прогрессивных веб-приложений добавляется пятый показатель – PWA (progressive web app). По данному показателю проверяется, регистрирует ли сайт Service Workers (посредники между клиентом и сервером, пропускающий через себя все запросы к серверу), работает ли офлайн и возвращает ли код 200.

Для показателя доступности проверяется, могут ли все пользователи получать доступ к контенту и эффективно перемещаться по сайту. Эта оценка зависит от понятности и воспринимаемости контента, возможности управлять интерфейсом и передвигаться по содержимому через Tab-навигации.

Параметр SEO оценивает соответствие страницы советам Google по поисковой оптимизации. Проверяется использование метатегов, доступ к

индексации, наличие атрибутов alt у изображений, адаптированность к мобильным экранам и другие характеристики.

Последним параметром является Best Practices, для которого проверяется безопасность сайта и использование современных стандартов веб-разработки. Оценка зависит от того, используется ли на сайте HTTPS, устаревшие API, правильная кодировка и другие параметры [25].

При оценке производительности приложения важно учитывать, насколько будут различаться показатели на локальном устройстве разработчика и уже на сервере для развертывания приложения. DevTools удобен тем, что разработчик может проверить приложение как на своем локальном хосте, так и развернутый на сервере сайт. Хотя и DevTools является самым популярным инструментом для разработчиков, существует множество сервисов, которые позволяют оценить производительность сайта. Одним из таких стоит отметить Goggle PageSpeed Insights. Данный инструмент, предоставляемый Google, является бесплатным и позволяет произвести анализ скорости загрузки веб-страницы на различных устройствах, включая мобильные и настольные компьютеры. Кроме того, он предоставляет пользователю рекомендации по ускорению загрузки страницы.

PageSpeed Insights оценивает скорость загрузки на основании следующих данных:

- 1) Собственные данные, полученные инструментом в ходе имитации загрузки веб-страницы, могут предоставить оценку реальной скорости загрузки, а также обнаружить возможные проблемы, которые могут негативно влиять на этот процесс. Вместе с тем, следует учесть, что подобные эксперименты проводятся в управляемых условиях, что может ограничивать возможность учесть все факторы, которые могут возникнуть в процессе реальной загрузки страницы пользователем.

- 2) Скорость загрузки страницы у реальных пользователей. Отчет об удобстве пользования браузером Chrome позволяет объективно оценить скорость загрузки страницы. Но доступ к набору данных ограничен,

эксперимент с имитацией страницы позволяет получить больше данных о скорости веб-страницы.

Данный инструмент предназначен для оценки скорости загрузки веб-страницы на основе 100-бальной шкалы. Оценка формируется на основании наблюдений за последние 28 дней, где все загрузки веб-страницы принимаются за 100%. После этого сервис проводит анализ процента быстрых, средних и низких загрузок страницы, что позволяет получить итоговую оценку. Таким образом, данный инструмент является эффективным способом контроля и улучшения скорости загрузки веб-страницы.

В отчете оцениваются два показателя: первая отрисовка контента (FCP) и первая задержка ввода (FID). Критерии оценки представляются следующими:

- Страница загружается медленно. FCP – более 2500 мс, FID – более 250 мс.
- Средняя скорость загрузки. FCP – от 1000 до 2500 мс, FID – от 100 до 250 мс.
- Низкая скорость загрузки. FCP – от 0 до 1000 мс, FID – от 0 до 50 мс.

Основываясь на данном анализе и других параметрах, инструмент дает общую оценку скорости работы сайта, шкала представлена следующим образом:

- 0-49 баллов. Если сайт получает такое количество баллов, то попадает в красную зону – страница загружается медленно.
- 50-89 баллов. Оранжевая зона – средняя скорость загрузки страницы.
- 90–100 баллов. Зеленая зона – высокая скорость загрузки страницы [26].

В рамках данной работы планируется использовать DevTools и брать данные преимущественно из вкладки Performance.

3.2 Разработка клиент-серверного приложения с помощью ReactJS и Flask для проведения последующего сравнения методов оптимизации

Для проведения сравнения способов ускорения рендеринга приложения было разработано клиент-серверное приложение, в котором будет некоторая нагрузка страницы контентом (изображения, текст). В качестве предметной области для такого приложения был сделан проект, который представляет собой визитную карточку издательского дома Bubble Comics. Информационная система будет использоваться для презентации продуктов компании, доступ к её контактным данным. Информационная система предполагает наличие базы данных, которая должна содержать ссылки на видео и путь к заставке для видео.

3.2.1 Постановка задачи по разработке клиент-серверного приложения

Данный проект направлен на решение задач конкретно бизнеса – пиар-сайт издательского дома. С точки зрения потребителя, разрабатываемая система должна предоставлять удобный интерфейс для ознакомления с проектами и продуктами издательского дома, контакты компании, ссылка на интернет-магазин компании. С точки зрения представителей издательского дома, информационная система должна позволять добавлять новый контент на страницу через специальную форму интерфейса.

В результате анализа предметной области были выявлены, что для функционирования пока что необходима одна сущность, которая будет называться Slide. Она представлена на рисунке 8. Данная сущность представляет собой информацию о видеоролике, который расположен на главной странице приложения. Slide будет содержать такие атрибуты как идентификатор видео, URL ролика, файловый путь к изображению обложки.

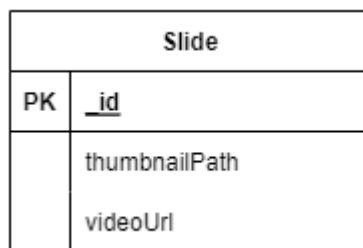


Рисунок 8 – ER-диаграмма

На рисунке 9 приведена структура разрабатываемой базы данных в формате JSON.

```

1  Slide
2  {
3      "_id": string,
4      "thumbnailPath": string,
5      "videoUrl": string,
6  }
```

Рисунок 9 – Пример записи Slide

В качестве базы данных было принято решение использовать документо-ориентированную MongoDB исходя из следующих соображений:

1) Отсутствие схемы. Данная система управления базами данных (СУБД) предназначена для работы с коллекциями разнообразных документов, которые могут отличаться по количеству полей, содержанию и размеру. При этом, каждый документ может иметь уникальную структуру, что является ключевой особенностью данной СУБД. Таким образом, данная система позволяет эффективно управлять различными сущностями, не требуя их идентичности по структуре. Это обеспечивает гибкость и удобство при работе с данными, а также повышает эффективность их обработки.

2) Структура каждого объекта представлена в крайне понятном формате, что является важным преимуществом.

3) Легко масштабируется.

4) Для хранения в данный момент используемых данных используется внутренняя память, что позволяет получать более быстрый доступ.

5) Данные хранятся в виде JSON документов.

6) MongoDB поддерживает динамические запросы документов (document-based query).

7) Отсутствие сложных JOIN запросов.

8) Нет необходимости маппинга объектов приложения в объекты базы данных.

Помимо этого, были созданы диаграммы вариантов использования и последовательности, представленные на рисунках 10 и 11.

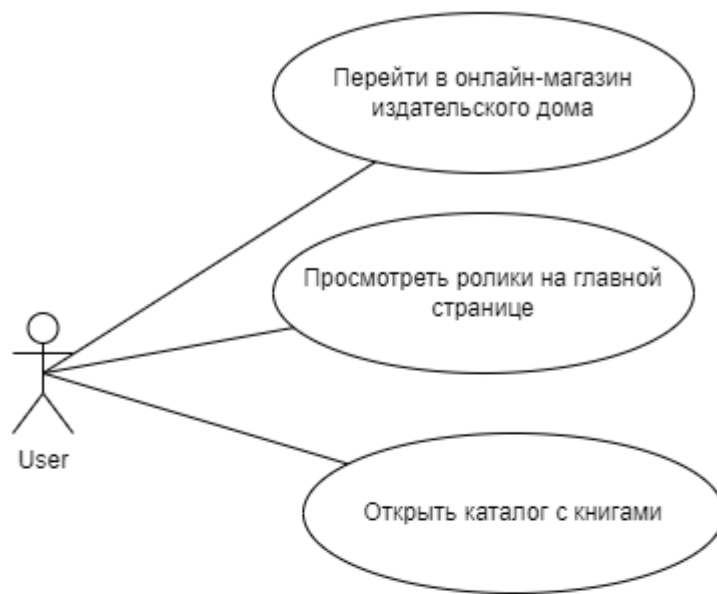


Рисунок 10 – Use-Case диаграмма системы

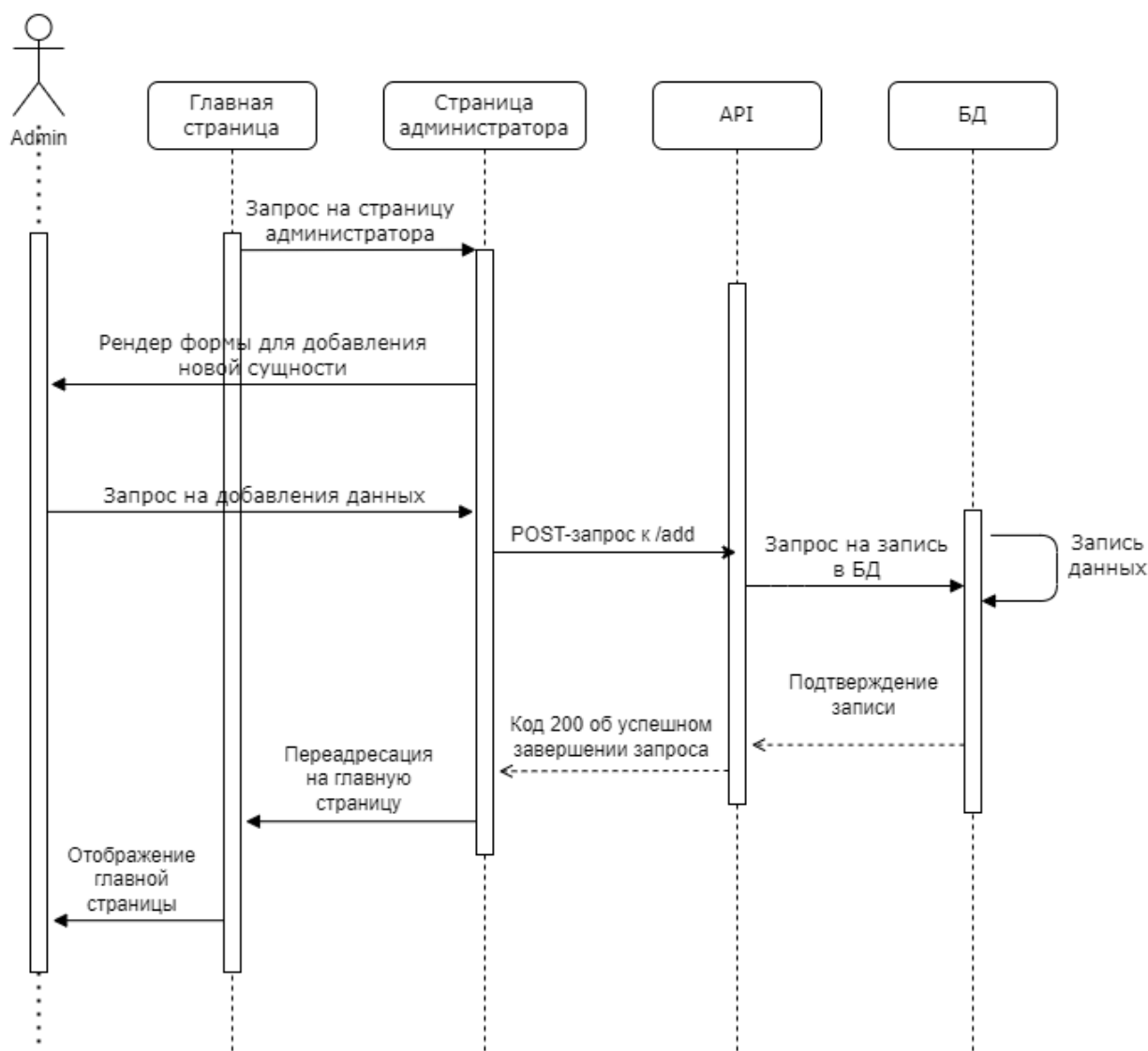


Рисунок 11 – Диаграмма последовательности для добавления нового ролика администратором

Для контроля версий и изменений в проекте была использована система контроля версий Git. Для хранения исходного кода и использования сервиса CI/CD можно использовать GitLab.

3.2.2 Проектирование макета

Для проектирования прототипа было решено использовать такой инструмент как Figma. Данное приложение можно использовать как на локальных машинах, так и в браузере. Веб-версия позволяет организовать совместную работу. Простота интерфейса не отнимает много времени для освоения.

Таким образом был разработан прототип главной страницы сайта издательского дома. Макет главной страницы представлен на рисунке 12 и дополнительная страница для администратора представлена на рисунке 13.



