МИНИСТЕРСТВО НАУКИ И ВЫСШГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет *ИСТ* Кафедра *ИВК*

К ЗАЩИТЕ ДОПУСТИТЬ
Зав. кафедрой

_____ / _____
подпись инициалы, фамилия
«____» _____ 20___ г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема ***Увеличение производительности реактивных web-страниц***

Обучающийся _____ / *К. С. Бурханов* /
подпись инициалы, фамилия

Обозначение работы *МД-УлГТУ-09.04.02-18/1019-2020* Группа *ИСТмд-21*

Направление подготовки *09.04.02 «Информационные системы и технологии»*
код, наименование

_____

Руководитель работы _____ / *В. В. Родионов* /
подпись, дата инициалы, фамилия

Рецензент _____ / *В. Н. Негода* /
подпись, дата инициалы, фамилия

Консультанты:

_____ _____ / _____ /
наименование раздела подпись, дата инициалы, фамилия

_____ _____ / _____ /
наименование раздела подпись, дата инициалы, фамилия

Ульяновск, 2020

Аннотация

Магистерская диссертация Бурханова Кирилла Сергеевича по теме «Увеличение производительности реактивных web-страниц». Руководитель Родионов Виктор Викторович. Выпускная работа защищена на кафедре «Измерительно-вычислительные комплексы» Ульяновского государственного технического университета «2» июля 2020 года.

Текст диссертации: с. 74, глав 3, прил. 1, ил. 9, табл. 13, ист. 12

Ключевые слова: реактивное программирование, DOM, гидратация, HTML, веб-приложения, React.

Данная работа посвящена исследованию такой проблематики, как производительность реактивных фреймворков. Были исследованы текущие способы решения проблем с производительностью, проанализированы и протестированы популярные фреймфорки на наличие данных проблем. На основе полученных данных, были выявлены технологии и способы решения, которые были использованы при создании своего микрофреймворка и помогли добиться производительности в 6 раз лучше, чем у популярных фреймворков, при этом не теряя те плюсы, которые предоставляет реактивность. Также данная работа позволила выявить общие положения, которые необходимо соблюдать, чтобы добиться еще большей производительности.

Содержание

Перечень использованных терминов и обозначений

Open-source – Открытое программное обеспечение.

SOLID (сокр. от англ. single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion) в программировании – 5 основных принципов объектно-ориентированного программирования и проектирования.

Асинхронность – характеризует процессы, не совпадающие во времени.

User-friendly – интуитивно понятен.

Canvas - это новый элемент HTML5, который предназначен для создания растрового изображения при помощи JavaScript.

SVG (от англ. Scalable Vector Graphics) — язык разметки масштабируемой векторной графики

HTML (от англ. HyperText Markup Language) — стандартизированный язык разметки документов во Всемирной паутине

CSS (от англ. Cascading Style Sheets) — формальный язык описания внешнего вида документа, написанного с использованием языка разметки.

TCP (от англ. Transmission Control Protocol) — один из основных протоколов передачи данных интернета, предназначенный для управления передачей данных.

Парсинг (Parsing) – это принятое в информатике определение синтаксического анализа.

Кэширование – способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов.

Скрипт (сценарий) — это последовательность действий, описанных с помощью скриптового языка программирования.

DNS (от англ. Domain Name System) — компьютерная распределённая система для получения информации о доменах.

Нативный (англ. native) — это прикладная программа или язык программирования, который были разработаны для использования на определённой платформе или устройстве.

Рендеринг (англ. rendering) — термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы.

Фреймворк (англ. framework) — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Транзакции (англ. transaction) — группа последовательных операций, которая представляет собой логическую единицу работы с данными.

Пользовательский интерфейс (UI — англ. user interface) — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

JS – JavaScript

Введение

Современная веб-разработка сосредоточена на компонентно-ориентированном подходе, который пришел из мобильной разработки и успешно внедрился в веб с помощью таких компаний, как Facebook и Google.

Они создали реактивные фреймворки (angular, react), которые полностью изменили подход к привычной веб-разработке, сделав ее качественной и масштабируемой, что позволяет создавать приложения любой сложности с помощью веб технологий.

Так как Facebook и Google разрабатывают полностью динамические приложения, где более 90% компонентов – реактивны, то есть они имеют свое состояние и работают в фоновом режиме, ожидая обновление данных, чтобы изменить интерфейс, поэтому использование реактивных фреймворков полностью оправдывает себя.

Но большинство веб-сайтов не такие. Количество динамических частей в приложении составляет 20-30% и использование реактивных фреймворков замедляет работу сайта, особенно на мобильных устройствах.

На сегодняшний день проблема производительности реактивных фреймворков стоит достаточно остро, ведь производительность влияет не только на процент отказов пользователей, но и на продвижение в поисковых системах, а решение данной проблемы должно быть комплексное, сохраняя все удобства компонентного подхода.

На текущее время уже существует реактивные фреймворки, где производительность рендеринга сравнима с нативным JavaScript. При этом они все равно имеют проблему медленной первой загрузки, особенно на мобильных устройствах.

Поэтому **целью данной диссертационной работы** является проверка гипотезы увеличения производительности с помощью частичной гидратации компонентов.

Гидратация (или регидратация) – это повторный рендеринг, который производиться на клиенте.

Основной недостаток гидрататации заключается в том, что она негативно влияет на время до интерактивности. Даже при улучшении первой отрисовки. Поэтому этап гидратации является наименее производительным во всех реактивных фреймворках на данный момент.

Частичная гидратация позволяет гидрировать только те компоненты, которые действительно нужно. Не тратить время и ресурсы на проработку всего DOM дерева. Данный подход является расширением идеи прогрессивной гидрататации, где отдельные части (компоненты/отображения/деревья), которые должны быть прогрессивно гидрированы, анализируются на предмет малой или отсутствующей интерактивности. Для этих в основном статических частей соответствующий код JavaScript затем преобразуется в «инертные» ссылки и декоративную функциональность, уменьшая отпечаток на стороне клиента почти до нуля.

Поэтому для того, чтобы проверить производительность частичной гидратации необходимо решить такие **задачи в данной работе**:

1. Исследовать принципы реактивности;
2. Изучить текущие применяемые технологические решения в популярных фреймворках;
3. Проанализировать и выявить методы для достижения лучшей производительности как динамических, так и частично-динамических веб-приложений;
4. Реализовать микрофреймворк, с функционалом сравнимым с тем, который используется в самых популярных реактивных фреймворках;
5. Произвести тестирование производительности на реальных примерах с помощью разработанного микрофреймворка.

При этом **практическая ценность** данной магистерской диссертации заключается в возможности использования результатов исследования в виде

готового JavaScript фреймворка для решения практических задач при проектировании и создании веб-сайтов и приложений.

Положения выносимые на защиту:

1. Частичная гидратация значительно уменьшает время, когда веб-приложение становится интерактивным;
2. Маленький размер и скорость работы микрофреймворка позволяет использовать его для создания мобильных веб-приложений (при медленном интернете и слабом процессоре);
3. Реализация микрофреймворка с частичной гидратацией не накладывает какие-либо ограничения на разработку и его использование.

1 Исследование применения реактивного программирования в веб-разработке

В данной главе производится общее описание задачи и проводится анализ методов решения этой задачи

1.1 Обзор решаемой задачи

1.1.1 Проблемы веб-разработки

Популярность JavaScript растет изо дня в день. Количество и качество технологий и фреймворков, которые построены на данном языке постоянно растет и улучшается open-source сообществом. Но при этом до сих пор существует множество проблем при проектировании и разработки веб-приложений (веб-сайтов).

Выделяют 5 основных таких проблем:

1. Сложная масштабируемость кода (архитектура)
2. Медленная скорость разработки и внедрения
3. Не использование принципов SOLID, что ведет к повторению кода
4. Отзывчивость UI (на стороне клиента)
5. Количество ресурсов, используемых приложением

Поэтому были разработаны реактивные фреймворки (Vue, AngularJS, React), которые отчасти решают данные проблемы и делают это по-своему. Данные фреймворки служат каркасом для отдельных страниц приложения, позволяют разработчикам меньше волноваться о структуре кода или поддержке в то время, когда они сосредоточены на создании сложных элементов интерфейса.

Преимущества использования JavaScript фреймворков:

1. Эффективность – проекты, которые раньше заняли бы месяцы и сотни строчек кода, сейчас могут быть реализованы намного быстрее с хорошо структурированными готовыми шаблонами и функциями.

2. Безопасность – лучшие JavaScript фреймворки имеют фирменную систему безопасности и поддерживаются крупным сообществом, члены которого и просто пользователи выступают в роли тестировщиков.
3. Расходы – большинство фреймворков с открытым кодом и бесплатны. Поскольку они помогают программистам быстрее разрабатывать пользовательские решения, итоговая цена веб приложения будет ниже.

Получая все преимущества фреймворков, нам приходится жертвовать производительностью и количеством потребляемых ресурсов. Для небольших веб-приложений это не критично, но если разрабатывается крупное приложение рассчитанное на использование миллионами людей, то каждый мегабайт ОЗУ и каждая миллисекунда затраченная на запуск и работу фреймворка крайне важна.

1.1.2 Реактивное программирование

Реактивному программированию – это парадигма программирования с асинхронными потоками данных. Асинхронность и поток – это две важные составляющие, на которых основывается реактивность.

Асинхронность – выполнение процесса в неблокирующем режиме, что позволяет программе продолжить обработку и не «зависать».

Асинхронный код убирает блокирующую операцию из основного потока программы, так что она продолжает выполняться, но где-то в другом месте, а обработчик может идти дальше. Проще говоря, главный «процесс» ставит задачу и передает ее другому независимому «процессу» (Рисунок 1).

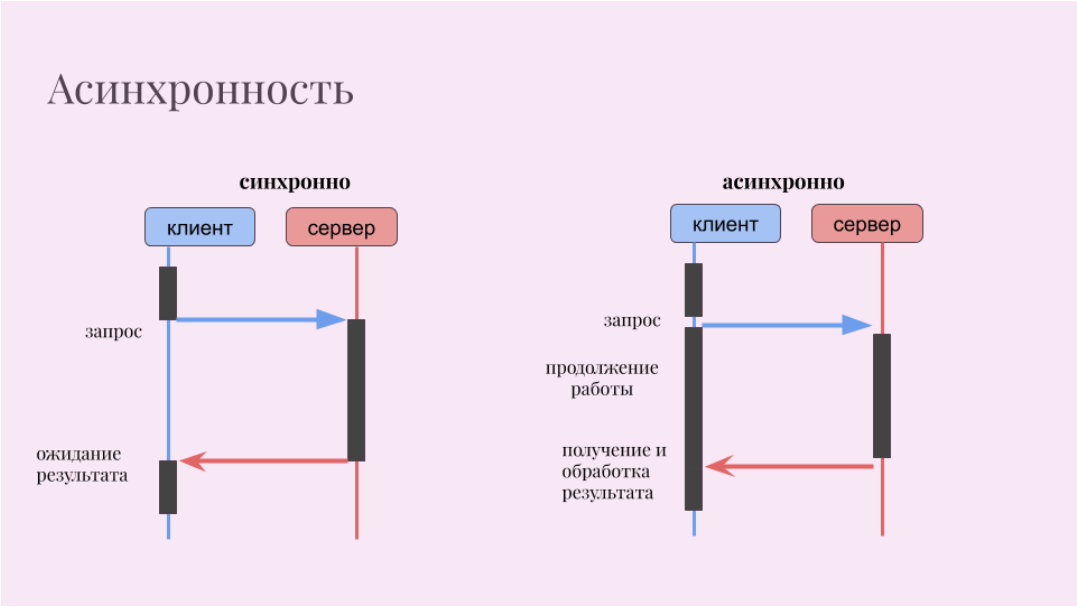

Рисунок 1 – Отличие синхронности от асинхронности

Потоки — это массив данных, отсортированных по времени, который может сообщать, что данные изменились.

Потоки могут транслировать и подписываться на данные. В течение жизненного цикла потоки могут транслировать три сигнала: данные, ошибку и завершение.

