ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО"

(УНИВЕРСИТЕТ ИТМО)

Факультет программной инженерии и компьютерной техники

Направление (специальность) 09.04.04 Программная инженерия

Образовательная программа Веб-технологии

Дисциплина — Проектирование и анализ языков веб-решений

Курсовой проект (работа)

ТЕМА: Сравнительный анализ JavaScript фреймворков для управления состоянием React приложения

ВЫПОЛНИЛ
Студент группы P41081 _____ П. В. Баранов
№ группы подпись, дата ФИО

ПРОВЕРИЛ преподаватель _____И. Б. Государев
ученая степень, должность подпись, дата ФИО

САНКТ-ПЕТЕРБУРГ

2021 г.

Оглавление

1 Введение

Веб-приложения интегрированы в жизнь практически каждого человека. За ответом на определенный вопрос чаще всего люди обращаются к Интернету, большинство задач (повседневных или рабочих) выполняются с помощью веб-технологий. Спрос пользователей на интерактивность приложений породил переход бизнес логики с серверов на сторону браузера.

Отсюда следует, что появилась потребность в правильной организации данных на стороне клиента. Ранее используемая событийно ориентированная архитектура оказалась недостаточной для управления сложной бизнес логикой. Решением данной задачи стали state management фреймворки. Они позволяют:

- Структурно организовать взаимодействие пользователя с данными;
- Наладить взаимодействие ui компонентов друг с другом;
- Оптимизировать процесс перерендеринга приложения

Эти факты обуславливают актуальность темы данного исследования.

Цель курсовой работы – сравнительный анализ JavaScript фреймворков управления состояния для формирования представления об организации архитектуры сервисов в современных реалиях, а также о возможностях используемых для этого библиотек.

Объект исследования – JavaScript state management фреймворки.

Задачи исследования:

1. Провести обзор зарубежных и отечественных научных источников, описывающих процесс управления состоянием веб-приложений и используемых инструментов.
2. Подобрать необходимые для анализа JavaScript фреймворки.
3. Провести эксперимент и собрать необходимые метрики для каждого решения.

4. Провести сравнительный анализ, используя собранные в ходе эксперимента метрики.

2 Основные понятия

В настоящее время интернет играет важную роль в жизни людей. Взаимодействие с информацией (ее поиск, хранение и передача) в большинстве своем построено на работе state менеджмента приложений. Необходимость создания и поддержки бизнес логики растет, а, соответственно, растет потребность в библиотеках, решающих эту проблему.

Сфера работы с состоянием веб-приложений достаточно хорошо развита. Существующие решения можно классифицировать по нескольким признакам:

- По типу состояния:
  - Немутируемое состояние;
  - Мутируемое состояние;
  - Единое хранилище
  - Набор независимых хранилищ
- По принципу обновления:
  - Запуск через dispatch, обработка в reducer, возвращение нового состояния;
  - Вызов соответствующего action метода и изменение текущего объекта;
- По взаимодействию с ui слоем:
  - Через хуки;
  - Через компоненты высшего порядка;

Существует множество инструментов и фреймворков, выполняющих задачу организации бизнес логики разными способами. Чтобы разработать надежную и поддерживаемую архитектуру веб-приложения, необходимо выбрать стек технологий, подходящий под специфику приложения.

3 Сравнительный анализ фреймворков для управления состоянием приложения

3.1 Исследование существующих инструментов

На данный момент существует множество инструментов для настройки бизнес логики React приложений, начиная от встроенных (useState, useReducer) и заканчивая устанавливаемыми фреймворками, которые передают данные через контекст провайдеры. В данной курсовой работе рассматриваются фреймворки, следовательно далее приведен список подходящих под данную задачу инструментов:

- **Redux**

Redux – библиотека для взаимодействия с бизнес логикой, основанная на flux подходе. В приложении заранее декларируется состояние, действия(экшены), которые могут его поменять и логика ответов на каждое такое действие. Последнее является «редьюсером» - это чистая функция, которая принимает два аргумента - предыдущий стейт и экшен - и возвращает копию стейта с изменениями, вызванными действием. При сравнении нового и старого состояния redux эффективно определяет, в каких данных произошел апдейт и вызывает перерисовку только в тех компонентах, которые от них зависят. Зависимость в компонентах описывается через хук useSelector. Стейт приложения может быть разделен по коду но при запуске приложения комбинируется в единый стор через combineReducers.