Рисунок 12 – Главная страница

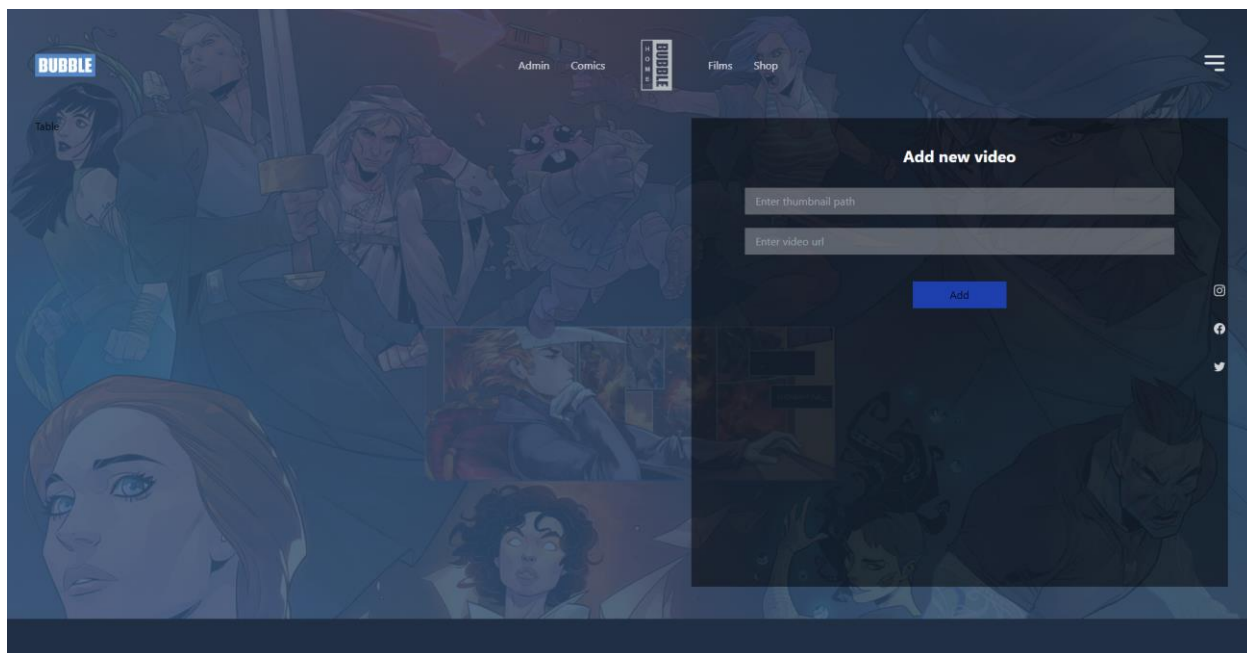


Рисунок 13 – Страница администратора для добавления нового видео

Для верстки спроектированных прототипов предполагается использование такого фреймворка как Tailwindcss. Данное решение обосновывается тем, что front-end приложение данной системы будет

разрабатывается при помощи библиотеки React JS, с которой Tailwindcss хорошо интегрируется.

3.2.3 Проектирование клиентского приложения

Клиентская сторона системы была разработана с помощью библиотеки ReactJS и фреймворка Tailwindcss. Архитектура проекта представлена в виде компонентов и собственных хуков для реализации определенных сценариев приложения. Сами компоненты написаны на JSX – расширения языка JavaScript.

React исходит из принципа тесной связи между логикой рендеринга и другой логикой пользовательского интерфейса, такой как обработка событий, изменение состояния во времени и подготовка данных для отображения. В отличие от традиционного подхода, когда разметка и логика разделяются и помещаются в разные файлы, React использует слабо связанные компоненты, которые объединяют в себе как разметку, так и логику. Это позволяет более эффективно управлять компонентами, а также повышает гибкость и удобство при работе с данными. Таким образом, React является технологией, которая позволяет улучшить процесс разработки пользовательского интерфейса и повысить эффективность работы с данными.

Каждое выражение JSX после компиляции превращается в вызов функции JavaScript, результатом которой является объект JavaScript. Это означает, что JSX может быть использован внутри конструкций if и циклов for, а также присваиваться переменным, передаваться в функции в качестве аргументов и возвращаться из функций. Такой подход обеспечивает максимальную гибкость и удобство при работе с JSX, а также позволяет использовать его в различных контекстах и с различными функциями. Таким образом, JSX является мощным инструментом для разработки пользовательского интерфейса, который позволяет создавать более эффективный и гибкий код.

По умолчанию, React DOM обеспечивает защиту от межсайтового скриптинга (XSS), экранируя все значения, включенные в JSX, перед их

рендерингом. При этом, все значения преобразуются в строки, что гарантирует, что никакие внешние данные не будут внедрены в приложение без явного указания. Такой подход обеспечивает максимальную безопасность при работе с данными и предотвращает возможные атаки XSS.

На главной странице сделаны элементы с навигацией по разделам приложения, фоновое изображение с применением эффекта градиента, лента видеоряда с анимацией прокрутки, два изображения на переднем плане и текст. Для воспроизведения выбранного видео было прописано открытие модального окна с плеером.

Получившаяся структура проекта представлена на рисунке 14.

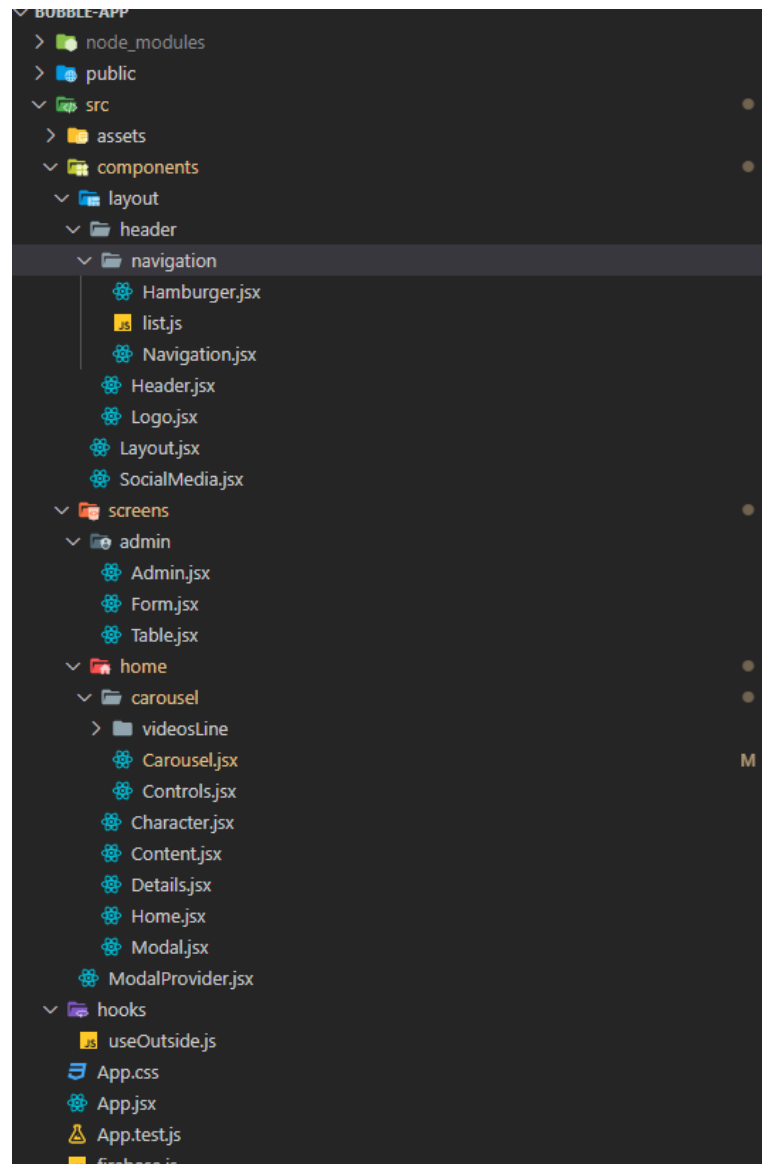


Рисунок 14 – Структура клиентского приложения на React JS

3.2.4 Проектирование серверного приложения

Для реализации API системы был выбран фреймворк Flask. Этот веб-фреймворк считается лучшим для создания небольших статических сайтов и легковесных веб-приложений, он имеет интуитивно понятный синтаксис и простую структуру, можно редактировать большую часть инструментов под задачи. Также есть инструменты для отладки и тестирования – unit-тесты, встроенный сервер разработки, обработчик запросов. С новой версии 2.0 поддерживается асинхронность.

Как уже говорилось ранее, в качестве СУБД была выбрана MongoDB. Интеграция Python с MongoDB происходит через такой драйвер как PyMongo. Это официальный драйвер MongoDB для Python, который можно использовать для выполнения всех видов запросов над данными, хранящиеся в базе данных.

На рисунке 15 представлен файл app.py с описанием методов API.

```
bubleapp-backend > app > app.py
1  import os
2  from datetime import datetime
3  from flask import Flask, request, jsonify, make_response
4  from flask_cors import CORS, cross_origin
5  from flask_pymongo import PyMongo
6
7  application = Flask(__name__)
8  cors = CORS(application)
9  application.config["MONGO_URI"] = 'mongodb://' + os.environ['MONGODB_USERNAME'] + ':' + os.environ['MONGODB_PASSWORD'] +
10
11  mongo = PyMongo(application)
12  db = mongo.db
13
14
15  @application.route('/api')
16  @cross_origin()
17  def index():
18      _slides = db.slides.find()
19      item = {}
20      data = []
21      for slide in _slides:
22          item = {
23              '_id': str(slide['_id']),
24              'thumbnailPath': slide['thumbnailPath'],
25              'videoUrl': slide['videoUrl'],
26          }
27          data.append(item)
28      resp = make_response(jsonify(
29          status=True,
30          data=data
31      ))
32      resp.headers['Access-Control-Allow-Origin'] = '*'
33      return resp
34
35  @application.route('/add', methods=['POST'])
36  @cross_origin()
37  def add():
38      data = request.get_json(force=True)
39      item = {
40          'thumbnailPath': data['thumbnailPath'],
41          'videoUrl': data['videoUrl']
42      }
```

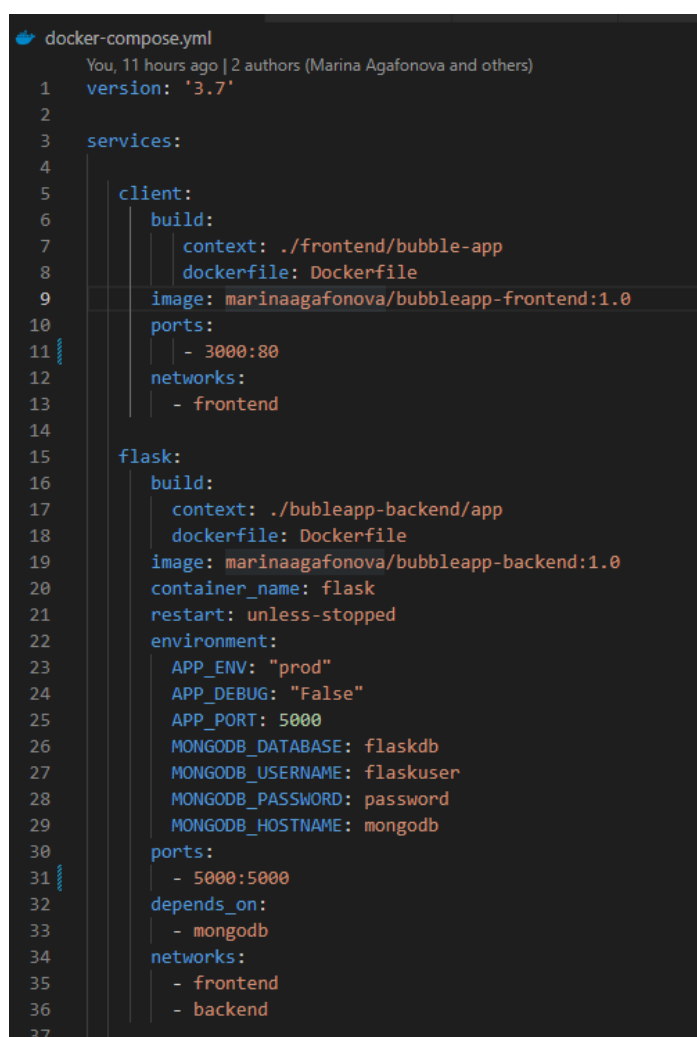
Рисунок 15 – Пример одного из методов API

3.2.5 Сборка проекта в Docker и настройка CI/CD

Для сборки проекта в Docker контейнеры были написаны Dockerfile для клиентского и серверного приложений индивидуального. Далее в docker-compose.yml файл были описаны все сервисы, которые будут задействованы в проекте:

- 1) client – клиентское приложения на React JS,
- 2) flask – серверное приложения на фреймворке Flask,
- 3) mongodb – СУБД MongoDB.

На рисунке 16 представлен фрагмент по описанию первых двух сервисов



```
docker-compose.yml
You, 11 hours ago | 2 authors (Marina Agafonova and others)
1  version: '3.7'
2
3  services:
4
5      client:
6          build:
7              context: ../frontend/bubble-app
8              dockerfile: Dockerfile
9          image: marinaagafonova/bubbleapp-frontend:1.0
10         ports:
11             - 3000:80
12         networks:
13             - frontend
14
15     flask:
16         build:
17             context: ../bubleapp-backend/app
18             dockerfile: Dockerfile
19         image: marinaagafonova/bubbleapp-backend:1.0
20         container_name: flask
21         restart: unless-stopped
22         environment:
23             APP_ENV: "prod"
24             APP_DEBUG: "False"
25             APP_PORT: 5000
26             MONGODB_DATABASE: flaskdb
27             MONGODB_USERNAME: flaskuser
28             MONGODB_PASSWORD: password
29             MONGODB_HOSTNAME: mongodb
30         ports:
31             - 5000:5000
32         depends_on:
33             - mongodb
34         networks:
35             - frontend
36             - backend
37
```

Рисунок 16 – Сервисы client и flask

Сборка и развертывание происходит с помощью Docker Compose. Это инструмент для запуска и управления мультиконтейнерными приложениями.

Он помогает создать изолированную среду, в которой содержатся все необходимые зависимости.