Все реактивные фреймворки построены на двух паттернов программирования [7]:

1. Observer (наблюдатель) – следит за изменениями данных и реагирует на них, чтобы представление было всегда актуальным;
2. Iterator (итератор) – подразумевает последовательный обход элементов контейнера и позволяет получать доступ к элементам контейнера без необходимости знать имплементацию контейнера (управление потоками и реализация методов подписки на изменение состояния).

Принято, что реактивность в веб перешла из мобильных приложений, чтобы реализовать компонентный подход, который может работать кроссплатформенно и независимо от версии и технологий браузера.

Первые реактивные фреймворки – Angular (Google) и ReactJS (Facebook). В дальнейшем сообщество разработчиков создали open-source решение: VueJS – это user-friendly фреймворк, с синтаксическим сахаром и «магическими» методами, которые производят предкомпиляцию компонентов, упрощая код, что улучшает его поддержку в дальнейшем.

1.1.3 Метрики производительности

Определим метрики, которые будут оценивать производительность и количество потребляемых ресурсов (Рисунок 2).

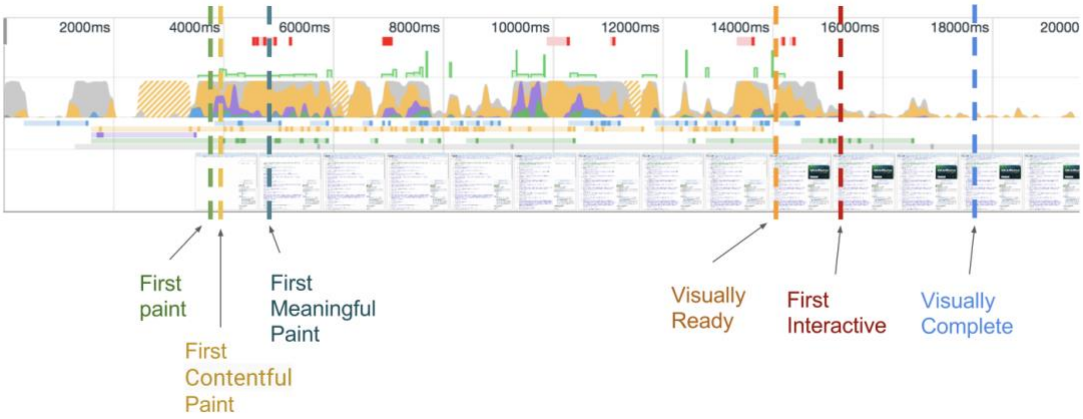

Рисунок 2 – Pipeline загрузки веб-страниц

1.1.3.1 Время до первого байта (Time to First Byte)

Время между кликом по ссылке и поступлением первого фрагмента содержимого на клиентскую часть, обычно это веб-браузер.

1.1.3.2 Первая отрисовка (First Paint)

Момент, когда страница только начала отрисовываться, то есть это время, когда пользователь видит пустую страницу впервые. Скрипты еще не загрузились и рендеринг на стороне клиента еще не произошел.

1.1.3.3 Первая отрисовка контента (First Contentful Paint)

Это время, когда пользователь видит что-то полезное, отрисованное на странице. То, что отличается от пустой страницы. Это может быть всё, что угодно: первая отрисовка текста, первая отрисовка SVG или первая

отрисовка Canvas. Если первая отрисовка контента занимает очень много времени то:

- Возможно, есть проблемы на уровне соединения;
- Ресурсы (например, сама HTML-страница) очень тяжелые и чтобы их доставить нужно время.

1.1.3.4 Первая значимая отрисовка (First Meaningful Paint)

Это время, когда весь главный контент отрендерился и появился на странице. Для каждой ситуации он разный, но обычно главным контентом считается:

- Шапка и текст блога;
- Содержимое для поисковиков;
- Критичные для интернет-магазинов картинки.

1.1.3.5 Первое взаимодействие (First Interactive)

Это время, когда веб-приложение становится интерактивным. Начинает реагировать на пользовательские события и обрабатывать информацию на стороне клиента. Первое взаимодействие считается случившимся, если удовлетворены все условия:

- Произошла первая значимая отрисовка;
- Сработал DOMContentLoaded;
- Страница визуально готова на 85%.

Метрика первого взаимодействия разделена на две метрики:

- Время до первого взаимодействия (TTFI);
- Время до первого последовательного взаимодействия (TTCI).

1.1.3.6 Первое последовательное взаимодействие (Time to First Consistently Interactive)

Используя реверсивный анализ, который подразумевает анализ трейса с конца, находится период, когда загрузка ресурсов неактивна на протяжении 5 секунд и в этот период отсутствуют длинные задачи.

Такой период называется тихое окно (quiet window). Время после тихого окна и перед первой, с конца, длинной задачей будет временем до первого последовательного взаимодействия.

1.1.3.7 Время до первого взаимодействия (Time to First Interactive)

Определение этой метрики отличается от первого последовательного взаимодействия. Трейс анализируется со старта и до конца. После того, как произошла первая значимая отрисовка, находят тихое окно в 3 секунды. Этого достаточно, чтобы сказать, что страница интерактивна. Но в трейсе могут присутствовать одинокие задачи во время или после тихого окна.

Одинокие задачи выполняются далеко после первой значимой отрисовки и изолированы периодом выполнения в 250 мс (envelope size) и одну секунду тишины до и после этого периода. Иногда одинокие задачи, время которых занимает больше 250 мс, могут сильно влиять на быстродействие страницы.

1.1.3.8 Предполагаемая задержка ввода (Estimated Input Latency)

Данная метрика оценивает время, которое требуется приложению для ответа на ввод пользователя в течение самого загруженного 5-секундного окна загрузки страницы. Временем этого аудита является участок от первой значимой отрисовки до конца трейса, что составляет примерно 5 секунд после времени до интерактивности. Если задержка превышает 50 мс, пользователи могут воспринимать веб-приложение, как слишком медленное.

1.1.4 Рассматриваемые этапы работы веб-приложений

Следует разделять запуск веб-приложения по этапам, чтобы исследовать каждый их них и найти наилучшее решение конкретно для каждого этапа.

Так как в данной работе поднимается вопрос производительности frontend части веб-приложений, то и рассматривать этапы работы этих приложений мы будем в контексте frontend’а (без затрагивания backend и API веб-приложений).

Также следует оговорить, что мы рассматриваем производительность динамических веб-ресурсов, то есть ресурсов, данные которых могут меняться и после или во время загрузки веб-страницы. Статический рендеринг рассмотрен не будет, так как является сильным упрощением серверного рендеринга и для большинства приложений, например где имеется авторизация, он не подходит.

На данный момент для достижения хорошей производительности выделяют такие основные этапы работы реактивных приложений:

1. Server-side rendering – первичный рендеринг приложения на стороне сервера. Он позволяет собрать DOM дерево и отправить на клиент;
2. Загрузка данных (фреймворк, настроенные компоненты и т.д.);
3. Парсинг JavaScript-скриптов и их исполнение;
4. Гидратация компонентов – совмещение виртуального DOM дерева с реальными, который был получен на первом этапе SSR, для достижение интерактивности;
5. Обновление данных и внесение изменений в DOM.

К сожалению, не на все этапы мы можем повлиять прямо, но существуют различные методы, технологии и принципы, которые позволят достичь максимальной производительности на каждом из них.

1.1.4.1 Рендеринг на стороне сервера (SSR)

SSR (Server-Side Rendering, серверный рендеринг) — рендеринг на сервере клиентской части или универсального приложения в HTML;

Первичное построение DOM дерева веб-приложений – одна из самых затратных операций, поэтому ее необходимо выносить в «облако», чтобы расчеты и создание DOM происходило на стороне сервера и клиент мог воспользоваться приложением даже на низкопроизводительных и устаревших устройствах.

К тому же, серверный рендеринг исключает необходимость в дополнительных запросах данных со стороны клиента, так как сервер берёт всю работу на себя, прежде чем отправить ответ.

Такой подход позволяет добиться быстрой первой отрисовки и первой содержательной отрисовки [8]. Выполнение логики страницы и рендеринг на сервере позволяют избежать отправки клиенту большого количества JavaScript, что приводит к меньшему времени до интерактивности. Этот подход хорошо сработает на широком диапазоне устройств и сетевых условий, а также откроет возможности для интересных браузерных оптимизаций вроде потокового парсинга документа.

Оптимизации работы серверной части давно известны и применяются по всех языках программирования:

1. Оптимизация алгоритмов;
2. Кэширование.

Оптимизация алгоритмов будет затронута в последующих этапах. Кэширование же затронуто не будет, так как не является предметной областью изучения в данной работе.

1.1.4.2 Загрузка сторонних ресурсов

Под загрузкой данных подразумевается:

1. Получение основной HTML-страницы;

2. Получение дополнительных ресурсов, необходимых для работы веб-приложения (css, js, шрифты).

В данной работе будут рассмотрены методы ускорения загрузки ресурсов, которые:

– Можно применять во всех браузерах;
– Не зависят от соединения;
– Не зависят от сервера.

1.1.4.3 Парсинг JavaScript-скриптов и их исполнение

Каждый веб-браузер работает на своем JavaScript-движке и повлиять на производительность парсинг и исполнение скриптов невозможно и не нужно, ведь они достаточно оптимизированы и поддерживаются крупными корпорациями.

Единственное, что необходимо учитывать: чем больше размер скрипта, тем дольше он будет парситься и исполняться, поэтому важно сохранять минимальный размер JavaScript-ресурсов.

1.1.4.4 Гидратация компонентов

Гидратация (или регидратация) – это повторный рендеринг, который производиться на клиенте [8]. Это новое решение, тем не менее не лишённое определённых проблем с производительностью.

Основной недостаток серверного рендеринга (SSR) с гидрататацией заключается в том, что такой негативно влияет на время до интерактивности. Даже при улучшении первой отрисовки. Поэтому этап гидратации является наименее производительным во всех реактивных фреймворках на данный момент. Он и будет основным этапом, который будет оптимизироваться в данной работе.

1.1.4.5 Обновление данных и внесение изменений в DOM

Так как все реактивные фреймворки работают с DOM деревом и производят работу по-разному, необходимо рассмотреть все основные способы в контексте метрик производительности, которые нам важны:

- Время добавления 1000 элементов (Node) на страницу;
- Время замены 1000 элементов на старинце новыми элементами;
- Время перестановки пары элементов (1000 элементов);
- Время запуска фреймворка;
- Время интерактивности фреймворка;
- Память после загрузки страницы;
- Память после обработки 1000 элементов;
- Память после добавления 1000 элементов и их очистки.

1.2 Анализ методов решения задачи

В данной части работы будут изучены и проанализированы все основные подходы, которые используются как отдельно, так и в совокупности с реактивными фреймворками для обеспечения более быстрой загрузки и обработки данных.

1.2.1 Использование сторонних ресурсов в веб-браузере

Веб-приложения и сайты устроены так, что они используют множество сторонних ресурсов. Кроме основного HTML документа, веб-браузер также работает с изображениями, CSS и JavaScript ресурсами, сторонними шрифтами и многими другими медиа-ресурсами.

Существует несколько технологий, которые позволяют ускорить загрузку и работу данных ресурсов.

1.2.1.1 Протокол HTTP/2

Данный протокол существенно ускоряет открытие сайтов за счет следующих особенностей:

- Соединения: несколько запросов могут быть отправлены через одно TCP-соединение, и ответы могут быть получены в любом порядке. Отпадает необходимость держать несколько TCP-соединений;
- Приоритеты потоков: клиент может задавать серверу приоритеты — какого типа ресурсы для него более важны, чем другие;
- Сжатие заголовка: размер заголовка HTTP может быть сокращен.

Так как несколько запросов могут быть отправлены через одно и тоже соединение, то имеет смысл разбивать ресурсы на несколько частей. Пока одна небольшая часть всего JavaScript ресурса парсится и исполняется, другие части продолжают загружаться. К тому же, группировка частей по использованности их на страницах позволит браузеры кэшировать часто используемые ресурсы автоматически.

1.2.1.2 Отложенное и асинхронное исполнение JavaScript ресурсов

JavaScript является блокирующим ресурсом для парсера. Это означает, что JavaScript блокирует разбор самого HTML-документа. Когда парсер доходит до тега <script> (не важно внутренний он или внешний), он останавливается, забирает файл (если он внешний) и запускает его.