Как можно видеть из описания сервисов, API системы расположено на 5000 порту, а клиентская часть – на 3000. Общение этих двух сервисов происходит посредством Nginx. Nginx – это веб-сервер и почтовый прокси, который работает под управлением операционных систем семейства Linux/Unix и Microsoft. Изначально продукт разрабатывался только под Unix-системы. Первые релизы тестировались FreeBSD, Linux, Solaris, но позже разработчик добавил совместимость с платформой Windows. Nginx отличается от других аналогов в данном сегменте тем, что использует более эффективный принцип обработки входящих данных. Вместо традиционного подхода, когда каждый запрос пользователя обрабатывается в целом, Nginx разбивает каждый запрос на несколько мелких, что упрощает их обработку. В терминологии Nginx такие запросы называются рабочими соединениями. Такой подход позволяет значительно ускорить процесс обработки запросов и повысить эффективность работы с данными

После обработки каждое соединение собирается в одном виртуальном контейнере, чтобы трансформироваться в единый первоначальный запрос, а после отправляется пользователю. Одно соединение может одновременно обрабатывать до 1024 запросов конечного пользователя.

Веб-сервер использует специальный выделенный сегмент памяти, который называется «пул» (pool), для того чтобы уменьшить нагрузку на оперативную память. Этот пул является динамическим и автоматически расширяется при увеличении длины запроса. Это позволяет оптимизировать использование ресурсов и повысить производительность веб-сервера.

Веб-сервер применяется в следующих ситуациях:

- 1) Выделенный порт или IP-адрес. Если на сервере присутствует большое количество статичного материала (картинки, тексты и т. д.) либо файлов для загрузки пользователями, то Nginx используют, чтобы выделить

под данные операции отдельный IP-адрес либо порт. Таким образом нагрузка на сервер распределяется.

2) Прокси-сервер. Когда пользователь загружает страницу сайта, на которой расположен статичный контент, Nginx сначала кэширует данные у себя, а потом возвращает результат. При следующих запросах данной страницы ответ происходит в разы быстрее.

3) Распределение нагрузки. При запросе страницы сайта, пользователю выдается ответ в синхронной последовательности. Nginx использует асинхронный режим. Все запросы обрабатываются на разных этапах. Такой подход повышает скорость обработки.

4) Почтовый сервер. Поскольку в веб-сервер встроены механизмы аутентификации, то его часто используют для перенаправления на почтовые сервисы после прохождения авторизации клиентом.

В Nginx встроены механизмы защиты. Информация передается по зашифрованному каналу через протоколы SSL/TLS. Веб-сервер Nginx идеально подходит для сайтов, на которых содержится в основном статический контент. Он также способен выступить как редирект для почтовых сервисов либо в роли прокси-сервера. Простота и гибкость настройки позволяет масштабировать продукт без особых усилий.

Для данного проекта был написан следующий конфиг, представленный на рисунке 17, для перенаправления запрос с клиента на сервер. Если будет совпадение по одному из значений location, то адрес будет адресован к сервису flask с 5000 портом.

```
frontend > bubble-app > nginx.conf
You, 14 hours ago | 1 author (You)
1 upstream web-app_api {
2     server flask:5000;
3 }
4
5 server {
6     listen 80;
7
8     location /api {
9         proxy_pass http://web-app_api;
10        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
11        proxy_set_header Host $host;
12        proxy_redirect off;
13    }
14
15    location /add {
16        proxy_pass http://web-app_api;
17        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
18        proxy_set_header Host $host;
19        proxy_redirect off;
20    }
21
22    location / {
23        root /usr/share/nginx/html;
24        index index.html index.htm;
25        try_files $uri $uri/ /index.html;
26    }
27
28 }
```

Рисунок 17 – Конфиг файл nginx

Далее для реализации CI/CD на GitLab был написан файл .gitlab-ci.yml, представленный на рисунке 18.

```
🔥 .gitlab-ci.yml
  You, 13 hours ago | 1 author (You)
1  stages:
2    - docker prod
3
4  docker_prod:      You, 13 hours ago • Uncommitted changes
5    stage: docker prod
6    image: creatiwww/docker-compose:latest
7    before_script:
8      - docker login -u \${DOCKERHUB\_USER} --password \${DOCKERHUB\_PASSWORD}
9    script:
10     - docker-compose -f docker-compose-prod.yml build
11     - docker-compose -f docker-compose-prod.yml push
12    after_script:
13     - docker-compose -f docker-compose-prod.yml down --rmi all
14    cache:
15      paths:
16        - ./frontend/bubble-app/node_modules
17    rules:
18     - if: $CI_COMMIT_TAG
19    tags:
20     - docker
```

Рисунок 18 – Описание сборки в GitLab

Перед выполнением скрипта происходит авторизация на DockerHub, куда будут опубликованы два сервиса с клиентским и серверным приложениями. На стадии script происходит сначала сборка проекта, а затем публикация на DockerHub. По завершению выполнения этих двух инструкций команда `docker-compose down` останавливает все сервисы, связанные с конфигурацией Docker Compose. В отличие от команды `stop`, она также удаляет все контейнеры и внутренние сети, связанные с этими сервисами – но не указанные внутри тома. Весь этот сценарий запускается при условии, чтобы был создан commit тег в репозитории проекта.

На рисунке 19 можно увидеть результат настройки CI/CD для репозитория в GitLab. В результате на DockerHub появятся два репозитория:

marinaagafonova/bubbleapp-frontend и marinaagafonova/bubbleapp-backend.

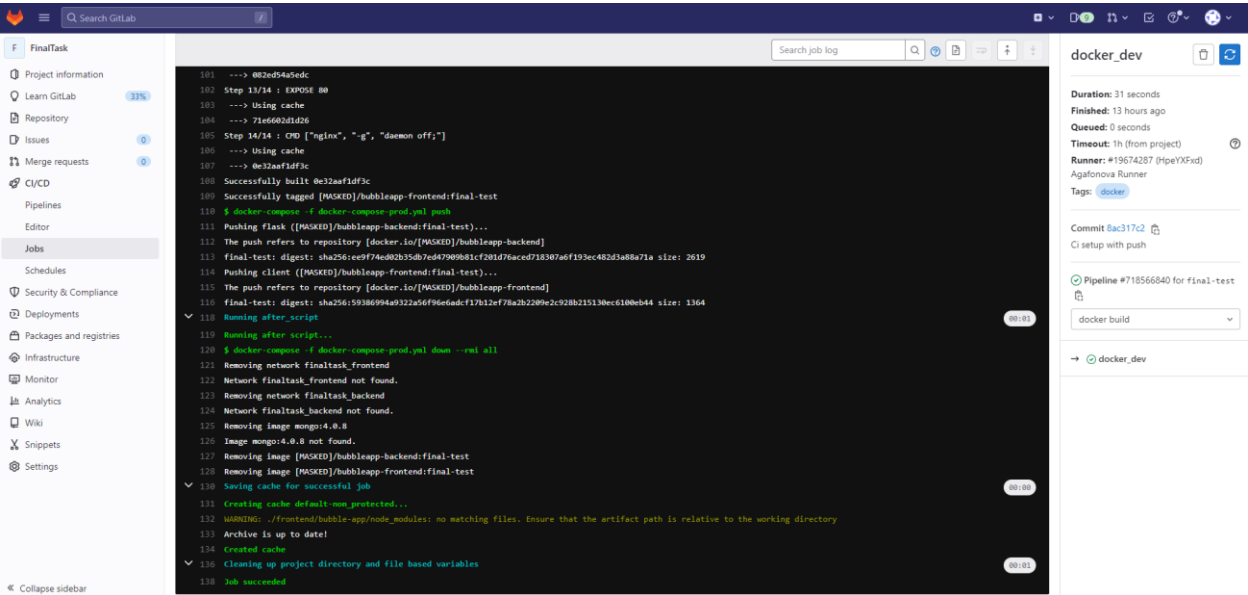


Рисунок 19 – Job по сборке и публикации проекта на DockerHub

3.3 Сравнение методов оптимизации для разработанного приложения

Был сделан замер производительности получившего приложения с помощью инструмента DevTools. Результат, сформированный во вкладке «Performance» представлен на рисунке 20. Как показано на диаграмме, время загрузки составляет 3 миллисекунд, время выполнения скриптов JavaScript – 1424 миллисекунд, а время рендеринга – 220 миллисекунд.

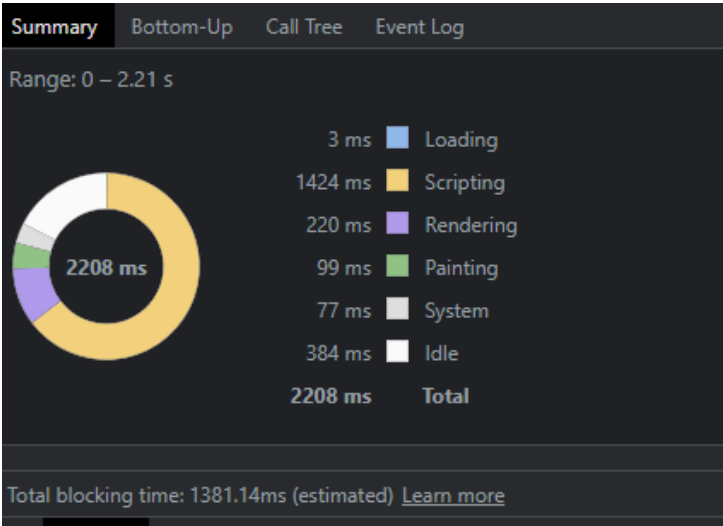


Рисунок 20 – Суммарная информация производительности приложения

Помимо этого, был составлен отчёт во вкладке «Lighthouse». Результат оценивания по пяти критериям представлен на рисунке 21.

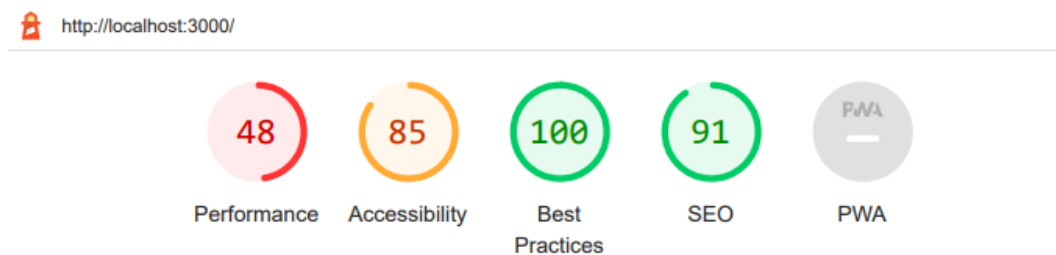


Рисунок 21 – Оценки из отчета Lighthouse

При анализе отечественных и иностранных источников было выявлено, что одним из самых популярных методов оптимизации является кеширование. После оценки получившего кода приложения было решено реализовать следующие способы кеширования для React JS приложения: React.memo и moize.

3.3.1 Кэширование – применение React.memo

React.memo – это компонент высшего порядка React, используемый для пропуска повторных рендеров. Слово «memo» относится к термину «memoization» (мемоизация). Мемоизация – это метод оптимизации, используемый для ускорения приложений путем хранения результатов дорогостоящих вызовов функций и возврата кэшированного результата вычислений при повторном использовании тех же входных данных. При оборачивании компонента с помощью React.memo приложение React будет использовать последнюю отрисованную версию этого компонента [27].

Для рассматриваемого проекта мемоизация была применена для компонента «Home», который представляет главную страницу приложения. Код компонента представлен на рисунке 22, а применение memo() находится на 20 строке кода. На рисунке также можно увидеть расчет объема применяемых библиотек (отображаются зеленым шрифтом справа от строки с импортированием пакета). Такая информация была получена с помощью расширения для редактора кода VS Code, которое называется «Import Cost» [28].

```

Home.jsx
src > components > screens > home > Home.jsx
1 import React from 'react' 6.9k (gzipped: 2.7k)
2 import Layout from '../../layout/Layout'
3 import Content from './Content'
4 import Details from './Details'
5 import Character from './Character'
6 import Carousel from './carousel/Carousel'
7 import { memo } from 'react' 4.1k (gzipped: 1.8k)
8
9 const Home = () => {
10   return (
11     <Layout>
12       <Content />
13       <Carousel />
14       <Character />
15       <Details />
16     </Layout>
17   )
18 }
19
20 export default memo(Home)
21

```

Рисунок 22 – Компонент Home с применением мемо()

Как и с первоначальной версией проекта, была проведена оценка производительности в «Performance» и «Lighthouse». Измерения в Performance представлены на рисунке 23, а в Lighthouse – на рисунке 24.

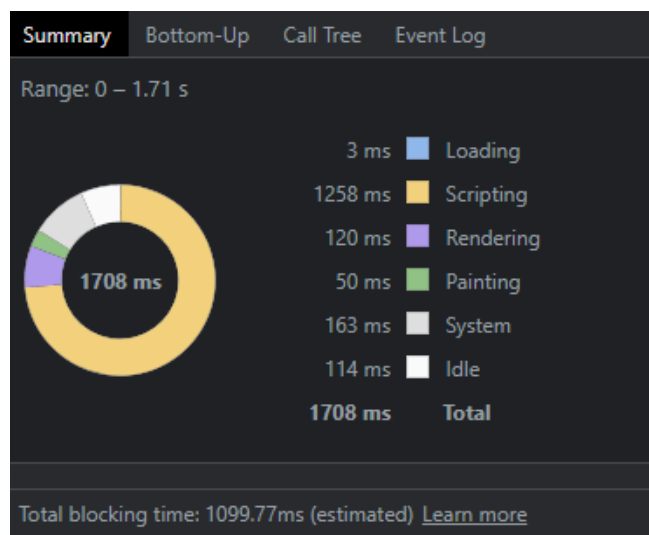


Рисунок 23 – Суммарная информация производительности приложения после применения мемо()



Рисунок 24 – Оценки из отчета Lighthouse для версии с применением мемо

3.3.2 Кэширование – применение moize

Второй способ кэширование, который можно реализовать на рассматриваемом проекте, – это moize, библиотека по мемоизации JavaScript кода. Сама функция moize принимает несколько значений без каких-либо дополнительных настроек. Эта функция выполняет поверхностное сравнение пропсов (props, входные данные React-компонента) и контекста компонента на основе строгого равенства [29]. Moize была так же применена к компоненту Home, код которого представлен на рисунке 25.

```

Home.jsx
src > components > screens > home > Home.jsx
1 import React from 'react' 6.9k (gzipped: 2.7k)
2 import Layout from '../layout/Layout'
3 import Content from './Content'
4 import Details from './Details'
5 import Character from './Character'
6 import Carousel from './carousel/Carousel'
7 import moize from 'moize' 18.7k (gzipped: 6.2k)
8
9 const Home = () => {
10   return (
11     <Layout>
12       <Content />
13       <Carousel />
14       <Character />
15       <Details />
16     </Layout>
17   )
18 }
19
20 export default moize(Home, { isReact: true })
21

```

Рисунок 25 – Компонент Home с применением moize()

Измерения в Performance представлены на рисунке 26, а в Lighthouse – на рисунке 27.

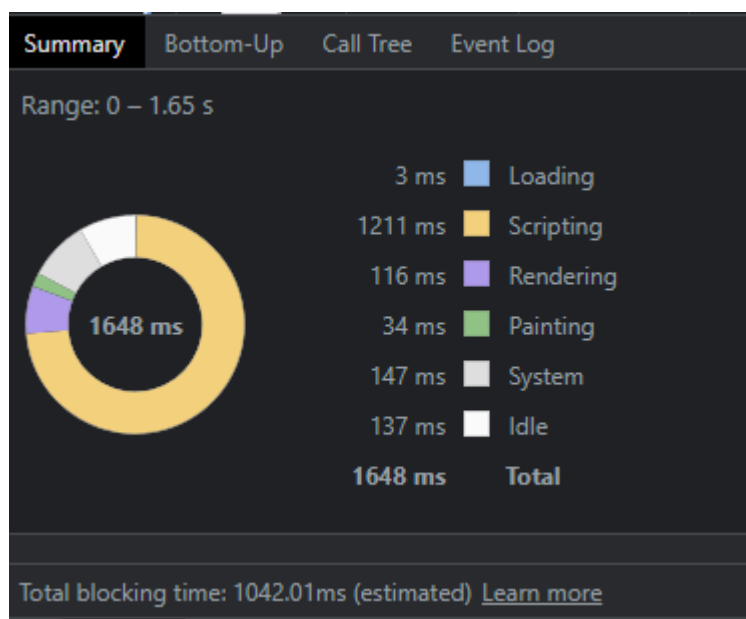


Рисунок 26 – Суммарная информация производительности приложения после применения moize()

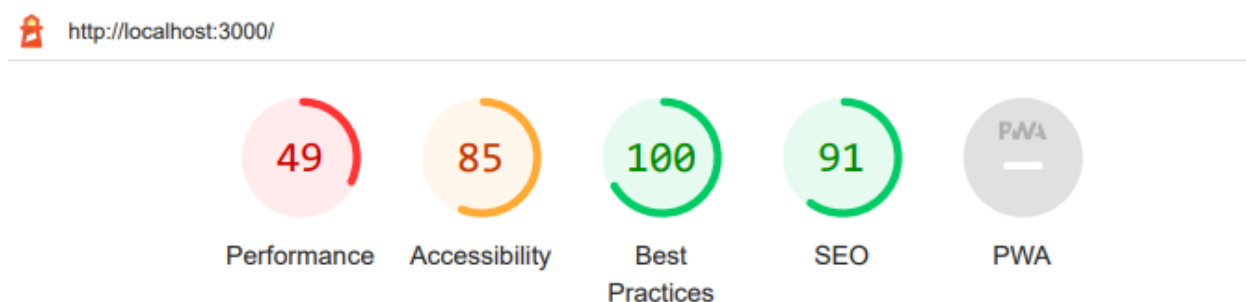


Рисунок 27 – Оценки из отчета Lighthouse для версии с применением moize

3.3.3 Lazy loading («ленивая загрузка») изображений

Lazy loading стал одним из способов оптимизации React приложений. Этот метод используется для предотвращения загрузки части контента, который не требуется в отображении при начальной загрузке страницы.

В экосистеме React существует множество пакетов, которые помогают оптимизировать приложение с помощью ленивой загрузки. Для рассматриваемого приложения была выбрана библиотека «react-lazyload». Применения данной библиотеки осуществляется путём оборачивания компонентов, по отношению к которым нужно использовать ленивую загрузку, в компонент LazyLoad. Для него можно указывать различные

атрибуты. К примеру, высоту элемента, свойство «once», которое означает, что после отображения содержимого оно не будет управлять react lazyload [30]. В разработанном проекте компонент LazyLoad был применен к нескольким изображениям. Один из примеров показан на рисунке 28, где демонстрируется компонент Character, который содержит изображение персонажа на главной странице приложения.

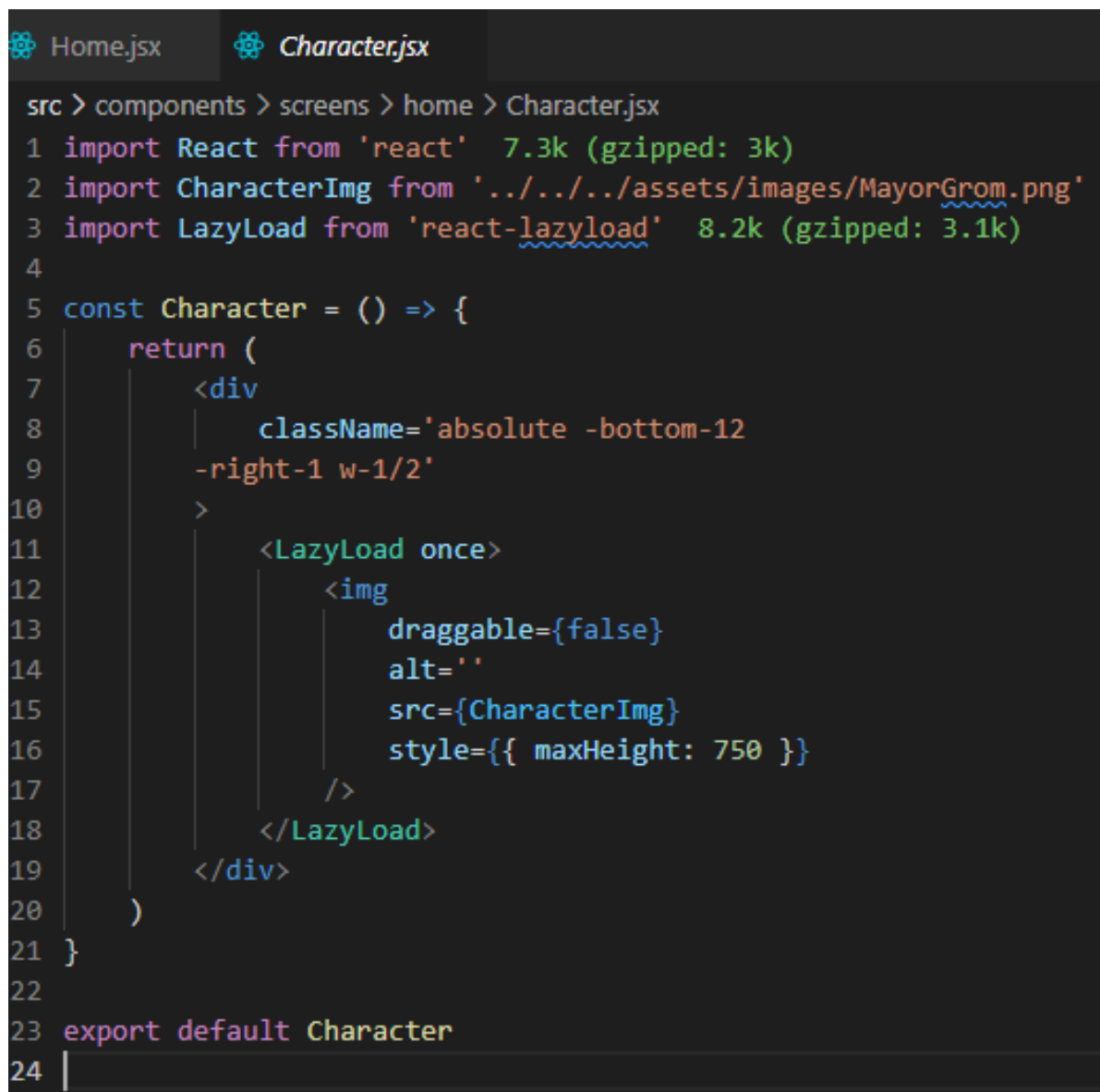
The image is a screenshot of a code editor with a dark theme. At the top, there are two tabs: 'Home.jsx' and 'Character.jsx', with 'Character.jsx' being the active tab. Below the tabs, the file path 'src > components > screens > home > Character.jsx' is displayed. The code is written in JavaScript/React and includes imports for 'React' from 'react' (7.3k gzipped: 3k), 'CharacterImg' from a local path '../assets/images/MayorGrom.png', and 'LazyLoad' from 'react-lazyload' (8.2k gzipped: 3.1k). The 'Character' component is defined as a function that returns a JSX element. This element consists of a 'div' with the class 'absolute -bottom-12 -right-1 w-1/2'. Inside the 'div' is a 'LazyLoad' component with the 'once' prop, which contains an 'img' element. The 'img' element has props 'draggable={false}', 'alt=""', 'src={CharacterImg}', and a style object with 'maxHeight: 750'. The code ends with 'export default Character'.

Рисунок 28 – Компонент Character с применением LazyLoad

Измерения в Performance представлены на рисунке 29, а в Lighthouse – на рисунке 30.

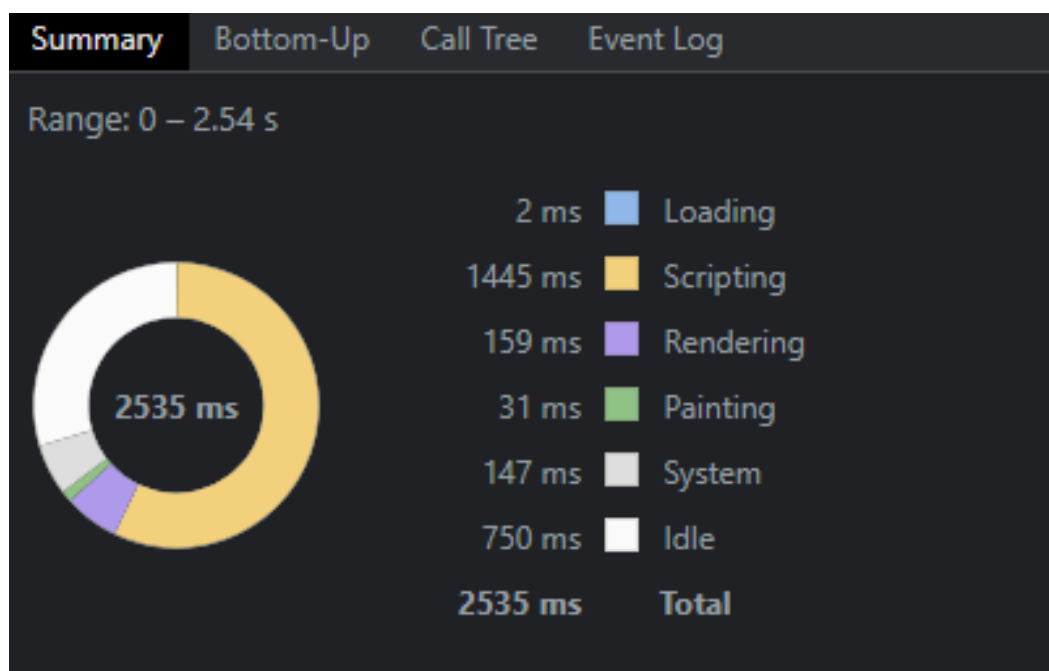


Рисунок 29 – Суммарная информация производительности приложения после применения LazyLoad

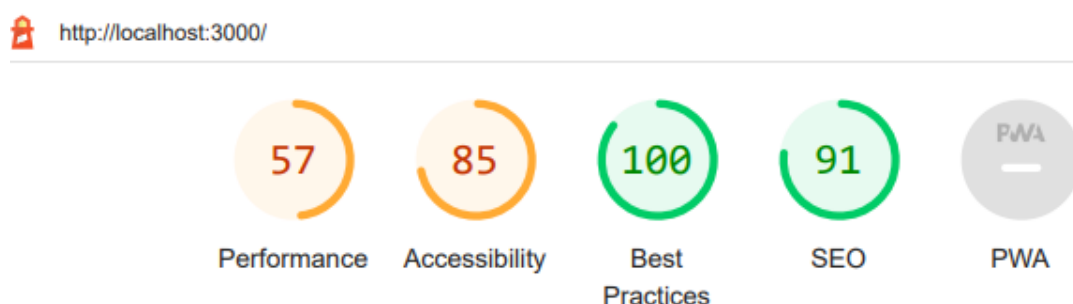


Рисунок 30 – Сценки из отчета Lighthouse для версии с применением LazyLoad

3.3.4 Сравнение реализованных способов

В таблице 1 представлена сводная информация по рассмотренным методам оптимизации. В первой колонке обозначена версия приложения, для которого указаны значения измерений. Далее представлены четыре параметра измерений в Performance вкладке инструмента DevTools. Все эти значения относятся к временному значению в миллисекундах. И в последней колонке указана оценка из отчета Lighthouse из 100 возможных баллов.