Такое поведение может доставить проблемы, если мы загружаем несколько JavaScript-файлов на странице, так как это увеличивает время первой отрисовки, даже если документ на самом деле не зависит от этих файлов.

Асинхронное исполнение (async) – указывает браузеру, что скрипт может быть выполнен асинхронно. Парсеру HTML нет необходимости останавливаться, когда он достигает тега <script> для загрузки и выполнении.

Выполнение может произойти после того, как скрипт будет получен параллельно с разбором документа.

Отложенное исполнение (defer) – указывает браузеру, что скрипт должен быть выполнен после того, как HTML-документ будет полностью разобран.

1.2.1.3 Предзагрузка (preload)

Предзагрузка позволяет браузеру как можно скорее загрузить и кэшировать ресурс (например, скрипт или таблицу стилей). Это полезно, когда ресурс понадобится через несколько секунд после загрузки страницы — и вы хотите ускорить процесс.

1.2.1.4 Предварительное соединение (preconnect)

Предварительное соединение указывает браузеру, к каким доменам нужно подключиться заранее, чтобы ускорить установку соединения в будущем.

1.2.1.5 Предварительное получение записей DNS (dns-prefetch)

Предварительное получение записей DNS, позволяет браузеру заранее выполнить DNS-resolving (получение DNS записей) для домена, чтобы ускорить начальное соединение с ресурсом, который будет имет доступ через домен.

1.2.1.6 Ленивая загрузка ресурсов (lazy-loading)

Паттерн Lazy-loading (Ленивая Загрузка) подразумевает отказ от загрузки дополнительных данных, когда в этом нет необходимости. Вместо этого ставится маркер о том, что данные не загружены и их надо загрузить в случае, если они понадобятся. Например, в самом низу веб-страницы находится изображения, их нет смысла загружать сразу, потому что пользователь находится в самом верху страницы. Но когда он спустится в низ, они должны подгрузиться.

1.2.2 Реактивность DOM

Если посмотреть на реактивные фреймворки глобально, то существует всего два разных микроподхода (fine-grained) рендеринга DOM:

1. Фреймворк используют существующую систему компонентов, где функция рендера является вычисляемой (то есть реагирует на изменения состояния компонента с помощью паттерна Observable) и с помощью patch/diff применяет изменения к DOM;
2. Либо компоненты связываются непосредственно напрямую с системой привязки DOM и все обновления дерева выполняются нативно – с помощью VanilaJS.

1.2.2.1 Вычисляемая рендер-функция

Данный способ используют все популярные фреймворки: React, Angular, Vue.

Во-первых из-за простоты разработки, когда все данные компоненты – реактивны.

Во-вторых из-за простоты подхода, когда изменения в DOM вносятся с помощью поиска разницы (diff) виртуального DOM дерева, которое было вычислено фреймворком, и реального DOM. И применения (patch) этой разницы в дальнейшем.

1.2.2.2 Система привязки DOM

Данные подход является новым в реактивной разработке. Команда, которая разрабатывает Vue, уже более года работает нам внедрением данной системы рендеринга в фреймворк. Но уже сейчас существует множество микро-фреймворков, которые позволяют протестировать данный подход и получить базовые метрики производительности.

Плюс данного подхода в том, что он использует нативные функции и методы JavaSript и DOM, что позитивно влияет на производительность. Она становится в один ряд с VanilaJS.

1.2.3 Гидратация компонентов

Так как гидратация – это достаточно тяжелая операция и сильно влияет на время до интерактивности, то уже существует несколько подходов в ее реализации. Некоторые подходы уже внедрены во фреймворки, но проблема интерактивности до сих пор стоит на первом месте.

1.2.3.1 Полная гидратация

После того, как произошел серверный рендеринг и HTML документ был отправлен на клиент, туда же отправляется JavaScript с настройками компонентов и шаблонами. После получения всех нужных ресурсов фреймворк производит полное сравнение виртуального DOM и реального (patch/diff).

Из-за этого сильно страдает производительность: при большом количестве используемых компонентов на веб-странице необходимо обработать каждый узел DOM, каждое его свойство и даже текст, ведь он тоже является узлом.

Проблема в том, что веб-страница – это не веб-приложение, где каждый элементов динамичен, хотя и там доля динамических компонентов далеко не 100%. Поэтому нет необходимости сравнивать каждый узел реального DOM с виртуальным.

Положительным моментом является то, что фреймворки с fine-grained подходом работают гораздо быстрее, и их интерактивность почти такая, как и в нативном варианте.

1.2.3.2 Прогрессивная (потоковая) гидратация

Прогрессивная гидратация – это использование потокового серверного рендеринга в совокупности с гидратацией.

Потоковый серверный рендеринг позволяет посылать HTML фрагментами, которые браузер может постепенно рендерит по мере получения.

При таком подходе отдельные части сформированного сервером приложения загружаются постепенно вместо единоразовой инициализации всего приложения, как это делается сейчас. Это может уменьшить количество JavaScript, необходимого для интерактивности страниц, так как клиентское обновление низкоприоритетных частей страницы можно отложить, чтобы предотвратить блокировку основного потока.

Данный способ имеет право на жизнь, но только на очень длинных страницах, например, лента новостей в соц. сетях (хоть это и не потоковая гидратация, но смысл один и тот же). Проблема статичных компонентов также не решается.

1.2.3.3 Частичная гидратация

Реактивные фреймворки были разработаны для создания полноценных веб-приложений. А как известно все компоненты и элементы приложения – динамические (Facebook, Google поиск). При этом возникает вопрос, как разрабатывать веб-приложения с меньшей динамической составляющей, например: веб-сайты, интернет-магазины, сайты для услуг (доставка питания, парикмахерские, СТО).

Именно для этого и создана частичная гидратация, которая на данный момент не реализована ни в одном из фреймворков. Она позволяет гидрировать только те компоненты, которые действительно нужно. Не тратить время и ресурсы на проработку всего DOM дерева.

Частичную гидрататацию сложно реализовать. Данный подход является расширением идеи прогрессивной гидрататации, где отдельные части (компоненты/отображения/деревья), которые должны быть прогрессивно гидрированы, анализируются на предмет малой или отсутствующей интерактивности. Для этих в основном статических частей соответствующий код JavaScript затем преобразуется в «инертные» ссылки и декоративную функциональность, уменьшая отпечаток на стороне клиента почти до нуля.

1.2.3.4 Ленивая гидратация

Также, как паттерн Lazy-loading (Ленивая Загрузка) подразумевает отказ от загрузки дополнительных данных, когда в этом нет необходимости, так и ленивая гидратация подразумевает отказ от процесса гидратации, когда в этом нет необходимости.

Компоненты должны становится динамическими при некоторых условиях:

1. Когда они расположены в видимой пользователем области веб-сайта;
2. Когда пользователь начал взаимодействовать с компонентом;
3. Или когда процессор достаточно освобожден для нагруженных вычислений.

Ленивая гидратация значительно улучшает такие метрики, как «время до интерактивности» и «предполагаемая задержка ввода», что видно из отчетов работы Google Lighthouse страницы (рисунки 3 и 4) – сервис с открытым кодом представляет собой инструмент для оценки качества сайтов, который работает в автоматическом режиме. Он может провести анализ производительности и простой аудит UX.

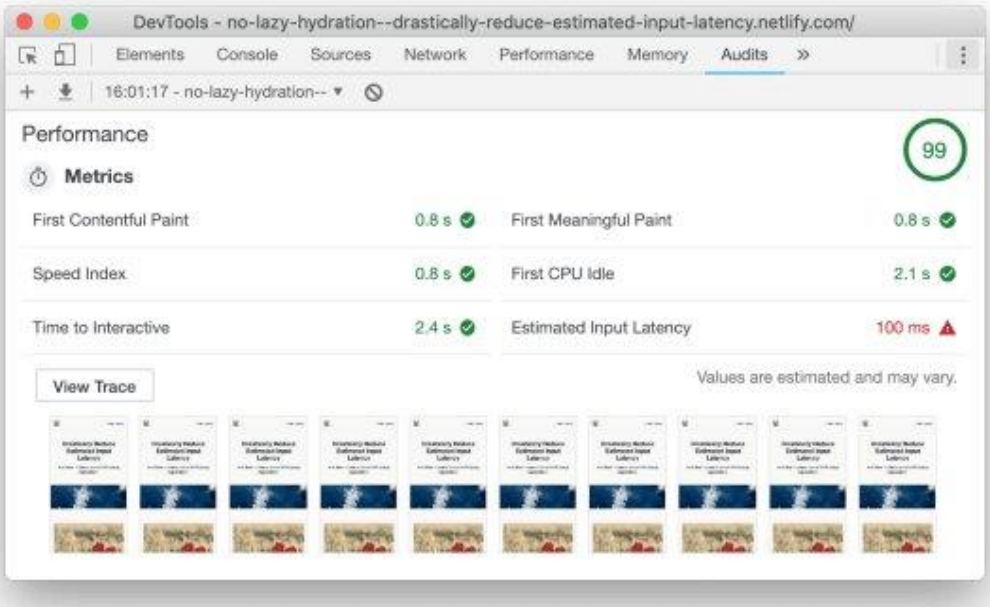

Рисунок 3 – Пример отчета lighthouse без использования ленивой гидратации

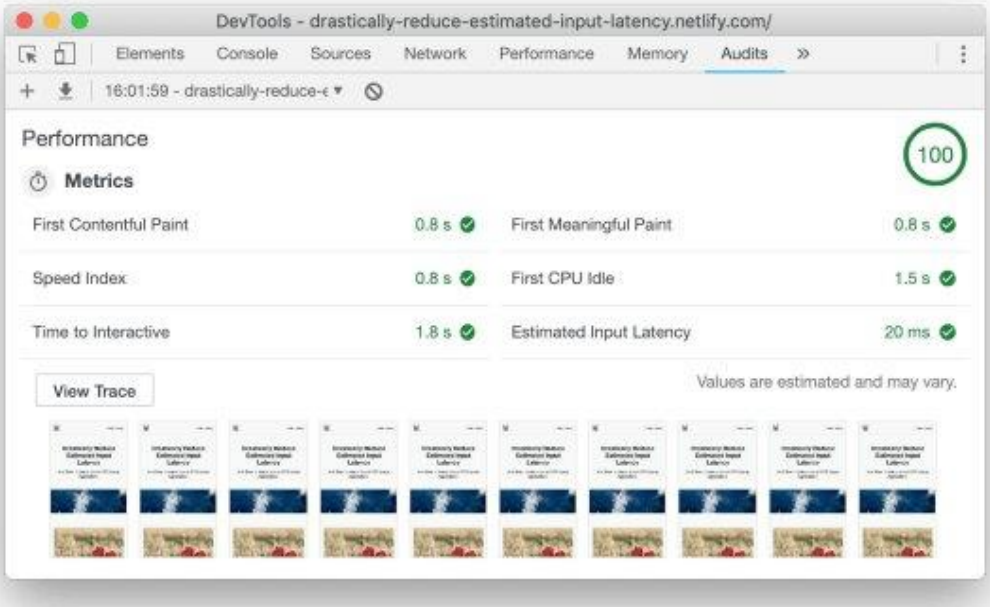

Рисунок 4 – Пример отчета lighthouse с использованием ленивой гидратации

1.3 Постановка задачи на исследование и разработку

Как видно из исследования работы реактивных фреймворков, существует множество методов и способов реализовать эту реактивность. При этом существует множество проблем с производительностью, которые проявляются как в динамических приложениях, так и в частично-динамических, где также присутствуют и статические компоненты.

В силу того, что данная парадигма приносит значительное удешевление и ускорение веб-разработки, необходимо:

1. Исследовать принципы реактивности;
2. Изучить текущие применяемые технологические решения в популярных фреймворках;
3. Проанализировать и выявить методы для достижения лучшей производительности как динамических, так и частично-динамических веб-приложений;
4. Реализовать микрофреймворк, с функционалом сравнимым с тем, который используется в самых популярных реактивных фреймворках;
5. Произвести тестирование производительности на реальных примерах с помощью разработанного микрофреймворка.

Цель данной работы – это проверить гипотезу увеличения производительности с помощью частичной гидратации компонентов.