Таблица 1 – Время загрузки, выполнения JavaScript, рендеринга и оценка производительность из Lighthouse

Версия	Performance				Lighthouse
	Loading, ms	Scripting, ms	Rendering, ms	Total, ms	Performance mark
оригинальная	3	1424	220	2208	48/100
memo	3	1258	120	1708	50/100
moize	3	1211	116	1648	49/100
lazyload	2	1445	159	2535	57/100

По приведенным данным можно сделать вывод, что для рассмотренного React JS приложения наилучшим способом оптимизации оказался метод кэширования с применением функции moize, т.к. по всем временным показателям он показал наименьшие результаты.

Из проведенного сравнения можно выделить следующие тренды:

1) Для несложного многостраничного React JS приложения без взаимодействия с серверной частью оказался наиболее эффективным такой способ оптимизации как кэширование, а именно – с применением функции moize из одноименной библиотеки.

2) Многопоточность представляется возможным реализовать только в случаях обмена большим количеством информации между клиентской и серверной частями системы или сложных вычислениях, которые можно разбить на независимые от друг друга части.

3) Существует метод «lazy loading», который помогает снизить объем данных при первоначальной загрузке в случае, если на странице веб-приложения присутствует контент, который изначально не виден пользователю и может быть загружен несколько позже. К примеру, это может быть лента постов, где контент, который будет виден только после прокрутки, может быть загружен гораздо позже, чем то содержимое, которое пользователь видит при первой загрузке страницы.

Анализ ведущих трендов показал, что использование рассмотренных методов может дать небольшой прирост в производительности даже для небольшого клиент-серверного приложения.

4 ИССЛЕДОВАНИЕ СЛУЧАЕВ НА КОММЕРЧЕСКОМ ПРОЕКТЕ, ПОВЛИЯВШИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ КЛИЕНТСКОГО ПРИЛОЖЕНИЯ

4.1 Рефакторинг проекта в связи с изменением подхода к реализации собственного компонента и переходом на иную библиотеку по управлению состоянием приложения.

Команде front-end разработки нужно было разработать единый интерфейс для многомодульной системы. Помимо базовых UI компонентов (таблицы, формы для заполнения и т.д.) на макетах присутствовали нестандартные решения, для разработки которых необходимо либо использовать дополнительные библиотеки, либо разрабатывать собственное решение с нуля. Таким нестандартным решением являлось представление списка объектов в виде графа. Трудность состояла в том, что каждый узел такого графа должен быть визуально соединен со следующим по порядку узлом, при этом у самих объектов могла быть зависимость по типу «родитель-потомок», которая должна была отображаться в виде бокового ребра. Примерный вид требуемого представления представлен на рисунке 31.

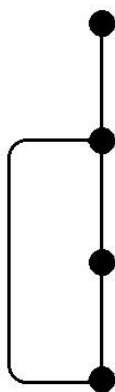


Рисунок 31 – Приблизительное представления требуемого графа

Первой задачей по разработке данного графа было исследование на тему того, какие библиотеки на данный момент позволяют отрисовать подобный граф и насколько они позволяют кастомизировать полученное решение под

собственные стили и требования. Самой подходящей из существующих решений была библиотека `visx` от компании Airbnb.

`Visx` – это коллекция графических примитивов для React-приложений. По своей сути данное решение является дополнением и адаптацией библиотеки `d3` для приложений, реализованных с помощью `ReactJS` фреймворка. Преимущество библиотеки в том, что она разбита на множество пакетов. Это способствует тому, что итоговая сборка будет занимать небольшой размер по памяти. Так же библиотека спроектирована так, что разработчикам не нужно следовать определенной схеме работы с компонентами: можно использовать собственную систему управления состоянием приложения, свою библиотеку анимации или подходящее `CSS-in-JS` решение.

После попытки отрисовать граф с помощью `visx` и адаптировать полученный от API массив объектов был сделан вывод, что все-таки данное решение не совсем подходило по бизнес-требованиям `Revenue Management`: для отрисовки графа средствами `visx` было необходимо преобразовать данные в иерархичный вид, т.е. в массиве первый элемент должен являться «корнем» дерева, а остальные элементы должны быть зависимы от данного элемента как «потомки». Если для случая, когда в полученном массиве нет своих зависимостей типа «родитель-потомок», ещё было удобно организовать иерархичное представление для отрисовки необходимого представления, то уже при наличии хотя бы одной связи возникали трудности в возможностях, которая предоставляла библиотека `visx`. А именно проблема возникла с плохо кастомизируемым компонентом ребра графа. Как было видно на рисунке 31, ребро зависимости «родитель-потомок» должно отображаться слева от общего графа, углы должны быть скругленными.

В связи с тем, что других более подходящих библиотек не было найдено, было принято решение о разработке представления графа полностью с нуля при помощи базовых `svg` элементов.

В изначально утвержденном стеке технологий для разработки клиентского приложения была такая библиотека как `Redux Symbiote`, которая

позволяла получать данные от сервера в зависимости от состояния приложения (загрузка и т.д.).

Если говорить о самом Redux, то это инструмент для управления состоянием данных и пользовательским интерфейсом в JavaScript приложениях с большим количеством сущностей. Обычно данная библиотека используется в связке с такими JavaScript фреймворками, как React, Vue, Angular. Redux решает следующие задачи.

Управление состоянием приложения, работающего с большим количеством данных:

- 1) замена встроенных средств работы с состоянием в React,
- 2) легкое масштабирование приложения,
- 3) избавление от ошибок, связанных с объектом состояния,
- 4) упрощенная отладка,
- 5) повышенная производительность приложения.

Redux базируется на трех принципах, из которых следует характер работы с ним.

1) Единый источник состояния. Одной из важных особенностей приложений, созданных на основе библиотеки Redux, является централизованное хранение данных о состоянии приложения. Все данные хранятся в одном месте, без дублирования. Глобальное состояние организовано в виде дерева объектов, которое называется «state tree». Кроме того, в Redux используются такие понятия, как «источник состояния» и «хранилище». Единый источник данных необходим для централизации и отложенной обработки приложения. Это позволяет упростить процесс управления состоянием приложения и повысить его надежность и масштабируемость.

2) Доступ к состоянию является только для чтения. Глобальное состояние заблокировано для записи. Компоненты приложения могут читать из него, но не переписывать значение. Это позволяет предотвратить

непредсказуемые изменения. При изменениях нужно отправить в состояния действие (action).

3) Изменения происходят только через редукторы. Когда в состояние поступает действие, его обрабатывают редьюсеры (reducers). Это функции, результат выполнения которых зависит только от входных данных. Редьюсеры на основе действия и объекта состояния компонента генерируют новый объект состояния. Результатом работы редьюсера является новый объект состояния с актуальными данными, основывающиеся на информации из объекта-действия. Этот новый объект попадает в дерево состояний вместо старого [31].

Что касается библиотеки `redux-symbiote`, то она позволяет реализовать работу с редьюсерами в более компактном и читабельном виде. Т.е. данный инструмент позволяет избавиться от написания шаблонного кода, который ещё называют «бойлерплейт». В API библиотеки представлена функция `createSymbiote`, которая создаёт «симбиойт», но как указано в примере документации, при деструктуризации возвращается действия (actions) и редьюсер. Когда происходит обращение к `actions.setValue`, то симбиойт вызывает обработчик действия с предыдущим состоянием и всеми переданными аргументами, распределенными после состояния.

Помимо двух обязательных аргументов (`initialState` и `symbiotes`) у функции `createSymbiote` есть третий опциональный параметр. В документации репозитория библиотеки он обозначен как `namespace`, значением по умолчанию которого является пустая строка. Но данный параметр может быть передан как в виде строки, так и в виде объекта. Если будет передана только строка, то значение этой строки будет присвоено параметру `namespace`, который по сути является префиксом для каждого типа действия. Сам же объект может содержать следующие поля: `namespace`, `defaultReducer`, `separator`.

Если вернуться к вопросу отрисовки графа, то он состоял из трёх компонентов: `circle`, который представлял узел графа, и двух элементов `path`, первый – для отрисовки ребра, которое соединяло узел со следующим по рядку узлом (на рисунке 31 это выглядит как обычная вертикальная линия), второй

– для отрисовки ребра зависимости. Так же было необходимо реализовать расчет отступа для линии зависимости от основной линии графа. Величина этого отступа зависит от того, сколько зависимостей встречается в промежутке между «родителем» и «потомком». Т.е. необходимо отрисовать ребра таким образом, чтобы они были читабельные, не накладывались друг на друга.

Помимо самой отрисовки графа, стояла задача сделать его интерактивным, т.к. у каждого объекта, который по сути являлся узлом графа, присутствовала возможность в плане просмотра дополнительной информации, редактирования объекта или удаления объекта. Все эти действия должны влиять на отрисовку графических компонентов. Т.е. если была нажата кнопка по раскрытию описания, то нижерасположенные узлы должны сдвинуться вниз на необходимую величину. Так же должны перерисоваться все необходимые ребра, начиная с выбранного ребра. К сожалению, работа с `svg` не позволяла делать полностью адаптивную верстку, т.е. обновление координат происходило через собственные обработчики. Приходилось ставить `ref`-атрибуты на появляющиеся формы (описание, редактирование объекта), чтобы знать точные размеры дочернего компонента, т.к. его размеры динамически устанавливались в зависимости от параметров входного объекта. Такое количество `ref`-объектов и обработчиков для обновления координат объектов усложняло читабельность кода и было не совсем надёжным в плане отрисовки графа (т.е. была вероятность возникновения неверного поведения (бага) при какой-то частной ситуации). По этой причине было решено разработать второй вариант реализации графа, в реализации которого будут использоваться только `html`-элементы.

У `redux-symbiote` был так же отмечен такой недостаток, как очень малое количество материалов для ознакомления с данной библиотекой. Т.е. на данный момент существует только краткое описание API с примерами на странице GitHub репозитория [32] и обзорная статья на Habr [33]. По причине того, что команда по разработке клиентской части была собрана в короткие сроки из новых сотрудников, которые вряд ли могли иметь опыт работы с

redux-symbiote, было принято решение о миграции архитектуры проекта с использования redux-symbiote на redux-toolkit.

У redux-toolkit имелась обширная документация и множество других ознакомительных материалов, при сравнении двух реализаций небольшого приложения с помощью redux-toolkit и redux-symbiote было выявлено, что вариант на redux-toolkit получился более компактным, интуитивно понятным. Отпала необходимость использовать «эффекты» для общения с API системы, как это было в варианте с redux-symbiote.

Сама библиотека Redux Toolkit появилась в результате того, что разработчики Redux решили устранить те недостатки, которые были в обычном Redux:

1) Одним из недостатков рекомендованных паттернов для написания и организации кода была их сложность и большое количество бойлерплейта. Использование многословных паттернов может привести к нежелательным последствиям, таким как ухудшение читаемости и поддерживаемости кода. Таким образом, необходимость использования сложных паттернов для написания и организации кода является серьезным ограничением для разработчиков и может затруднить процесс создания качественного и эффективного программного обеспечения.

2) Отсутствие встроенных средств управления асинхронным поведением и побочными эффектами является одним из недостатков многих современных языков программирования. Это приводит к необходимости выбора подходящего инструмента из множества дополнений, написанных сторонними разработчиками. Такой подход может быть неэффективным и затруднить процесс разработки, поскольку не все инструменты могут быть совместимы с основным кодом и требуют дополнительных усилий для интеграции. Кроме того, выбор подходящего инструмента может быть сложным из-за большого количества доступных вариантов.

Redux Toolkit выполняет следующие функции:

1) упрощает работу с типичными задачами и кодом Redux,

2) позволяет использовать лучшие практики (best practices) Redux по умолчанию,

3) предлагает решения, уменьшающие недоверие к бойлерплейтам.

В Redux Toolkit можно выделить следующие наиболее значимые функции:

1) `configureStore` – создание и настройка хранилища,
2) `createReducer` – описание и создание редьюсера,
3) `createAction` – функция, возвращающая функцию создателя действия для заданной строки типа действия,

4) `createSlice` – объединение функционала `createReducer` и `createAction`,

5) `createSelector` – утилита, которая была реэкспортирована из библиотеки `Reselct` для простоты использования [34].

Так же вместе с Redux Toolkit было принято решение об использовании RTK Query. Это инструмент для получения и кэширования данных, предназначенный для упрощения распространенных случаев загрузки данных в веб-приложении, избавляя от необходимости вручную писать логику загрузки и кэширования данных. По сути, RTK Query является дополнением, включенный в пакет Redux Toolkit, его функциональность построена поверх других API в Redux Toolkit.

Клиентским веб-приложениям обычно требуется получать данные с сервера для их последующего отображения. Также им обычно необходимо вносить обновления в эти данные (создание новой сущность, обновления полученного объекта, удаление объекта) и синхронизировать кэшированные данные на клиенте с данными на сервере. Это усложняется необходимостью реализации других вариантов поведения, используемых в современных приложениях:

1) отслеживание состояния загрузки для отображения UI спиннера (визуальный элемент пользовательского интерфейса для демонстрации состояния загрузки данных),

- 2) избежание дублирования запросов на один и те же данные,
- 3) управление временем хранения кэша при взаимодействии пользователя с интерфейсом.

Само ядро Redux всегда было очень минимальным – Redux никогда не включал ничего встроенного, чтобы предоставлять возможность реализации различных вариантов использования. В документации Redux были общие паттерны по действиям (actions) в жизненном цикле запроса для отслеживания состояния загрузки и результатов запроса. И в итоге `createAsyncThunk` функция в API Redux Toolkit была разработана для абстрагирования от этого типичного шаблона. Однако, пользователям по-прежнему приходится писать значительные объемы логики редьюсера для управления состоянием загрузки и кэшированными данными.