1.4 Выводы по главе

Реактивный программирование – это действительно инновационная парадигма, особенно в мире веб-технологий, так как позволяет тратить меньше времени на разработку, и при этом получать более тестируемый код, компонентный подход и позволяет разрабатывать пользовательские интерфейсы с меньшим риском ошибок.

Существует множество решений и способов ускорить различные этапы работы веб-приложений. Но у всех способов есть недостаток: они не рассчитаны на создание частично-динамических приложений, который

гораздо больше, чем полностью динамических. Поэтому вопросы гидрирования и работы с DOM деревом остаются открытыми до сих пор.

В данной главе были исследованы и выявлены основные принципы и способы работы с DOM деревом и его гидрированию, которые могут поспособствовать реализации наиболее производительных частично-динамических веб-приложений и сайтов.

2 Разработка реактивного микрофреймворка

2.1 Анализ и выбор способа решения задачи

2.1.1 Реактивность и DOM

Исходя из исследований пункта 1.2.2, существует два подхода, которые используются в реактивных фреймворках для связывания данных и виртуального дерева элементов с DOM:

1. Вычисляемая рендер-функция;
2. Система привязки DOM.

Оба подхода уже реализованы частью фреймворков и уже существуют замеры производительности основных метрик, которые представлены в таблицах 2-7.

На текущий момент самые популярные фреймворки с наибольшей поддержкой open-source сообществом, которые используют «вычисляемую рендер-функцую», являются: Vue, React и Angular. Именно они и будут участвовать в анализе.

Со стороны фреймворков, которые используют «систему привязки DOM», не существует решений, который удовлетворяли бы потребности рынка, поэтому будут отобраны самые производительные из них. Это Sinuous, Solid и Surplus.

Тестирование и анализ метрик из пункта 1.1.4.5 производится на Razer Blade 15 Advanced, характеристики ЭВМ представлены в таблице ниже.

Таблица 1 – Характеристики ЭВМ – Razer Blade 15 Advanced

Процессор	i7–8750H
RAM	32 GB
ОС	Ubuntu 20.04 (Linux 5.4.0-21, mitigations=off)
Веб-браузер	Chrome 81.0.4044.113 (64-bit) V8 – JS Engine

2.1.1.1 Вычисляемая рендер-функция

Сравнение производительности React, Angular и Vue приведено относительно VanilaJS в соответствии с метриками, указанными в пункте 1.1.4.5, и отображено в таблицах ниже [9].

Таблица 2 – Таблица сравнения времени взаимодействия с 1000 элементов (Node) на странице

Фреймворк	VanilaJS	Vue	React	Angular
Добавление (мс)	119,3	167,0	170,0	167,3
Замены (мс)	114,3	139,1	141,0	144,1
Перестановка (мс)	53,0	68,5	461,2	464,7

Таблица 3 – Таблица сравнения времени запуска и интерактивности фреймворков

Фреймворк	VanilaJS	Vue	React	Angular
Запуск (мс)	23,1	62,4	99,1	194,4
Интерактивность (мс)	1884,4	2313,1	2529,8	2879,7
Работа в главном потоке (мс)	143,2	210,9	260,6	302,1

Таблица 4 – Таблица сравнения используемой фреймворками

Фреймворк	VanilaJS	Vue	React	Angular
После загрузки страницы (МБ)	1,1	1,2	1,3	2,7

Продолжение таблицы 4

После обработки 1000 элементов (МБ)	1,6	4,0	3,9	5,1
После добавления 1000 элементов и их очистки (МБ)	2,3	2,6	3,2	4,3

Исходя из данных таблиц, наилучшем образом, в сравнение с VanilaJS, работает Vue. Производительность выше, а используемая память меньше, если сравнивать React и Angular.

2.1.1.2 Система привязки DOM

Сравнение производительности Sinuous, Solid и Surplus приведено относительно VanilaJS в соответствии с метриками, указанными в пункте 1.1.4.5, и отображено в таблицах ниже [9].

Таблица 5 – Таблица сравнения времени взаимодействия с 1000 элементов (Node) на странице

Фреймворк	VanilaJS	Sinuous	Solid	Surplus
Добавление (мс)	119,3	122,3	124,7	125,6
Замены (мс)	114,3	121,4	119,5	123,4
Перестановка (мс)	53,0	53,7	53,5	56,3

Таблица 6 – Таблица сравнения времени запуска и интерактивности фреймворков

Фреймворк	VanilaJS	Sinuous	Solid	Surplus
Запуск (мс)	23,1	21,2	22,4	22,6
Интерактивность (мс)	1884,4	1882,2	1884,2	1959,2
Работа в главном потоке (мс)	143,2	147,2	149,1	150,7

Таблица 7 – Таблица сравнения используемой фреймворками

Фреймворк	VanilaJS	Sinuous	Solid	Surplus
После загрузки страницы (МБ)	1,1	1,1	1,1	1,1
После обработки 1000 элементов (МБ)	1,6	2,3	2,1	2,2
После добавления 1000 элементов и их очистки (МБ)	2,3	2,4	2,3	2,5

Исходя из данных таблиц, все фреймворки работают почти одинаково по производительности. Sinuous и Solid делят первое место.

2.1.2 Микрофреймворк

Так как уже существуют фреймворки, то нет необходимости разрабатывать функционал, который они предоставляют и который нам требуется. Поэтому имеет смысл за основу, которая будет доработана до наших требований, взять существующий фреймворк.

Как видно, ключевые метрики производительности «системы привязки DOM» гораздо лучше, чем у «вычисляемой рендер-функция», и схожи с тем, будто фреймворк не используется, хотя все его возможности и плюсы сохраняются. Именно поэтому «системы привязки DOM» и будет выбрана для реализации работы с DOM.

Необходимо проанализировать фреймворки и определить, какой соответствует требованиям, необходимым для реализации решения:

1. Возможность использовать свой алгоритм гидратации;
2. Рендеринг DOM должен использоваться с помощью «системы привязки DOM»;
3. Возможность использовать свой синтаксис для определения DOM и логики его построения (и гидратации);

4. Возможность определения и отличия реактивных данных от статичных данных (для реализации гидратации для частично-динамических веб-приложений).

Если ни один из фреймворков нельзя использовать как базовый, рассмотреть возможность создания своего реактивного фреймворка.

2.1.2.1 Sinuous

Sinuous – это реактивный фреймворк, который использует аналогичную систему реактивности, как в «S.js», что предоставляет такие возможности, как:

1. Автоматические обновления – когда данные изменяются, фреймворк автоматически изменяет вычисляемые свойства, которые использует эти данные (реактивность);
2. Последовательный timeline – позволяет с помощью дискретных «тиков» гарантировать, что автоматическое обновление данных произойдет своевременно и не будет изменено, пока «тик» не закончится;
3. Транзакции – множество сигналов об изменении данных преобразуются в транзакции;
4. Автоматическое удаление вычисляемых и не используемых данных.

Sinuous не использует изоляцию контекстов для построения компонентов и соблюдения компонентного подхода, но при этом поддерживает три способа записи/создания DOM элементов:

1. Tagged template – собственный компилятор HTML когда в набор DOM элементов;
2. JSX – это препроцессор, который добавляет синтаксис XML к JavaScript;
3. Hyperscript – это обертка на обычном DOM, который обычный JavaScript код превращает в элементы DOM.

Фреймворк позволяет использовать Hyperscript – это означает, что имеется возможность создать свой компилятор (в дальнейшем) для преобразования код в Hyperscript, а оттуда в инструкции DOM. Что позволяет в момент компиляции понимать, какие части получившегося DOM являются реактивными, для создания частичной гидратации в дальнешейм.

Также данные фреймворк позволяет использовать свой алгоритм гидратации, но данный функционал (api) не является документированным.

2.1.2.2 Solid

Solid позиционирует себя, как JavaScript библиотека, с декларативным подходом для разработки пользовательских интерфейсов. Не использует виртуальный DOM и позволяет работать с данными по модели реактивности.

В отличии от Sinuous, данная библиотека поддерживает только JSX и сама компилирует его в инструкции DOM. Что не позволит генерировать свой Hyperscript код и соответственно использовать функции определения реактивности данных для реализации частичной гидратации.

2.1.2.3 Выбор технологии

Таблица 8 – Таблица сравнения Sinuous и Solid

Функционал	Sinuous	Solid
Рендеринг	Система привязки DOM	Система привязки DOM
Синтаксис	Tagged Template, Hyperscript, JSX	JSX

Так как Sinuous имеет возможность использовать HyperScript, то для реализации своего микрофреймворка будет выбран, как основным.

На его основе будут разработаны модули, которые расширят базовые возможности Sinuous, добавив функционал наиболее популярных фреймворков, а также модуль частичной гидратации для решения задач производительности.

2.1.3 Парадигма программирования

Как было указано в пункте 1.1.1, существует 5 глобальных проблем при разработке веб-приложений. Некоторые уже решаются с помощью существующих фреймворков, но в виду того, что данные фреймворки используют низкопроизводительные технологии, необходимо реализовать свое решение данных проблем, используя Sinuous.

Проблемы которые нужно решить:

1. Возможность переиспользования кода;
2. Поддержка agile-подхода;
3. Обеспечить отзывчивость UI, с помощью обеспечения согласованности взаимодействий пользователя с UI во всем приложении;
4. Простая интеграция в процесс веб-разработки;
5. Оптимизация требований и процесса проектирования;
6. Медленный переход от стадии создания дизайна приложения к его разработке.

Для решения таких проблем идеально подходит компонентно-ориентированный подход [1], который используется во всех популярных реактивных фреймворках.

Поэтому требуется разработать свое решение, используя компонентный подход.

Для этого необходимо использовать ограниченные контексты исполнения кода. То есть каждый участок кода (список, поле ввода и другие более сложные компоненты) должен работать изолировано в своем контексте, что и будет определять компонент – изолированный и переиспользуемый участок кода.

Гидратация компонентов также должна происходить, учитывая контекст исполнения кода и придерживаться компонентно-ориентированному подходу.

2.1.4 Гидратация

Требуемое решение должно быть универсальным и подходить как для динамических, так и статических веб-приложений и сайтов. При этом узким местом в производительности гидратации является то, что в текущих решениях данный процесс происходит независимо от того, является ли элемент динамическим и следует ли его гидрировать сейчас, ведь он может не находиться в видимой пользователю области веб-страницы.

Исходя из этого необходимо использовать частичную гидратацию. При использовании фреймворка и создании компонентов программистом, не должно специально указываться, какие части компонента реактивные, а какие статичные. Иначе это усложнит разработку и тестирование пользовательских интерфейсов.

Частичная гидратация может быть совмещена с потоковым способом, для достижения лучшей производительности, но только, если этот способ будет использовать при линейной, а не вложенной структуре компонентов. То есть потоковая гидратация совместно с частичной может использоваться только тогда, когда набор данных является итерационным.

Также необходимо реализовать совмещения с ленивой гидратацией, что улучшит показатели для длинных веб-страниц или приложений с большим количеством компонентов, с которыми будет взаимодействовать пользователь только при определенных условиях. Эти условия могут быть обозначены программистом при разработке компонентов интерфейса.

2.1.5 Серверный рендеринг

Способы оптимизации серверного рендеринга не будут рассмотрены в данной работе.

Но разрабатываемое решение должно подразумевать корректную работу рендеринга на стороне сервера или ее возможность в дальнейшем.

Также важно, что исследованные способы ускорения работы веб-приложений применяются на стороне сервера, поэтому разработаны не будут, так как самой реализации серверного рендеринга нет. Важно учитывать, что «ленивая загрузка» реализуется на стороне клиента, поэтому она может быть разработана, как расширение функционала фреймворка в дальнейшем.

2.2 Разработка концепции компонентов, их моделей и алгоритмов решения задачи

2.2.1 Компоненты

Для разработки компонентов следует перенять опыт популярных фреймворков (Vue) и реализовать схожий функционал. Поэтому компоненты должны обладать таким набором функционала:

- Иметь локальные переменные для реализации бизнес-логики компонента;
- Поддерживать вычисляемые свойства;
- Поддерживать передачу данных между компонентами для реализации их связывания.