Сообщество React осознало, что «выборка и кэширование данных» на самом деле представляет собой другой набор проблем, чем «управление состоянием». Из-за того, что варианты использования по кэшированию данных достаточно различны, стоит использовать инструменты, специально созданные для необходимого варианта использования выборки данных. Как указано в самой документации библиотеки RTK Query, разработчики вдохновлялись другими инструментами, которые первыми предложили решения для получения данных, таких как Apollo Client, React Query, Urql и SWR, но они добавили уникальный подход к дизайну своего API:

- 1) логика выборки и кэширования данных построена на основе API-интерфейсов `createSlice` и `createAsyncThunk` в Redux Toolkit,
- 2) функциональные возможности RTK Query можно использовать с любым уровнем пользовательского интерфейса, т.к. Redux Toolkit не зависит от пользовательского интерфейса,
- 3) эндпоинты API определяются заранее, включая то, как генерировать параметры запроса из аргументов и преобразовывать ответы для кэширования,

4) RTK Query так же способен генерировать хуки самого фреймворка React, которые инкапсулируют весь процесс выборки данных, предоставляют данные и параметр `isLoading` для компонентов и управляют временем хранения кэшированных данных по мере монтирования (первоначального рендеринга компонента в DOM) и размонтирования (DOM-узел, созданный компонентом, удаляется) компонентов,

5) RTK Query представляет опции «`cache entry lifecycle`», которые позволяют использовать такие варианты использования, как потоковая передача обновлений кэша через сообщения веб-сокета после выборки исходных данных,

6) Генерация фрагментов API из схем OpenAPI и GraphQL,

7) RTK Query полностью написан на TypeScript и предназначен для обеспечения удобного использования в TypeScript решениях [35].

Из всей функциональности RTK Query можно выделить следующее:

1) `createApi()` – данная функция является ядром функциональности RTK Query. Она позволяет определить набор эндпоинтов, описать, как должны извлекать данные из них, включая возможность преобразования этих данных (с помощью опции `transform`). В большинстве случаев данная функция должна использоваться один раз для каждого приложения, следуя подходу «one API slice per base URL», т.к. один фрагмент API на базовый URL-адрес [36].

2) `fetchBaseQuery()` – функция, которая представляет собой небольшую оболочку для обычной функции `fetch`. Цель этой оболочки – упростить HTTP-запросы. Она генерирует метод по выборки данных, совместимый с параметром параметром конфигурации `baseQuery` RTK Query. Принимаются все стандартные параметры из интерфейса `RequestInit`, а также `baseUrl`, функцию `prepareHeader`, дополнительную функцию `fetch`, функцию `paramsSerializer` и тайм-аут [37].

3) `setupListeners()` – утилита, которая используется для включения поведения `refetchOnFocus` и `refetchOnReconnect`. В качестве входного аргумента требуется метод `dispatch` хранилища (`store`) приложения [38].

4) Хуки для запросов/эндпоинтов, которые можно использовать в результате применения `createApi` функции: `useQuery`, `useMutation`, `useQueryState`, `useQuerySubscription`, `useLazyQuery`, `useLazyQuerySubscription`, `usePrefetch`.

`UseQuery` – это React хук, который автоматически запускает получение данных из эндпоинта, «подписывает» компонент на кэшированные данные из хранилища `Redux`. Компонент будет ререндериться по мере изменения состояния загрузки и доступности данных. Аргумент запроса используется в качестве ключа кэша. Изменения аргумента запроса сообщит хуку о повторной выборке данных, если они еще не существуют в кэше, и хук вернет данные для этого аргумента запроса, как только он будет доступен. Данный хук сочетает в себе функциональность как `useQueryState`, так и `useQuerySubscription`, и предназначен для использования в большинстве ситуаций.

`UseMutation` – это React хук, который позволяет инициировать запрос на обновление для данной конечной точки и подписывается на компонент для чтения статуса запроса из хранилища `Redux`. Компонент будет перерисовываться при изменении статуса загрузки.

`UseQueryState` – это React хук, который считывает статус запроса и кэшированные данные из хранилища `Redux`. Компонент будет ререндериться по мере изменения состояния загрузки и доступности данных. Данный хук не запускает получение новых данных.

`UseQuerySubscription` – это React хук, который автоматически запускает получение данных из эндпоинта и «подписывает» компонент на кэшированные данные. Данный хук не возвращает статус запроса или кэшированные данные.

`UseLazyQuery` – это React хук, похожий на `useQuery`, но с ручным управлением, когда происходит получение данных. Этот хук включает в себя функциональность `useLazyQuerySubscription`.

UseLazyQuerySubscription – это React хук, похожий на useQuerySubscription, но с ручным управлением, как и UseLazyQuery. Данный хук не возвращает статус запроса или кэшированные данные.

UsePrefetch – это React хук, который можно использовать для предварительной загрузки данных [39].

Если применение Redux Toolkit и RTK Query коснулось всей архитектуры приложения, то следующее изменение было связано именно с методом отрисовки графа, постановка задачи которого описана в первой главе (см. рисунок 1). Был предложен совершенно иной подход – реализация элементов графа (узлов, ребер) только через стандартные html-элементы, а именно – div, универсальный контейнер для потокового контента. Такой подход позволял избавиться от лишних вычислений (пересчет координат в первой реализации) за счёт адаптивности стандартных html-элементов.

4.2 Сравнение производительности клиентского приложения от изначальной до конечной реализации

Чтобы проанализировать полученный результат в плане производительности, можно составить 4 варианта приложения, начиная с первоначальной реализации и заканчивая финальной версией. Данные варианты представлены в таблице 2.

Таблица 2 – Варианты комбинаций способа реализации графа и библиотеки для работы с Redux

	SVG	HTML
Symbiote	1 вариант	2 вариант
Toolkit	3 вариант	4 вариант

Первый вариант является первоначальной реализацией поставленной задачи – отрисовка графа и иных элементов интерфейса для назначенной функциональности. Была проведена оценка производительности в «Performance» и «Lighthouse». Измерения в Performance представлены на рисунке 32, а в Lighthouse – на рисунке 33.

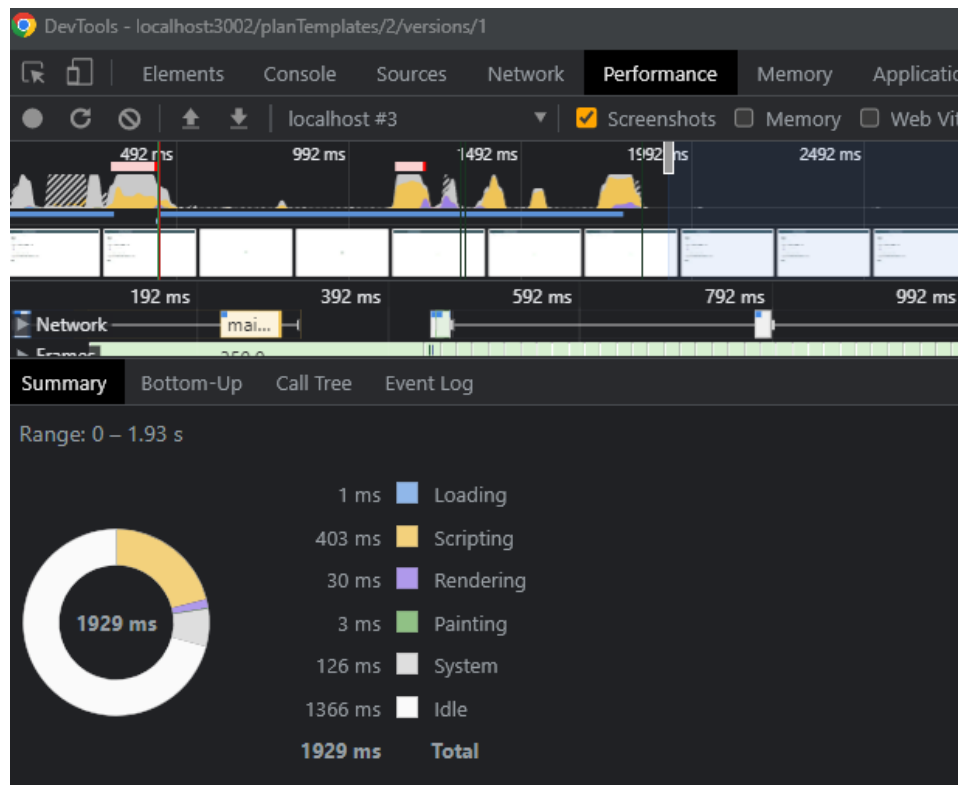


Рисунок 32 – Суммарная информация о производительности приложения (вариант 1)

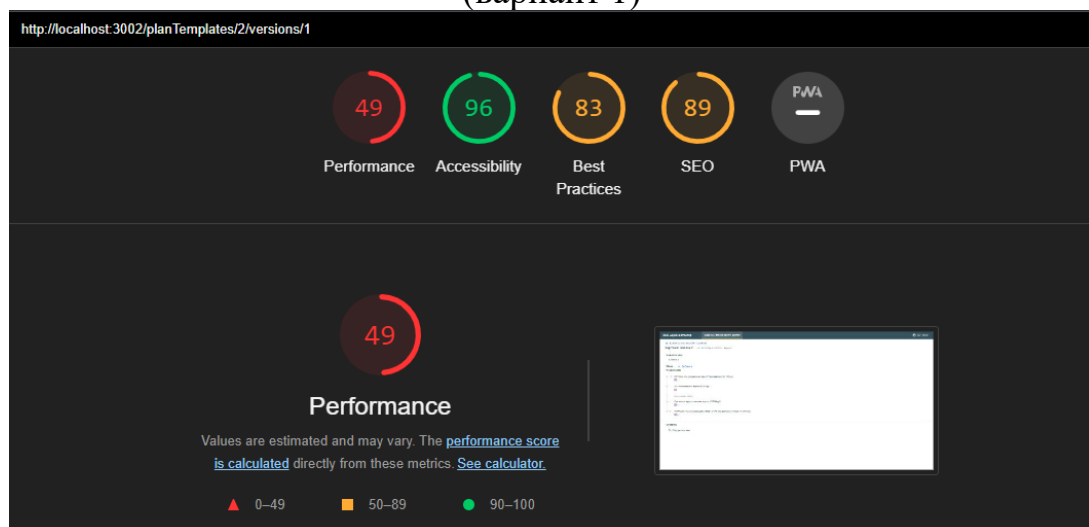


Рисунок 33 – Оценки из отчета Lighthouse (вариант 1)

В качестве второго варианта было реализовано сочетание библиотеки Redux-symbiote и реализация построения графа через стандартные HTML-элементы. Измерения производительности в Performance представлены на рисунке 34, а в Lighthouse – на рисунке 35.

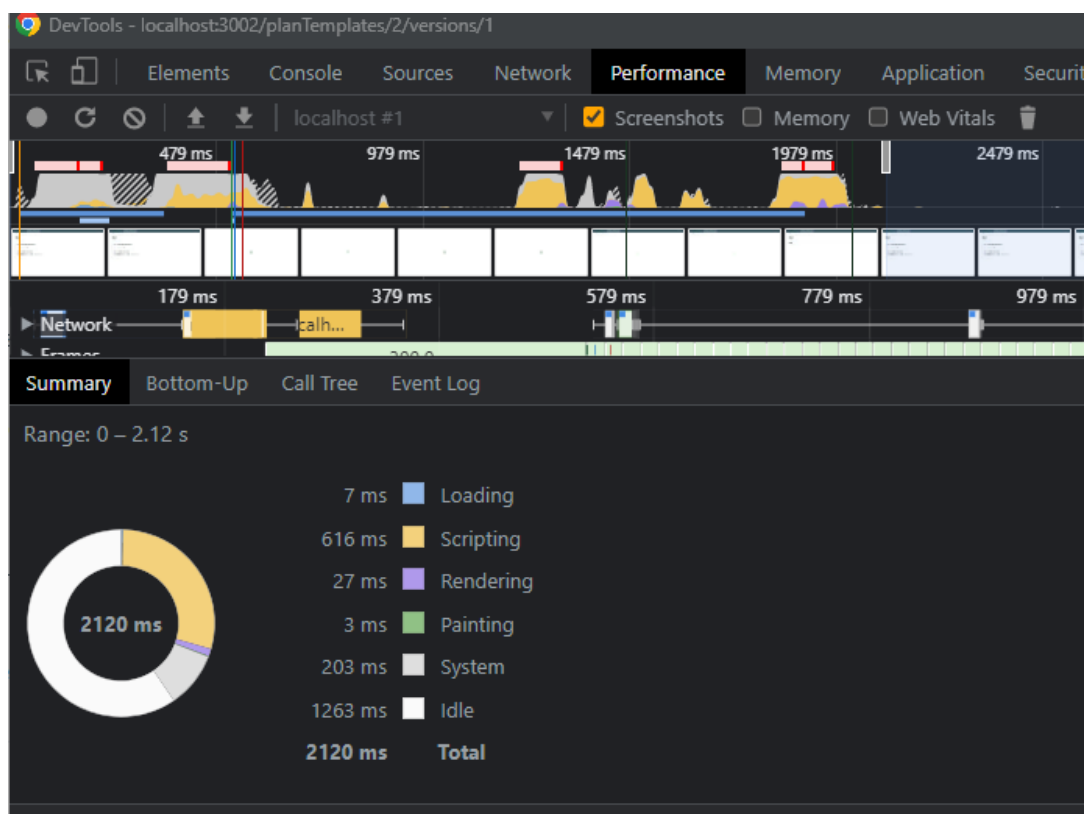


Рисунок 34 – Суммарная информация о производительности приложения (вариант 2)