При этом для того, чтобы реализовать частичную гидратацию, необходимо отделять реактивные данные от статичных, что следует учесть при создании локальных переменных. Поэтому компонент должен иметь такие наборы данных:

- Локальные переменные без зависимостей, то есть переменные, которые используются только для хранения информации (storage);
- Входные параметры, которую задаются при инициализации компонента в шаблоне рендеринга;
- Локальные переменные с зависимостями – реактивные данные.

2.2.2 Реактивные шаблоны компонентов

Для реализации шаблонов необходимо использовать Hyperscript, учитывая наши особенности:

- Компонентный подход;
- Поддерживать слоты – передаваемые HTML шаблоны в дочерние компоненты;
- Поддерживать условный рендеринг;
- Поддерживать итерационный рендеринг.

Необходимо учитывать, что в дальнейшем будет использовать свой шаблонизатор, которой ретранслирует шаблон в модифицированный hyperscript.

2.2.3 Реактивные данные компонентов

В Sinuous, который мы выбрали, как основной фреймворк, который реализует реактивность, используется 3 метода для реализации основного функционала:

1. Observable – возвращает функцию, которая может быть вызвана для получения значения реактивных данных. Если в функцию передан аргумент, то данные изменятся и произойдет вызов обновления всех связанных данных.
2. Computed – возвращает функцию, которая будет обновлена, если observable, который используется в переданной функции, будет изменен;
3. Subscribe – метод используется для того, чтобы совершать какие-либо действия при изменении observable, который используется в переданной функции.

Для того, чтобы отслеживать являются ли данные реактивные, необходимо, модифицировать функции Observable и Computed, чтобы они соответствовали одному интерфейсу. Так как в JavaScript нет интерфейсов,

то для данных функций можно добавить свойство в прототип данных функций или в их модификацию. Например, «_observable».

2.2.4 Динамические атрибуты DOM элементов

Sinuous на уровне фреймворка реализует реактивность атрибутов DOM элементов, если использованы реактивные данные, объявленные через observable или computed.

2.2.5 Динамические свойства (входные параметры) компонентов

Sinuous не использует компонентный подход и обособленные контексты исполнения кода, поэтому необходимо реализовать возможность передачи статических и реактивных входных параметров, а также свойств DOM элементов:

- CSS классы
- CSS-inline стили

Которые будут дополнять те, которые уже определены внутри дочернего компонента.

2.2.6 Условный рендеринг

В веб-приложениях и сайтах существует необходимость скрывать некоторые участки интерфейса и инициализировать компонент, исходя из некоторых условий. Например, при реализации выпадающего списка необходимо скрывать или показывать данный список, если был произведен клик на кнопку.

Для реализации условного рендеринга необходимо использовать алгоритм, описанный в приложении 1.

2.2.7 Итерационный рендеринг

Для разработки повторяющихся элементов, например, списков или каталога товаров, необходимо реализовать итерационный рендеринг, которые при этом позволят использовать реактивность.

Необходимо учитывать, чтобы алгоритм гидратации и рендеринга понимал, какой элемент виртуального DOM соотноситься с реальным. Для этого следует использовать вычисляемые ключи элементов списка.

Итерационный рендеринг используется в Sinuous, но при этом без вычисляемых ключей, которые необходимы для гидратации.

2.2.8 Частичная гидратация

При частичной гидратации пропускаются статичные части компонентов и первичный рендеринг.

Поэтому при разработке фреймворка необходимо маркировать переменные состояния, что позволит определять динамичность участок на лету.

2.2.9 Ленивая гидратация

Ленивая гидратация используется для длинных веб-страниц, это означает, что есть смысл производить гидратацию только тогда, когда конкретный участок страницы появился в видимой пользователем области.

Поэтому следует рассмотреть возможность создания специальных компонентов или методов, которые будут определять тип гидратация конкретного компонента, а также вложенных в него элементов и дочерних компонентов.

2.2.10 Сериализация и инициализация гидратации

Чтобы частичная гидратация была максимальна производительна необходимо определить формат, в котором экспортировать и импортировать дерево компонентов и начальный контекст (входные параметры компонентов).

Так как наилучший способ – это использовать нативный JavaScript и не использовать прослойку в виде парсеров и ретрансляторов, то необходимо реализовать метод сборки, который будет использоваться на этапе SSR и создавать JavaScript скрипт с набором инструкций для гидратации.

2.3 Выводы по главе

В данной главе были выявлены средства и технологии, которые требуется использовать, чтобы добиться наилучшей производительности.

Также вместо того, чтобы реализовывать свою реактивность, был выбран фреймворк Sinuous, который является наиболее быстрым (исходя их требуемых метрик) и предоставляет максимальные возможности индивидуализации, которые необходимо для реализации полного функционала.

При этом для того, чтобы дальнейшее применение и сравнение разрабатываемого фреймворка имело смысл, необходимо реализовать тот же набор функций, который есть в существующих реактивных фреймворках, занимающих лидирующие позиции.

3 Реализация микрофреймворка

3.1 Выбор технологий и средств реализации микрофреймворка

В виду того, что веб-технологии – это кроссплатформенные технологии, среда исполнения которых – это веб-браузер, то более важно при выборе технологий и средств реализации обращать внимание на версию JavaScript, которую поддерживают большинство веб-браузеров.

Именно поэтому, необходимо использовать Babel – преобразователь кода, которые позволит использовать новые возможности JavaScript (ES7) в старых браузерах [10].

Для реализации модульного подхода и ленивой загрузки данных модулей необходимо использовать статический модульный сборщик для приложений на JavaScript – webpack.

Архитектурой микрофреймворка был выбран монорепозиторий, потому что эта архитектура позволяет легко управлять внутренними

зависимостями, производить унификацию кода и легко проводить рефакторинг.

Для реализации данной архитектуры необходимо использовать менеджер пакетов Yarn, который поддерживает создание внутренних пакетов.

Для управления монорепозиторием был выбран Lerna – инструмент, который автоматически создает ссылки на внутренние пакеты.

3.2 Описание реализации

Реализация представляет набор взаимосвязанных модулей.

3.2.1 Модуль Component

Данный модуль представляет набор функций для реализации компонентного подхода.

3.2.1.1 Архитектура данных

Существует 6 видов данных, которые используются в компоненте для реализации необходимого функционала:

1. Stateful – данные, которые изменяются с течением времени и влияют на рендеринг компонента;
2. Stateless – данные, которые не изменяются с течением времени или не влияют на рендеринг компонента.

Исходя из этого, в компоненте присутствуют такие типы данных:

1. State – данные состояние (stateful), являются реактивными;
2. Data – локальные данные компонента. Имеют начальные значения, которые не являются настраиваемыми (не могут быть переданы в компонент из вне);
3. Props – свойства компонента, имею начальные значения, которые могут быть изменены посредством передачи их из вне;
4. Computed – вычисляемые свойства компонента, используются для преобразование данных состояний в нужный вид;

5. Slots (слоты) – механика для передачи фрагментов кода. Работают как свойства, то есть имеют значение по умолчанию, которое может быть изменено. Передается из компонента в компонент;
6. Options – опции компонента, не имеют начальных значений и являются механизмом наследования HTML атрибутов, настроек стилей и классов, а также различных технических свойств. (например, уникальный идентификатор элемента, который используется при работе с циклами).

3.2.1.2 Методы работы с иерархией компонентов

Позволяют регистрировать и удалять дочерние компоненты. А также устанавливать ссылку на родительский компонент:

1. addChildren – метод регистрации дочернего компонента;
2. removeChild – метод удаления дочернего компонента;
3. setParent – метод регистрации родительского компонента;

3.2.1.3 Методы передачи данных в компонент

Так как в компоненте существуют типы данных, которые могут быть переданы, то существует 3 метода их передачи:

1. passSlots – метод передачи фрагментов кода из родительского компонента;
2. passOptions – метод передачи наследуемых опций компонента;
3. passProps – метод передачи свойств компонента.

3.2.1.4 Методы регистрации начальных данных компонента

Каждый компонент имеет начальные данные (state, data, computed, props), которые необходимо регистрировать. Для того, чтобы данные не имели возможность перезаписи (из-за особенностей JavaScript), необходимо создавать новые экземпляры данных в момент инициализации компонента. Именно для этого и существуют методы, которые возвращают объект соответствующих данных:

1. Data;
2. Computed;
3. State;
4. Props.

3.2.1.5 Методы событийных хуков

Для реализации базовых событий компонентов используются хуки – вызов соответствующих методов компонента. На текущий момент реализованы два вида хуков:

1. Mount – когда компонент инициализирован и вставлен в DOM дерево (или примонтирован во время гидратации);
2. Unmount – когда компонент удален из DOM.

3.2.1.6 Методы рендера и гидратации

Для гидратации и рендера компонентов используются функции-обертки, которые запускают процессы рендеринга и гидратации компонентов, полученные в результате компиляции (пакет @siph/compilate).

3.2.1.7 Методы обработки опций компонентов

Для обеспечения наиболее быстрой обработки опций была выбрана следующая структура данных объекта опций, которая включает некоторые свойства:

1. On – объект, включающий пары ключ-значение, где ключ – это название события, а значение – это функций, которая будет запускаться, когда соответствующее событие произойдет;
2. Attr – объект, включающий пары ключ-значение, где ключ – это название HTML атрибута, а значение – это либо Literal (строковой и численные), либо Expression, изменяющийся вместе с состоянием компонента;

3. StaticStyle – объект, включающий пары ключ-значение, где ключ – это название CSS свойство, а значение – соответствующее значения данного свойства;
4. StaticClass – строка, включающая список классов, разделенные запятой;
5. Style – может представлять собой объект, массив объектов или функцию, которая будет запускаться, когда изменится состояние компонента. Главная результативная единица – это объект схожей с StaticStyle, но с добавлением реактивности;
6. Class – может представлять собой объект, массив объектов или функцию, которая будет запускаться, когда изменится состояние компонента. Главная результативная единица – это объект, включающий пары ключ-значение, где ключ – это название класса, а значение – Boolean, которое определяет должен ли использоваться данный класс или нет (значение также может являться функцией, которая возвращает Boolean);
7. Props – свойства компонента в виде пары ключ-значение, где ключ – это название свойства, а значение – это Expression или Literal.

Для того, чтобы опции имели возможность наследования при использовании дочерних компонентов, был реализованы методы объединения и обработки опций:

1. Options – обрабатывает массив входных опций (1-ый аргумент наследуемые опции, 2-ой аргумент опции компонента);
2. MergeOptions – объединяет опции компонента;
3. MakeOption – производит обработку опции для дальнейшего рендеринга, при этом данный метод не участвует в гидратации;
4. MakeCss – производит объединения StaticStyle и style, StaticClass и class;
5. MakeStyleProp – преобразовывает название CSS свойства из camelCase в kebab-case;

6. ArgToObject – преобразует аргумент свойства опции в объект. Используются для преобразования стилей в единый объект;
7. ArgToString – преобразует аргумент свойства опции в строку. Используются для преобразования классов в единую строку.

3.2.2 Модуль Compiler

Данный модуль используется для преобразования кода в ES6 модуль и определения реактивных частей компонента, которые используются в компиляции кода рендеринга и гидратации.

Для анализа кода используется babel. Babel – это преобразователь (transpiler) – программа, позволяющая менять исходный код одной программы на эквивалентный исходный код на другом языке. В случае с Babel, он переписывает современный Javascript на старый.

Именно поэтому данный пакет включает в себя нужные нам вложенные пакеты, которые позволят преобразовать написанный код в AST и обратно:

1. Parse – преобразовывает JavaScript код в AST дерево;
2. Traverse – пробегает по AST дереву и позволяет заменять и сохранять языковые конструкции;
3. Generator – преобразовывает AST дерево в код.

Исходный код компонента разделяется на блоки:

1. Template – HTML код компонента;
2. Script – JavaScript код компонента.

Блок template преобразуется в функции рендера и гидратации. Блок script преобразуется в ES6 модуль-конфиг компонента, где объявлены все его данные и методы.

3.2.2.1 Определение динамических частей компонентов

Динамической частью компонента может быть либо текст, либо значение HTML атрибута. При условном и циклическом рендеринге также

возможно, что отображение (или наличие в элемента в DOM дереве) DOM элемента будет динамическим.

Поэтому для определения динамических частей, необходимо проанализировать исходный JavaScript код компонента, выявить stateful данные.

Для этого используется знак $ в названии переменной при ее декларировании и конструкция стрелочной функции для вычисляемых свойств (листинг 1, 2, 3 и 4).

Листинг 1 – Инициализация реактивных данных (state)

```
let $counter = 1;
```

Листинг 2 – Инициализация вычисляемых данных (computed)

```
let computedCounter = () => {
     return counter + 1;
};
```

Листинг 3 – Инициализация свойств компонента (props)

```
let gridSize = Number | 24;
```

Листинг 4 – Инициализация локальных данных (data)

```
let counter = 1;
```

При инициализации свойств необходимо учитывать синтаксис инициализации. Первый аргумент – тип передаваемых данных, если указан Any или Observable, то данное свойство учитывается, как динамическое. Далее, после знака «|» указывается значение свойства по умолчанию, если не передано иное.

3.2.2.2 Преобразование JavaScript кода

Кроме определения данных, в компоненте также определяются его основные функции. Для объявление методов и хуков (mounted и unmounted) компонента используются функции (Листинг 5).

Листинг 5 – Объявление методов

```
function click(event) {
     alert('User just clicked on button');
}
```

Пример преобразования кода в ES6 модуль представлен в листинге 6 и 7.

Листинг 6 – Пример исходного кода компонента, блок script

```
let $counter = 10;
let size = Number | 24;
let intervalTimer = null;

let multiplier = () => {
    return counter * counter;
}

Function mounted() {
intervalTimer  = setInterval(() => {
           counter += 1;
});
}

Function unmounted() {
     clearInterval(intervalTimer)
}
```

Листинг 7 – Преобразованный код

```
export default {
  props: {
    size: {
      type: Number,
      default () {
        return 24;
      }
    }
  },
  data() {
    return {
      intervalTimer: null,
    }
  },
  state(o) {
    return {
      counted: o(10)
    }
  },

  computed(c) {
    return {
      multiplier: c(() => {
```

Продолжение листинга 7

```
          return this._state.counter() * this._state.counter();
      })
    }
  },
  methods: {
    mounted() {
      this._data.intervalTimer = setInterval(() => {
        this._state.counter(this._state.counter() + 1);
      });
    },
    unmounted() {
      clearInterval(this._data.intervalTimer)
    }
  }
}
```

3.2.2.3 Преобразование шаблона в рендер-функцию

Так как для тестирования было определено использование Hyperscript, то шаблон компонента преобразовывается в данную нотацию.

Листинг 8 – Пример HTML шаблона компонента

```
<template>
     <div class="button" :class="'new-button'" :style="[{ color: testColor
}]" @click="click" style="border-radius: 15px;" disabled>
           {{ s1 }}
           <slot class="s" tag="div">
                 Default button text
           </slot>
     </div>
</template>
```

Листинг 9 – Результат работы компилятора (рендер функция)

```
h(
  "div",
  [
    ctx.options,
    {
      staticClass: "button",
      staticStyle: {
        "border-radius": "15px",
      },
      class: "new-button",
      style: [{ color: ctx._computed.testColor }],
      attrs: {
        disabled: true,
```

Продолжение листинга 9

```
      },
      on: {
        click: ctx.click,
      },
    },
  ],
  [
    () => {
      return `${ctx._state.s1()}`;
    },
    slot(
      ctx,
      h,
      "default",
      "div",
      {
        staticClass: "s",
        props: {
          tag: "div",
        },
      },
      [`Default button text`]
    ),
  ]
);
```

3.2.2.4 Преобразование шаблона в функцию гидратации

Для гидратации используется другой подход. HTML шаблон преобразовывается в инструкции-объекты, которые запускаются на этапе гидратации DOM дерева.

Пример функции гидратации, полученный в результате работы компилятора представлен в листинге 10.

Листинг 10 – Результат работы компилятора (функция гидратации)

```
{
  _t: "h",
  t: "div",
  o: [
    ctx.options,
    {
      staticClass: "button",
      class: "new-button",
      style: [{ color: ctx._computed.testColor }],
```

Продолжение листинга 10

```
      on: {
        click: ctx.click,
      },
      _s: true,
    },
  ],
  c: [
    {
      _t: "t",
      t: () => {
        return `${ctx._state.s1()}`;
      },
    },
    -1,
  ],
};
```

3.2.3 Модуль Loader

Данный пакет используется для создания сборки проекта. А именно написан для использования в webpack в качестве loader – загрузчика.

Загрузчики (лоадеры) позволяют вебпаку обрабатывать не только файлы JavaScript, т.к. сам по себе webpack понимает только JS [12].

Загрузчики трансформируют все типы файлов в модули, которые затем можно добавить в граф зависимостей вашего приложения (а значит, и в итоговую сборку) (Рисунок 5).

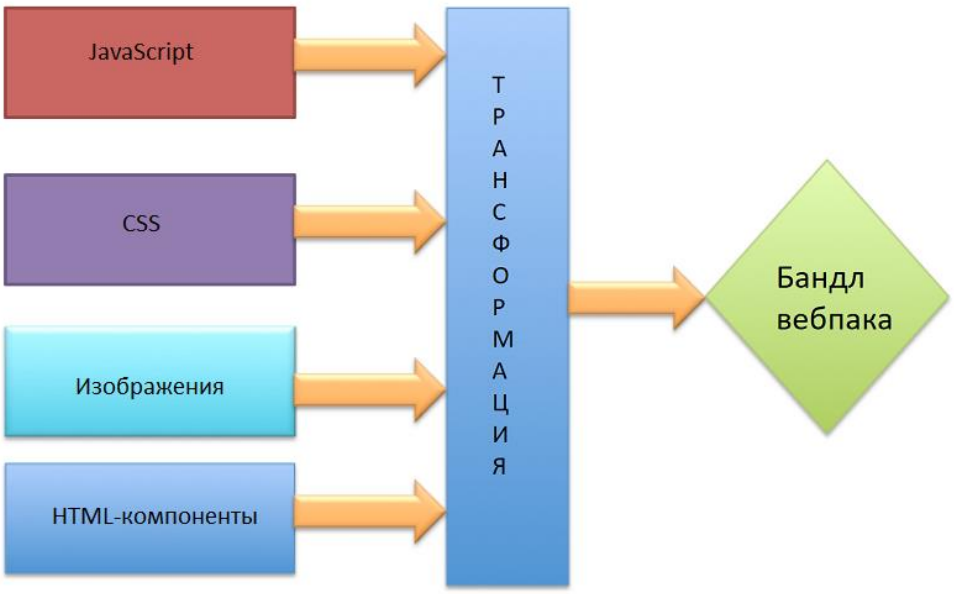

Рисунок 5 – Схема работы сборщика webpack

Данный пакет также используется для создание функциональных компонентов, которые не требуют инициализации, так как не имею своего состояния. Loader автоматически определяет наличие состояние у компонента (это происходит на этапе компиляции компонента) и не требует дополнительного кода.

Loader позволяет значительно уменьшить объем кода и вынести работу компилятора на сторону сервера, причем компилятор запускается единожды при обновлении версии приложения.

Из-за использования Loader итоговый код фреймворка снизился с 1.2мб до 7.1кб, что позволяет его использовать на мобильных устройствах.

3.2.4 Модуль Render

Данный пакет приносит такие возможности в фреймворк, как:

1. Loops – цикличный рендеринг компонентов и NodeElement;
2. Statement – условный рендеринг компонентов и NodeElement. Возможно использование конструкций if, else-if и else;
3. Slots – рендеринг внешнего Hyperscript кода.

Также для реализации компонентного подхода была изменена render функция (h) у Sinuous. Что позволяет инициализировать компоненты на лету,

регистрировать их дочерние компоненты и передавать свойства, опции и слоты, с помощью методов, описанных в пункте 3.2.1.

Также данный пакет реализует метод render, который позволяет произвести рендеринг компонента и его зависимостей на странице. Метод render также работает и на серверной стороне с помощью JSDOM.

Листинг 11 – Пример рендера компонента

```
import Siph from '@siph/i';
import { render } from '@siph/render';
import TestComponent from 'components/test';
import TestPage from 'page/test';

Siph.registerComponent(TestComponent);

render(TestPage, document.getElementById('layout'));
```

3.2.5 Модуль Hydration

Данный пакет позволяет произвести частичную гидратацию. Причем метод statement и slots реализованы заново, а метод loops был преобразован для получения начальных данных отрендеренных компонентов на стороне сервера.

Для того, чтобы частичная гидратация работало максимально производительно, выявлено несколько важных пунктов:

1. Так как сервер занимается первичной отрисовкой компонента, то первичная гидратация не нужна (кроме слушателей событий). Это означает, что изменение свойств и DOM дерево происходит только после изменения данных, но не при их инициализации;
2. Для реализации условного и цикличного рендеринга (в дальнейшем возможно и для других типов рендеринга), необходимо либо передавать начальные данные условий рендеринга с сервера, либо вычислять их на моменте гидратации;
3. Необходимо разделять компоненты с реактивными свойствами и статичными, потому что в большинстве случаев простые элементы

пользовательского интерфейса (например, кнопки) используются в статичных вариациях. Реактивные свойства изменяют логику работы компонентов настолько, что произвести частичную гидратацию при динамических типов свойств невозможно.

Для того, чтобы произвести гидратацию на стороне клиента, необходимо запустить всего один метод – hydrate.

Листинг 12 – Пример гидратации компонента

```
import Siph from '@siph/i';
import { hydrate } from '@siph/hydration';
import TestComponent from 'components/test';
import TestPage from 'page/test';

Siph.registerComponent(TestComponent);

hydration(TestPage, document.getElementById('layout'));
```

При этом в данном пакете используются пять методов, определяющих работу частичной гидратации:

1. initHydration – инициализации частичной гидратации;
2. hydrate – запуск процесса гидратации для инструкции, полученной в методе hydrate компонента;
3. hydrateTag – гидратация тега. Включает инициализацию компонента и передачу свойств, опций и слотов. Или гидрирование текста и HTML-атрибутов, если тегом является HTML-тег, а не зарегистрированный компонент;
4. hydrateSlots – метод гидратации слотов;
5. hydrateText – метод гидратации текстовых элементов (TextNode);
6. hydrateLoop – метод гидратации циклов;
7. hydrateStatement – метод гидратации условного рендеринга;
8. hydrateH – метод гидратации HTML-атрибутов и вложенных элементов;
9. hydrateProps – метод гидратации HTML-атрибутов и добавления слушателей событий.

Также важно учитывать, что в процессе компиляции функции гидрирования, статичные элементы заменяются символов «_» и пропускаются при гидрировании, а опции элементов очищаются от статичных данных.

3.2.6 Модуль Lazy

Данный пакет позволяет использовать динамический импорт компонентов и функционал разделение кода webpack для уменьшения размера итогового кода и увеличение скорости загрузки страниц.

Для того, чтобы компонент был загружен на странице только в момент его использования, можно использовать код, приведенный в листинге 13.

Листинг 13 – Динамическое подключение компонента

```
const IndexPage = import('../pages/index.sin')
```

Метод loadComponent данного модуля определяет, когда загружен компонент на страницу и позволяет вызывать нужный метод через механизм callback.

Данный модуль используется при рендеринга и гидратации, поэтому нет необходимости разрабатывать свои обработчики ленивой загрузки компонентов.

3.2.7 Модуль I

Данный пакет используется для управления основными функциями фреймворка:

1. registerComponent – глобальная регистрация компонента;
2. hasComponent – проверка, зарегистрирован ли компонент;
3. getHydrateComponent – получение экземпляра компонента для гидратации или «_», если компонент является статичным;
4. getComponent – получение экземпляра компонента.

3.2.8 Пример использования и работы микрофреймворка

Для того, чтобы начать использовать микрофреймворк необходимо поставить зависимости с помощью NPM:

```
npm -i @siph/compiler @siph/component @siph/hydration @siph/i @siph/lazy
@siph/loader @siph/render –save
```

Или с помощью Yarn:

```
yarn add @siph/compiler @siph/component @siph/hydration @siph/i @siph/lazy
@siph/loader @siph/render
```

Далее необходимо подключить сборщик проекта – webpack и внести в конфигурацию настройки сборки файлов с расширением sin (компонентов разработанного фреймворка):

```
{
  test: /\.sin$/,
  use: [
  {
    loader: '@siph/loader',
    options: {
      parseName(file) {
        file = camelize(file);
        let rootPath = camelize(path.resolve(__dirname, '../'));

          let componentPath = file
            .split(rootPath)
            .join('')
            .replace(/\.sin/i, '')
            .replace(/Components/, '')
            .replace(/(\s|\/)/g, '');

          return componentPath;
        }
    }
  }
 ]
}
```

Также в webpack необходимо указать точку входа – JavaScript файл, который будет загружать и регистрировать компоненты, а также производить рендеринг.