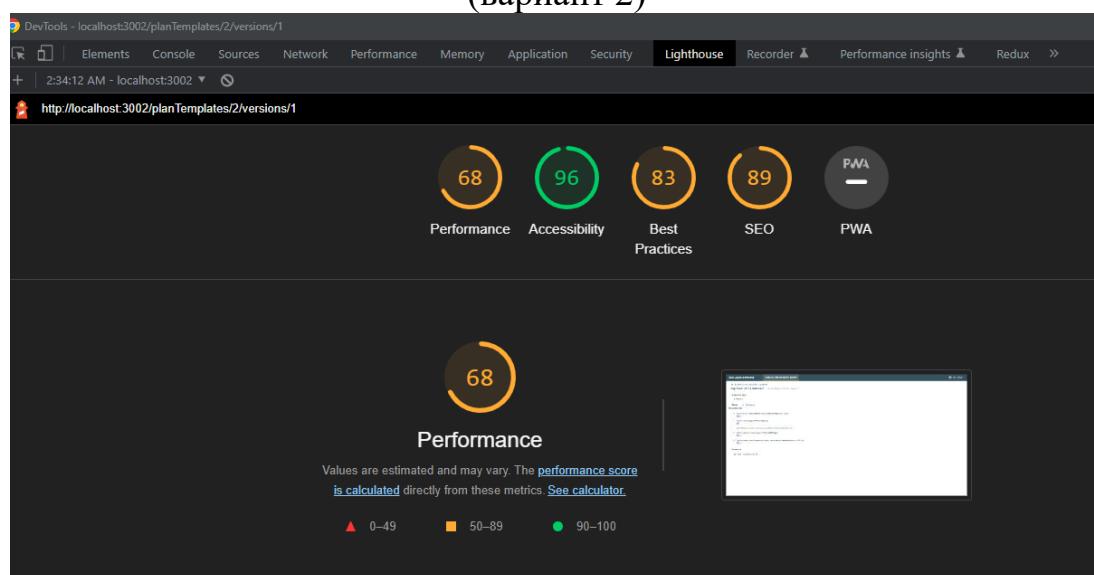


Рисунок 35 – Оценки из отчета Lighthouse (вариант 2)

Третий вариант являлся промежуточной стадией для перехода к окончательному варианту. Приоритетной задачей на проекте была миграция уже сделанной функциональности (фичи) на новую архитектуру приложения, в котором получения данных с сервера осуществлялось теперь с помощью

Redux-toolkit, а не redux-symbiote. Измерения производительности в Performance представлены на рисунке 36, а в Lighthouse – на рисунке 37.

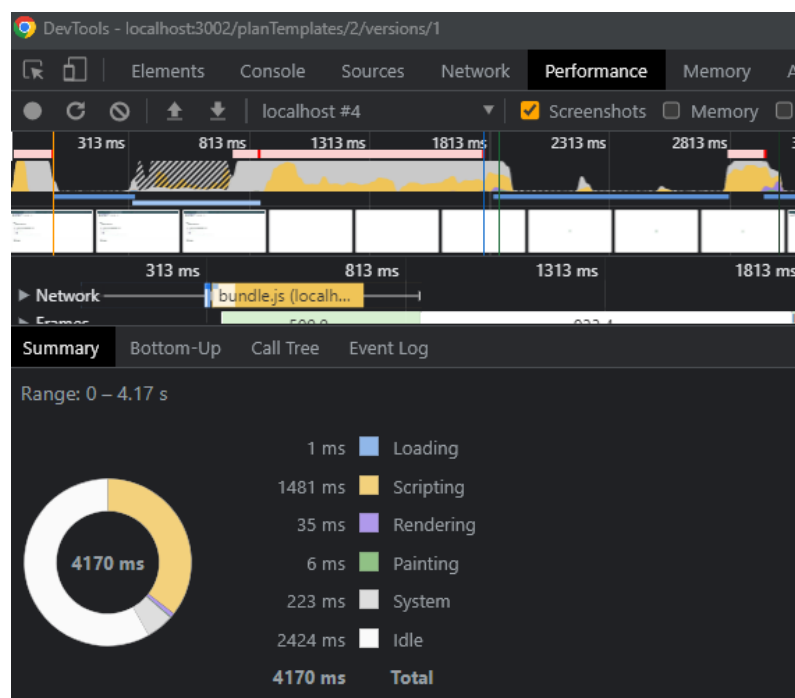


Рисунок 36 – Суммарная информация о производительности приложения (вариант 3)

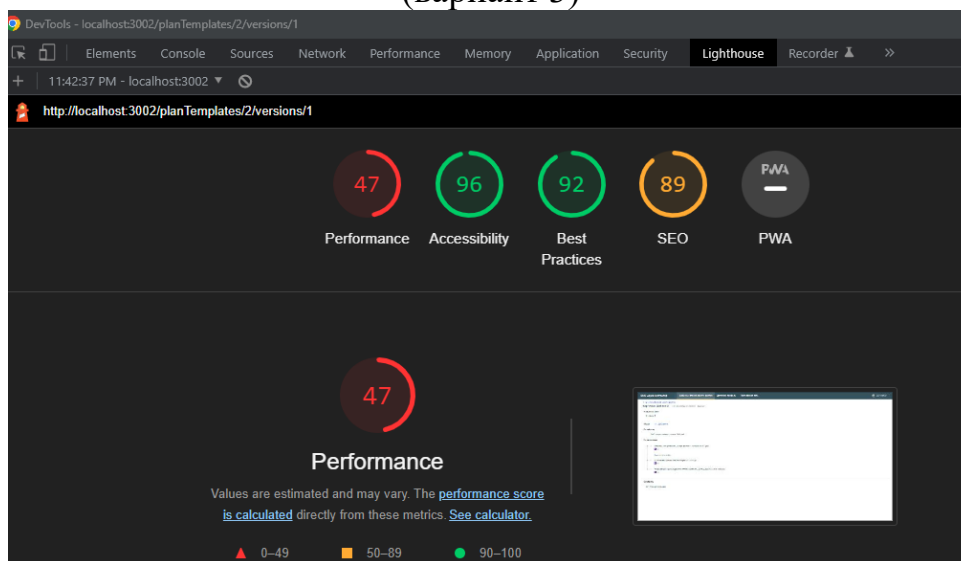


Рисунок 37 – Оценки из отчета Lighthouse (вариант 3)

Окончательный вариант представляет собой как применение инструмента Redux-toolkit для операций с данными, так и реализацию графа с помощью стандартных HTML-элементов. Измерения в Performance представлены на рисунке 38, а в Lighthouse – на рисунке 39.

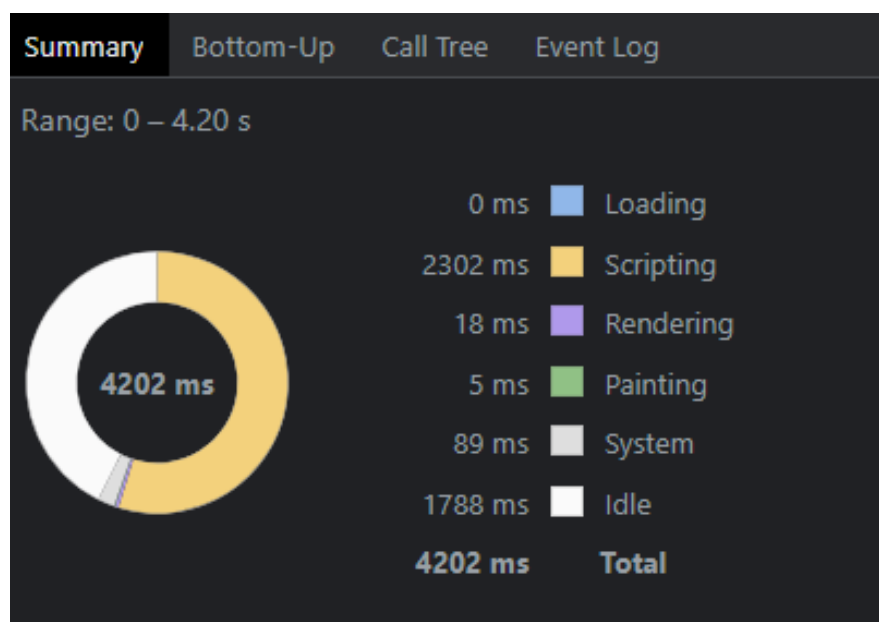


Рисунок 38 – Суммарная информация о производительности приложения (вариант 4)

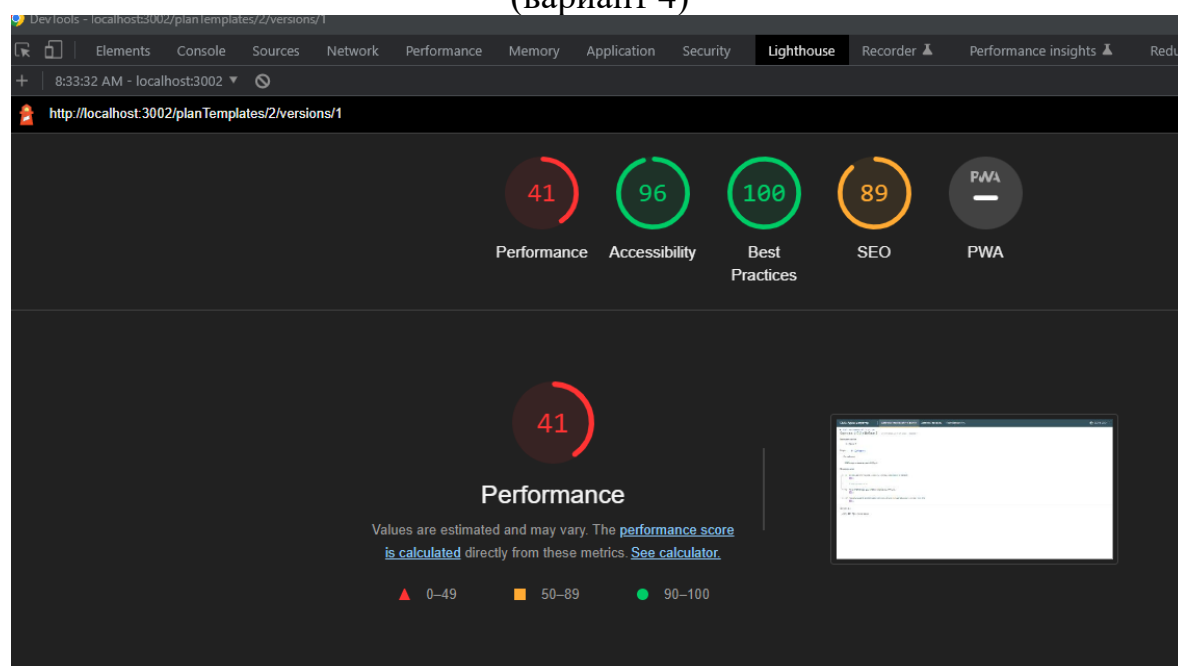


Рисунок 39 – Оценки из отчета Lighthouse (вариант 4)

В таблице 3 представлена сводная информация по реализованным вариантам. В первой колонке обозначен номер варианта. Далее представлены четыре параметра измерений в Performance вкладке инструмента DevTools. Все эти значения относятся к временному значению в миллисекундах. И в последней колонке указана оценка из отчета Lighthouse из 100 возможных баллов.

Таблица 3 – Время загрузки, выполнения JavaScript, рендеринга и оценка производительность из Lighthouse

Версия	Performance				Lighthouse
	Loading, ms	Scripting, ms	Rendering, ms	Total, ms	Performance mark
Вариант 1	1	403	30	1929	49/100
Вариант 2	7	616	27	2120	68/100
Вариант 3	1	1481	35	4170	47/100
Вариант 4	0	2302	18	4202	41/100

По сводным данным в таблице видно, что производительность улучшилась в плане времени рендеринга страницы после замены всех svg-элементов на div-контейнеры в реализации компонентов графа. Разницу в производительности между вариантами с symbiote и toolkit можно объяснить размером используемых библиотек, что в итоге повлияло на итоговый файл сборки: redux-symbiote – 40.0 КБ, redux-toolkit – 12.5 МБ [40].

4.3 Анализ бесконечного ре-рендеринга компонента из-за неверного применения функции из библиотеки по управлению состоянием компонента

После завершения рефакторинга и реализации новой функциональности командой тестировщиков была выявлена ошибка в поведении компонента DatePicker из библиотеки Ant design [41]. Для данного компонента задавалось начальное значение даты, но пользователь имел возможность поменять её. При попытке переключения на следующий или предыдущий месяц, а также на следующий или предыдущий год отображение необходимого месяца происходило, но секунду спустя всё возвращалось к изначальному виду. Т.е. для компонента было задано изначальное значение в виде 23 марта 2023 года, то пользователь при попытке переключить календарь для выбора даты на апрель видел, как на короткое время календарь для апреля все-таки отображался, но уже через секунду он видел март 2023 года.

При анализе данного некорректного поведения компонента было сформулировано предположение, что причиной может быть очень частый ре-

рендеринг компонента, в котором находился элемент `DatePicker`. Для подтверждения этого предположения использовался инструмент `Profiler` из `React Developer Tools` [42]. С помощью данного инструмента можно сделать запись, где будет видно, когда и какой компонент отрендерился заново. Всё это представлено в виде `flame`-графика. В настройках можно включить опцию по отображению информации о том, что спровоцировало рендеринг.