После того, как загрузчик разработанного фреймворка «зарегистрирован» и настроена точка входа, можно приступить к написанию компонентов.

Для этого создадим папку components в корне проекта и создадим файл «home.sin» – компонент главной страницы приложения.

Создадим hello-world приложение – счетчик. Для этого необходимо вставить данный код в созданный файл:

```
<template>
     <div class="main">
          <h1>Добро пожаловать на главную страницу</h1>
          <hr>
          <h2>Пример реализации счетчика (hello-world)</h2>
          <div class="button" @click="add">Добавить</div>
          <div class="button" @click="sub">Убавить</div>
          Текущие значение счетчика: {{ d }}
     </div>
</template>

<script>
let $d = 1;

function add()
{
     d += 1;
}

function sub()
{
     d -= 1;
}
</script>
```

Данный код реализует функционал счетчика, если нажать на кнопку «Добавить», то текущее значение счетчика увеличиться на 1 и отобразиться на html странице без перезагрузки страницы и специальных методов. Если нажать на «убавить», произойдет тоже самое, только значение убавиться на 1 (рисунок 6 и 7).

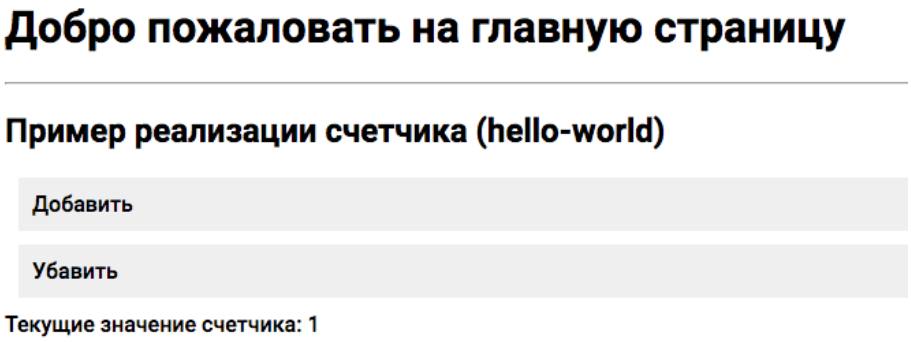

Рисунок 6 – Результат рендеринга компонента home.sin

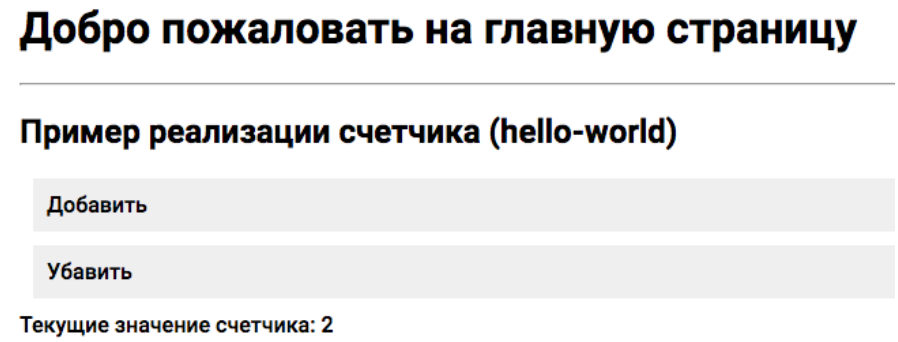

Рисунок 7 – Результат после нажатия на «Добавить»

Далее, в файле точке-входа необходимо зарегистрировать данный компонент. Для этого импортируем созданный компонент и глобальный инстанс Sinuous для регистрации данного компонента:

```
import Sinuous from '@siph/i';
import HomePage from 'components/home.sin'
```

Также для того, чтобы данный компонент отрендерился, необходимо подключить библиотеку render:

```
import { render } from '@siph/render';
```

Далее необходимо запустить функцию render и привязать ее к корневому узлу на html странице, которая будет создана далее:

```
render(HomePage, document.getElementById('layout'));
```

Теперь необходимо создать страницу приложения «index.html» в корне приложения, где подключить build-скрипт, который собирает webpack:

```
<html>
     <head>
           <title><title>Страница</title></title>
           <meta   name="viewport"   content="width=device-width,   initial-
scale=1.0, maximum-scale=1.0,user-scalable=0">
           <script src="./build/main.js"></script>
     </head>
     <body>
           <div id="layout"></div>
     </body>
</html>
```

Если открыть страницу index.html в браузере, то запуститься код, компонент поместиться на страницу и станет интерактивным, как это отображено на рисунках 6 и 7.

3.3 Результаты тестирования

Так как VueJS, React и Angular имеют схожую производительность, то для простоты будем использоваться сравнение с VueJS, потому что синтаксис однофайловых компонентов унаследован именно от данного фреймворка.

Код гидратации размещается в цикле, который выполняется заданное количество раз (10). После этого дата старта гидратации вычитается из даты окончания и вычисляется среднее значение.

Для VueJs (NuxtJs) такой подход не работал, так как не было возможности исправить код фреймворка. Поэтому код гидратации запускался не в цикле, а в ручную. При этом механизм замера скорости использовался такой же, как и в случае своей разработки.

Также помимо «синтетического» тестирования скорости работы был использован инструмент Google PageSpeed, который симулирует запуск на

мобильном устройстве и мог подтвердить результаты «синтетического» тестирования.

Таблица 9 – Характеристики устройства и ПО, на котором производилось тестирование производительности

ОС	macOS High Sierra 10.13.4 (17E199)
Браузер	Google Chrome. Версия 81.0.4044.138 (Официальная сборка), (64 бит)
Процессор	2,7 GHz Intel Core i5
Память	16 ГБ 1333 MHz DDR3
Графика	AMD Radeon HD 6770M 512 МБ

3.3.1 Размер фреймворка

Превышение размера кода всего в 130KB для всех наших ресурсов, может означать невозможность уложиться в 5 секундный интервал загрузки на стандартном телефоне и мобильной сети. Тем не менее некоторые из наших популярных фреймворков могут занимать больше сами по себе.

Таблица 10 – Таблица сравнения размеров

	Siph	Vue (NuxtJS)
Сборка (Gzip)	7.1кб	20.9кб
Страница с 10 000 статичных элементов	39кб	169кб

3.3.2 Производительность статичных веб-страниц

В данном тестировании было задействовано 10 000 статичных компонентов.

Таблица 11 – Таблица сравнения производительности гидратации

Siph	Siph (Функции)	NuxtJS (Vue)	NuxtJS (Функции)
85ms	68ms	455ms	238ms

Как видно из таблицы сравнение производительности гидратации, разработанный фреймворк (Siph) работает в 3.5 (6.69) раза быстрее, чем существующие популярные решения.

3.3.3 Производительность частично-динамических веб-страниц

В данном тестировании было задействовано 10 000 статичных компонентов с одним обработчиком событий нажатия на элемент (onclick).

Таблица 12 – Таблица сравнения производительности гидратации

Siph	Siph (functional components)	NuxtJS (Vue)	NuxtJS (Vue, functional components)
165ms	117ms	707ms	309ms

Как видно из таблицы сравнение производительности гидратации, разработанный фреймворк (Siph) работает в 2.6 (6) раза быстрее, чем существующие популярные решения.

3.3.4 Производительность динамических веб-страниц

В данном тестировании было задействовано 1000 динамических компонентов с одним обработчиком событий нажатия на элемент (onclick), с динамическими стилями и классами. Также были использованы слоты и цикличный рендеринг.

Таблица 13 – Таблица сравнения производительности гидратации

Siph	NuxtJS (Vue)
75ms	185ms

Как видно из таблицы сравнение производительности гидратации, разработанный фреймворк (Siph) работает в 2.64 раза быстрее, чем существующие популярные решения.

3.3.5 Google PageSpeed insights

Данный тест был произведен с симуляцией 3G соединения и замедления процессора до уровня мобильных устройств.

Результат работы инструмента Google PageSpeed для 10 000 частично-динамических компонентов также лучше у фреймворка Siph (рисунки 8 и 9) и составляет улучшение производительности в 2.5 раза.

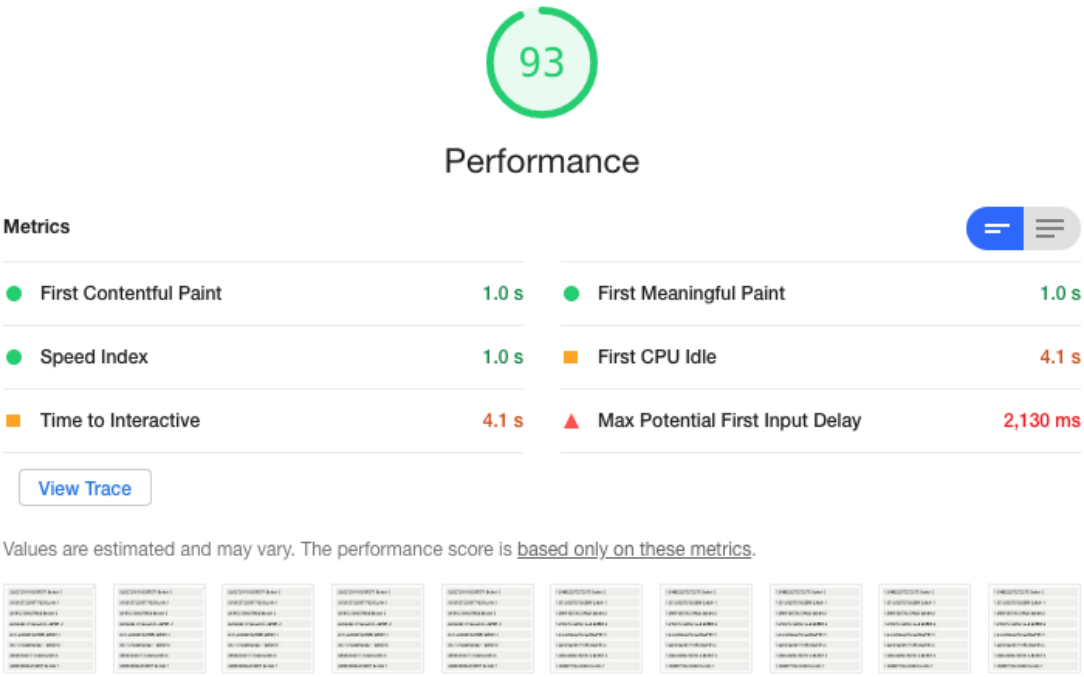

Рисунок 8 – Результат работы анализатора производительности для NuxtJS

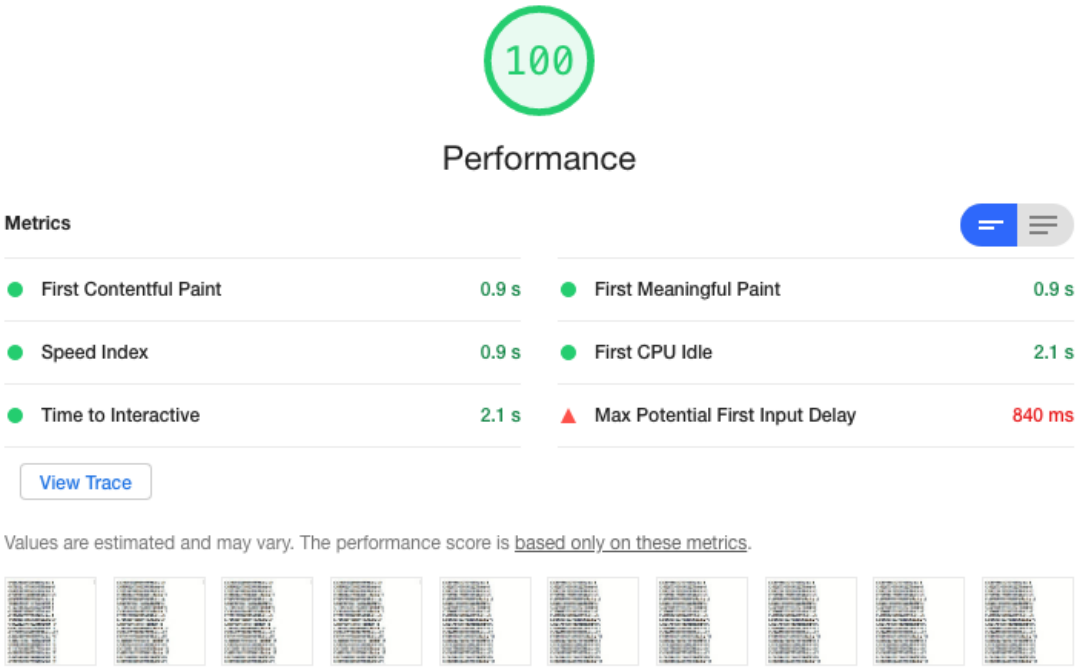

Рисунок 9 – Результат работы анализатора производительности для Siph (частичной гидратации)

3.4 Выводы по главе

В данной главе разработан фреймворк, который поддерживает частичную гидратацию и включает большинство основных функций.