Таким образом выяснилось, что каждую секунду обновлялся родительских компонент, где использовался хук `useQuery` из библиотеки `RTK query`. В аргументах этого хука был прописан параметр `selectFromResult`, который позволяет изменить возвращаемое значение хука для получения подмножества результата [43]. С помощью этого параметра была реализована логика, что в качестве возвращаемого результата будет либо `state.data`, либо пустой массив. После исследования вопроса о том, как именно работает хук `useQuery`, выяснилось, что вне зависимости от того, каким было до нового рендера значение результата работы хука, новый массив или объект алгоритмы библиотеки `RTK Query` считают за новое значение, что вызывало `LayoutEffect` [44]. Решением данной проблемы являлось следующим – была сделана отдельная переменная-константа в качестве значения по умолчанию, если хук возвращает значение типа `undefined`.

ЗАКЛЮЧЕНИЕ

При выполнении выпускной квалификационной работы была полностью достигнута цель исследования оптимизации рендеринга компонентов в ReactJS, а также решены все задачи, сформулированные во введении.

Было исследовано современное состояние проблемы оптимизации рендеринга компонента, реализованных с помощью фреймворка ReactJS. По результатам исследования были сделаны выводы, что на настоящий момент не существует подробного описания, насколько именно улучшается производительность от того или иного метода.

Исследована работа клиентского фреймворка ReactJS. Было выявлено, что ключевым моментом, влияющим на производительность приложения, является обновление виртуального DOM-дерева, т.е. элемент интерфейса, состояние которого изменилось из-за действий пользователя или обновления данных.

Исследованы методы оптимизации рендеринга в клиентских приложениях. Было разработано клиент-серверное приложение для проведения тестирования и сравнение популярных методов оптимизации. Было выявлено, что наиболее эффективным методом является кэширование, а именно мемоизация.

Были проанализированы случаи на коммерческом проекте, где замена библиотеки по управлению состоянием приложения, изменение подхода к реализации компонента и некорректное задание параметра `selectFromResult` хука `useQuery` влияли на производительность приложения. На основании этого анализа можно было убедиться, что размер применяемых в проекте библиотек оказывает влияние на производительность приложения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Зачем и как проверять скорость загрузки сайта? [Электронный ресурс] URL: <https://habr.com/ru/post/507746/> (дата обращения: 07.11.2021).
2. Торопкин Р.А., Зиновьев Я.В., Рассказов Н.С., Митрохин М.А. Технологии оптимизации работы сайта на примере аналитической системы публикационной активности пензенского государственного университета // Модели, системы, сети в экономике, технике, природе и обществе. 2020. № 4 (36). С. 71–78. URL: <https://cyberleninka.ru/article/n/tehnologii-optimizatsii-raboty-sayta-na-primere-analiticheskoy-sistemy-publikatsionnoy-aktivnosti-penzenskogo-gosudarstvennogo> (дата обращения 09.11.2021).
3. How to Make Your Website Load Faster With WebP Images. [Электронный ресурс] URL: <https://techstacker.com/load-website-faster-with-webp-images/mhzdwjfko9eb4fbep/> (дата обращения: 10.11.2021).
4. Start Performance Budgeting. [Электронный ресурс] URL: <https://medium.com/@addyosmani/start-performance-budgeting-dabde04cf6a3> (дата обращения: 10.11.2021).
5. Как ускорить загрузку сайта и улучшить поведенческие факторы. [Электронный ресурс] URL: <https://ru.megaindex.com/blog/service-workers-site-speed> (дата обращения: 11.11.2021).
6. Князев И.В. Продвинутое кеширование и оптимизация веб-приложений с помощью технологии SWR // Sciences of Europe. 2021. № 73 (1). С. 47-49. URL: <https://cyberleninka.ru/article/n/prodvinutoe-keshirovanie-i-optimizatsiya-veb-prilozheniy-s-pomoschyu-tehnologii-swr> (дата обращения: 17.11.2021).
7. Князев И.В. Анализ работы приложения с использованием server-side rendering: миграция, настройка и развертывание приложения Next.js // Sciences of Europe. 2021. № 76 (1). С. 71-74. URL: <https://cyberleninka.ru/article/n/analiz-raboty-prilozheniya-s-ispolzovaniem-server-side-rendering-migratsiya-nastroyka-i-razvertyvanie-prilozheniya-next-js> (дата обращения: 25.11.2021).
8. Background processing using web workers. [Электронный ресурс] URL: <https://angular.io/guide/web-worker> (дата обращения: 29.11.2021).

9. Бородин О.В., Егунов В.А. Многопоточная обработка изображений с использованием API Web-workers // CASPIAN JOURNAL: Control and High Technologies. 2021. № 3 (55). С. 33-46. URL: <https://cyberleninka.ru/article/n/mnogopotochnaya-obrabotka-izobrazheniy-s-ispolzovaniem-api-web-workers> (дата обращения: 3.12.2021).
10. Gowda V., Rangaswamy S. Addressing the Limitations of React JS // International Research Journal of Engineering and Technology (IRJET). 2020. № 7(4). С. 5065-5068. URL: <https://www.irjet.net/archives/V7/i4/IRJET-V7I4966.pdf> (дата обращения: 9.12.2021).
11. Reddy M.P., Mishra S.P. Analysis of Component Libraries for React JS // International Advanced Research Journal in Science, Engineering and Technology (IARJSET). 2021. № 8(6). С. 43-46. URL: https://www.researchgate.net/publication/353173122_Analysis_of_Component_Libraries_for_React_JS (дата обращения: 14.12.2021).
12. Mukhiya S.K., Hung H.K. An Architectural Style for Single Page Scalable Modern Web Application // International Journal of Recent Research Aspects. 2018. № 5(4). С. 6-13. URL: https://www.researchgate.net/publication/335259756_An_Architectural_Style_for_Single_Page_Scalable_Modern_Web_Application (дата обращения: 17.12.2021).
13. Агафонова М.В. Обзор методов оптимизации рендеринга приложения, реализованного с помощью клиентских фреймворков // Современное образование: традиции и инновации. – 2022. N. 1 – С. 130-135.
14. Бэнкс А. React: современные шаблоны для разработки приложений / А. Бэнкс, Е. Порселло – СПб.: Питер, 2022 – 320 с.
15. React – что это за библиотека компонентов: для чего нужен фреймворк [Электронный ресурс] URL: <https://blog.skillfactory.ru/glossary/react/> (дата обращения: 03.10.2022).
16. Использование хука эффекта [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-effect.html> (дата обращения: 07.10.2022).

17. Правила хуков [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-rules.html> (дата обращения: 09.10.2022).
18. Создание пользовательских хуков [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-custom.html> (дата обращения: 12.10.2022).
19. Использование хука состояния [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-state.html> (дата обращения: 15.10.2022).
20. Справочник API хуков [Электронный ресурс] URL: <https://ru.reactjs.org/docs/hooks-reference.html> (дата обращения: 20.10.2022).
21. React-компоненты шаблонов проектирования [Электронный ресурс] URL: <https://habr.com/ru/company/otus/blog/547564/> (дата обращения: 05.11.2022).
22. 5 продвинутых паттернов React-разработки [Электронный ресурс] URL: <https://proglib.io/p/5-prodvinutyh-patternov-react-razrabotki-2021-05-30> (дата обращения: 12.11.2022).
23. Flux: Архитектура приложений на React.js – всестороннее исследование [Электронный ресурс] URL: <https://medium.com/@marina.kovalyova/flux-the-react-js-application-architecture-773f515d068d> (дата обращения: 24.11.2022).
24. React | Введение в Flux [Электронный ресурс] URL: <https://metanit.com/web/react/5.1.php> (дата обращения: 10.12.2022).
25. Как оптимизировать сайты с помощью Lighthouse [Электронный ресурс] URL: <https://htmlacademy.ru/blog/articles/lighthouse> (дата обращения: 17.06.2022).
26. Как провести аудит страницы при помощи PageSpeed Insights и ускорить ее [Электронный ресурс] URL: <https://vc.ru/seo/298561-kak-provesti-audit-stranicy-pri-pomoshchi-pagespeed-insights-i-uskorit-ee> (дата обращения: 17.06.2022).
27. Что такое React.memo и как он работает [Электронный ресурс] <https://nuancesprog.ru/p/15532> (дата обращения: 19.06.2022).

28. Know bundle size when you import with VSCode Extension [Электронный ресурс] <https://devtips.theanubhav.com/posts/cost-of-import> (дата обращения: 19.06.2022).
29. Moize – npm [Электронный ресурс] <https://www.npmjs.com/package/moize> (дата обращения: 19.06.2022).
30. The top choices for React lazy loading libraries in 2021 [Электронный ресурс] <https://blog.logrocket.com/the-top-choices-for-react-lazy-loading-libraries-in-2021/> (дата обращения: 19.06.2022).
31. Redux - что это такое и зачем нужна библиотека JavaScript с простым API [Электронный ресурс] URL: <https://blog.skillfactory.ru/glossary/redux/> (дата обращения: 02.10.2022).
32. Репозиторий sergeysova/redux-symbiote [Электронный ресурс] URL: <https://github.com/sergeysova/redux-symbiote> (дата обращения: 05.10.2022).
33. Redux-symbiote – пишем действия и редьюсеры почти без боли [Электронный ресурс] URL: <https://habr.com/ru/post/442346/> (дата обращения: 09.10.2022)
34. Redux Toolkit как средство эффективной Redux-разработки [Электронный ресурс] URL: <https://habr.com/ru/company/inobitec/blog/481288/> (дата обращения: 24.10.2022).
35. RTK Query Overview [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/overview> (дата обращения: 26.10.2022).
36. createApi | Redux Toolkit [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/api/createApi> (дата обращения: 26.10.2022).
37. fetchBaseQuery | Redux Toolkit [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/api/fetchBaseQuery> (дата обращения: 29.10.2022).
38. setupListeners | Redux Toolkit [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/api/setupListeners> (дата обращения: 07.11.2022).
39. API Slices: React Hooks [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/api/created-api/hooks> (дата обращения: 15.11.2022).

40. Агафонова М.В. Обзор влияния различных библиотек и различной реализации компонента на производительность ReactJS приложения // Научно-технические инновации и веб-технологии. – 2023 (принята к публикации).
41. Ant Design - The world's second most popular React UI framework [Электронный ресурс] URL: <https://ant.design> (дата обращения 15.04.2023).
42. React Developer Tools – React [Электронный ресурс] URL: <https://react.dev/learn/react-developer-tools?ref=javascriptguides.com> (дата обращения 15.04.2023).
43. Queries | Redux Toolkit [Электронный ресурс] URL: <https://redux-toolkit.js.org/rtk-query/usage/queries> (дата обращения 15.04.2023).
44. Справочник API хуков – React [Электронный ресурс] URL: <https://ru.legacy.reactjs.org/docs/hooks-reference.html#uselayouteffect> (дата обращения 15.04.2023).
45. Обзор всех инструментов разработчика Chrome DevTools [Электронный ресурс] URL: <https://habr.com/ru/company/simbirsoft/blog/337116/> (дата обращения: 20.06.2022).
46. Multi Page Application with React [Электронный ресурс] URL: <https://itnext.io/building-multi-page-application-with-react-f5a338489694> (дата обращения: 15.06.2022).
47. Чем функциональные компоненты React отличаются от компонентов, основанных на классах? [Электронный ресурс] URL: <https://habr.com/ru/company/ruvds/blog/444348/> (дата обращения: 15.06.2022).
48. Hooks vs Class компоненты [Электронный ресурс] URL: <https://frontend-stuff.com/blog/react-components-hooks-vs-classes/> (дата обращения: 15.06.2022).
49. React Architecture Patterns for Your Projects [Электронный ресурс] URL: <https://blog.openreplay.com/react-architecture-patterns-for-your-projects> (дата обращения: 15.06.2022).
50. Wieruch R. The Road to React / R. Wieruch. – Leanpub, 2020. – 228 с.

51. Roldan C.S. React Design Patterns and Best Practices / C. S. Roldan. – Packt Publishing, 2019. – 350 с.
52. React Architecture Patterns for Your Projects [Электронный ресурс] URL: <https://blog.openreplay.com/react-architecture-patterns-for-your-projects> (дата обращения: 15.03.2023).
53. Обзор фреймворка TailwindCSS: чем он хорош и кому будет полезен [Электронный ресурс] URL: <https://timeweb.com/ru/community/articles/chto-takoe-tailwindcss-zachem-nuzhen-i-chem-horosh> (дата обращения: 18.03.2023).
54. Макет приложения в Figma [Электронный ресурс] URL: <https://www.figma.com/file/K3SBXhy2FlFddSAMZmZT4x/Bubble-app?node-id=0%3A1&t=GLaYMxTqL8ADb0zj-1> (дата обращения: 10.04.2023).
55. Репозиторий в GitLab [Электронный ресурс] URL: <https://gitlab.com/labs211/FinalTask/-/tree/main> (дата обращения: 10.04.2023).
56. Docker-репозиторий для серверного приложения проекта BubbleApp [Электронный ресурс] URL: <https://hub.docker.com/repository/docker/marinaagafonova/bubbleapp-backend> (дата обращения: 10.04.2023).
57. Docker-репозиторий для клиентского приложения проекта BubbleApp [Электронный ресурс] URL: <https://hub.docker.com/repository/docker/marinaagafonova/bubbleapp-frontend> (дата обращения: 10.04.2023).
58. Flask: веб-фреймворк в Python, документация [Электронный ресурс] URL: <https://blog.skillfactory.ru/glossary/flask/> (дата обращения: 20.03.2023).
59. MongoDB | Введение [Электронный ресурс] URL: <https://metanit.com/nosql/mongodb/1.1.php> (дата обращения: 22.03.2023).
60. How To Use MongoDB in a Flask Application [Электронный ресурс] URL: <https://www.digitalocean.com/community/tutorials/how-to-use-mongodb-in-a-flask-application> (дата обращения 22.03.2023).
61. GitLab CI/CD [Электронный ресурс] URL: <https://doka.guide/tools/gitlab-ci-cd/> (дата обращения 30.03.2023).