Данный фреймворк позволил произвести измерения производительности и выявить, что относительно текущих фреймворков возможно улучшение производительно веб-приложений от 2.5 до 6.69 раз на стороне клиента при его первичной загрузки.

Также в процессе разработки были выявлены слабые места в производительности текущих популярных фреймворках и сформированы положения, которые необходимо использовать для достижения максимальной производительности и реализации частичной гидратации:

1. Полностью отказаться от виртуального DOM;
2. Использовать tagget templates (генерация HTML кода с помощью конструкций template и innerHTML);

3. Не использовать реактивные библиотеки, в виду того, что автоматическое проставление зависимостей между переменными и функциями сильно замедляют скорость исполнение JavaScript;
4. Для достижения еще большой производительности необходимо отказаться от генерации HTML кода в runtime и переложить данную работу на компилятор (имеется в виду создание собственного компилятора шаблона компонента в нативный JavaScript);
5. Первичная гидратация на стороне клиента не должна взаимодействовать с кодом, а только запускать слушателей событий.

Заключение

В данной работе были проанализированы способы и технологии, которые применяются при разработке веб-приложений, для достижения максимальной производительности.

В результаты был разработан JavaScript микрофреймворк, который имеет большинство функций текущих популярных решений:

1. Цикличный рендеринг;
2. Условный рендеринг;
3. Динамические HTML-атрибуты;
4. Регистрация слушателей событий;
5. Динамические CSS-классы;
6. Динамические CSS-стили;
7. Слоты;
8. Динамический текст (TextNode).

Получившиеся решение доказало, что с помощью частичной гидратации компонентов возможно улучшение производительно веб-приложений от 2.5 до 6.69 раз на стороне клиента во время первичной загрузки, при этом качество и количество функционала соответствует требованиям рынка и не страдает от реализации и внедрения частичной гидратации.

Во время работы над проектом была изучена производительность таких фреймворков, как Vue, Angular и React. Статья [2], посвященная их сравнению опубликована в сборнике научных трудов по материалам XX Международной научно-практической конференции «Актуальные вопросы науки и практики» (5 мая 2020 года, НИЦ «Иннова»).

Разработанный микрофреймворк требует дальнейшей доработки, в соответствии с положениями (параграф 3.4), которые были выявлены в процессе разработки и анализа решений, что позволит добиться еще лучшей производительности и снизить объем кода.

Список использованных источников

1. 6 причин для использования компонентно-ориентированного подхода разработки UI [Электронный ресурс]. – Режим доступа: https://www.tandemseven.com/technology/6-reasons-component-based-ui-development/
2. Бурханов К.С., Родионов В.В. Сравнение производительности Vue, React, Angular [Текст] // Сборник научных трудов по материалам XX Международной научно-практической конференции «Актуальные вопросы науки и практики». – 2020. – С. 93.
3. Исчезающие фреймворки [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/414869/
4. Как разработать реактивные движок. Observables. [Электронный ресурс]. – Режим доступа: https://www.monterail.com/blog/2016/how-to-build-a-reactive-engine-in-javascript-part-1-observable-objects
5. Определения понятий Stateful и Stateless в контексте веб-сервисов [Электронный ресурс]. – Режим доступа: https://medium.com/@ermakovichdmitriy/определения-понятий-stateful-и-stateless-в-контексте-веб-сервисов-перевод-18a910a226a1
6. Почему вы должны выучить реактивное программирование [Электронный ресурс]. – Режим доступа: https://medium.com/corebuild-software/why-you-should-learn-reactive-programming-51b6ffc31425
7. Реактивное программирование. Начало [Электронный ресурс]. – Режим доступа: https://medium.com/@oxmap/реактивное-программирование-начало-4ad548a7d41c
8. Серверный и клиентский рендеринг в вебе [Электронный ресурс]. – Режим доступа: https://tproger.ru/translations/rendering-on-the-web/
9. Тестирование производительности JavaScript фреймворков [Электронный ресурс]. – Режим доступа: https://github.com/krausest/js-framework-benchmark

10. Babel плагин [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/490456
11. Google Lighthouse [Электронный ресурс]. – Режим доступа: https://artjoker.ua/ru/blog/google-lighthouse-google-mayak/
12. Webpack loaders [Электронный ресурс]. – Режим доступа: https://webpack.js.org/loaders/

ПРИЛОЖЕНИЕ А

Код модуля частичной гидратации

```
import { api } from 'sinuous';
import { _ } from '@siph/compiler/src/empty';
import Sinuous from '@siph/i';
import { options as parseOptions, h } from '@siph/component';
import { loadComponent } from '@siph/lazy';
import { loop } from '@siph/render';
import hydrateProps from './property';

let OBSERVER;
let STORAGE = {};

function hydrateH(context, el, options, children)
{
	hydrateProps(context, el, options);

	for (var i = 0; i < children.length; i++) {
		hydrate(context, el.childNodes[i], children[i]);
	}
}

function hydrateStatement(context, node, args)
{
	let parent = node.parentNode;
	let startIndex = 0;

	while((node = node.previousSibling) != null)
		startIndex++;

	let statementArgs = args.a;

	function hideNodes(children, startIndex, length)
	{
		for (var j = startIndex; j <= length; j++) {
			let node = children[j];
			if(node.nodeType !== Node.COMMENT_NODE) {
				node.replaceWith(document.createComment(''));
			}

			node = node.nextElementSibling;
		}
	}

	api.subscribe(() => {
		let currentIndex = startIndex;
		let foundCondition = false;

		for (var i = 0; i < statementArgs.length; i+= 3) {
			let condition = statementArgs[i];
			let size = statementArgs[i + 1];
			let component = statementArgs[i + 2];

			let currentNode = parent.childNodes[currentIndex];

			condition = typeof condition === 'function' ? condition() : condition;
```

```
                        if(condition && !foundCondition) {
                                foundCondition = true;
                                if(currentNode.nodeType === Node.COMMENT_NODE) {
                                        //  render
                                        let newNode = component.r(h.bind(context));
                                        currentNode.replaceWith(newNode);
                                } else {
                                        // hydrate
                                        markAsReady(currentNode);
                                        hydrate(context, currentNode, component.h);
                                }
                        } else {
                                hideNodes(parent.childNodes, currentIndex, size);
                        }

                        currentIndex += size;


                }
        });

}

function hydrateLoop(context, node, args)
{
        let condition = args.c;
        let parentNode = node.parentNode;
        let parentChildren = parentNode.childNodes;

        loop(context, args.c, args.k, (item, key) => {

                let node = args.r(h.bind(context), item, key);

                return node;
        }, (registerHydration) => {
                let items = args.c();

                for (var i = 0; i < items.length; i++) {
                        let node = parentChildren[i];
                        let item = items[i];
                        let itemKey = args.k(item, i);

                        if(node) {
                                if(node.getAttribute('data-key') == itemKey) {
                                        markAsReady(node);
                                        hydrate(context, node, args.h(item, i));
                                }
                        }

                        registerHydration(item, i, node);
                }
        }, node.parentNode);

}

/**
 * Maybe need same hydration algorithm as with props
 * Skip first time hydration ???
 */
function hydrateText(context, node, args)
{
```

```
	if(args.t === _) {
		return;
	}

	api.subscribe(() => {
		console.log(node, args.t())
		api.insert(node, args.t(), null);
	});
}


function getSlotNode(el, tag, path)
{
	let node = el;

	for (var i = 1; i < path.length; i++) {
		node = node.childNodes[path[i]];
	}

	return el;
}

function hydrateSlots(context, el, opts = {}, slots)
{
	let bindedNodes = {}

	let slotData = context._slotsData;

	// Find slot binding nodes
	for(let key in slots) {
		if(slotData[key]) {
			let node = getSlotNode(el, slotData[key].tag, slotData[key].path);
			bindedNodes[key] = node;
		} else {
			throw new Error(`There is no ${key} slot namespace defined`);
		}
	}

	// Hydrate slots per binded tag
	for(let key in slots) {
		let data = slotData[key];
		let node = bindedNodes[key];
		let childrenSlots = slots[key];
		let startNodeIndex = data.index;

		if(typeof node === 'undefined') {
			console.warn(el, opts, slotData, el[0]);
		}

		if(childrenSlots.length > node.length) {
			throw new Error('Slot hydration length mismatch');
		}

		for (var i = startNodeIndex; i < childrenSlots.length; i++) {
			hydrate(context, node.childNodes[i], childrenSlots[i]);
		}
	}
}

/**
 * Context setup
 */
```

```
function registerChildren(parent, child)
{
	if(child._functional) {
		parent.addChildren(_);
		return;
	}

	parent.addChildren(child);
	child.setParent(parent);
}

function hydrateTag(context, node, args)
{
	let el = args.t,
		opts = args.o,
		children = args.c;

	if(!Sinuous.hasComponent(el)) {
		hydrateH(context, node, opts, children);
		return;
	}

	let component = Sinuous.getHydrateComponent(el, opts);

	if(component === null) {
		return _;
	}

	context.addChildren(component);

	if(component._functional) {
		let newArgs = component.hydrate({
			_slots: opts.$slots,
		});

		if(opts.$slots) {
			hydrateSlots(component, node, opts, opts.$slots);
		}

		hydrate(context, node, newArgs);

		return;
	}

	component.passProps(opts.props);
	component.passOptions(opts);

	if(opts.$slots) {
		hydrateSlots(component, node, opts, opts.$slots);
	}

	node.$s = component;

	return hydrate(component, node, component.hydrate(component));
}

/**
 * Main hydration function
 */
function hydrate(context, node, args = null)
{
	hydrateIdle(context, node, args);
```

```
}

function markAsReady(node)
{
	node._hydrated = true;
}

function hydrateIdle(context, node, args)
{
	if(args === null || node === null) {
		return;
	}

	if(args._t === 'h') {
		hydrateTag(context, node, args);
	}

	if(args._t === 't') {
		hydrateText(context, node, args);
	}

	if(args._t === 'loop') {
		hydrateLoop(context, node, args);
	}

	if(args._t === 'statement') {
		hydrateStatement(context, node, args);
	}

	return _;

}


export default function initHydration(component, hydrationRoot, timeBenchmark = () => {}, callback = null)
{
	loadComponent(component, (instance) => {

		timeBenchmark('Hydration');

		let tree = [instance];

		Sinuous.clearHID();

		for (var i = 0; i < tree.length; i++) {
			let component = tree[i];
			let node = hydrationRoot.childNodes[i];
			let hydration = component.hydrate(component);

			hydrate(component, node, hydration);
		}

		instance.hook('mounted');

		if(callback) {
			callback(instance);
		}

		timeBenchmark('Hydration');

		return instance;
	});
```

```
}

/**
 * Filter out whitespace text nodes unless it has a noskip indicator.
 *
 * @param  {Node} parent
 * @return {Array}
 */
function filterChildNodes(parent) {
            return parent.childNodes;
   return Array.from(parent.childNodes).filter(
      el => el.nodeType !== 3 || el.data.trim() || el._noskip
   );
}
```