- **Mobx**

Mobx имеет иной подход, чем инструменты, основанные на flux. Это библиотека, использующая мутацию состояния. Позволяет объявить хранилище состояние через ES6 классы, декларируя в нем свойства как наблюдаемые(observable) и вычисляемые (computed). Вторые, описанные через геттеры(get method()) неявно подписываются на изменения тех вычисляемых и наблюдаемых свойств, которые в них используются.

Изменение свойств происходит через методы, объявленные как action и мутируют объект с данными. Компоненты объявляются как подписчики явно(через компонент высшего порядка observer), и обновляются только при изменении используемых используемых свойств хранилища. Стор приложения можно как комбинировать, так и оставлять отдельно.

- **Reatom**

Данный фреймворк взят как альтернатива двум наиболее популярным решениям в сфере React сообщества, в документации к данному open source проекту на момент написания работы имеется прямое сравнение с redux и mobx, выявление негативных качеств, в сравнении с reatom.

Reatom - декларативная и реактивная библиотека по управлению бизнес логикой приложения. Принцип ее работы основан на flux, как и в redux: для хранения данных декларируются атомы (элементы с информацией), в этот же момент производится подписка на экшены через функции редьюсеры, которые возвращают новое значение. В отличие от redux, reatom позволяет задавать хранилища с данными и действия по их изменению динамически, то есть они могут быть не объявлены в коде, но создастся в runtime приложения. Атомы могут подписываться друг на друга, аналогично принципу работы computed в mobx.

3.2 Выработка критериев сравнения фреймворков

Организация бизнес логики веб-приложения имеет свои особенности. Следовательно, необходимо подбирать фреймворк, исходя из этих особенностей. В контексте данной работы выделен следующий набор метрик анализа решений:

- Размер библиотеки(в gzip)
- Количество скачиваний в неделю
- Наличие api для отладки работы

- Время на вызов перерисовки списков при обновлении
- Время на вызов перерисовки матрицы при обновлении

3.3 Планирование эксперимента

Эксперимент заключается в сравнении различных JavaScript фреймворков по работе с состоянием приложения. В качестве объекта тестирования в эксперименте планируется использовать гит репозиторий с react приложением, разработанным в конструкторе create-react-app с шаблоном typescript. Тестируемые библиотеки:

- Redux;
- Mobx;
- Reatom;

Эксперименты будут разделены по гит веткам. Запуск сценариев будет организован с использованием react scripts.

С помощью каждого фреймворка необходимо разработать восемь тестовых сценариев:

1. Отрисовку списка, который обновляется каждые 100 мс.
2. Отрисовку списка, который обновляется каждые 100 мс, при условии, что каждые 500 мс на фоне происходит операция по добавлению элемента в большой список.
3. Отрисовка матрицы 100x100 элементов, которая обновляется каждые 100 мс.
4. Отрисовка матрицы 200x200 элементов, которая обновляется каждые 100 мс.
5. Отрисовка матрицы 400x400 элементов, которая обновляется каждые 100 мс.

В ходе эксперимента планируется собрать следующие метрики для сравнения фреймворков управления состоянием:

- Размер загружаемого кода для работы библиотеки

- Скорость вызова перерисовки

Также для каждого фреймворка будет собрана информация по наличию api для отладки.

3.4 Проведение эксперимента

3.4.1 Redux - отрисовка списка

3.4.1.1 Отрисовка списка при изменении каждые 100мс.

Redux при отрисовке показывает среднее время в 1 мс:

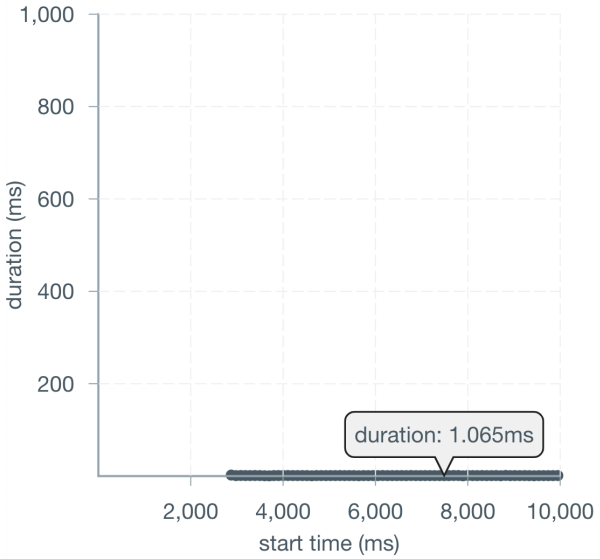


Рис.1 Время обновления списка redux

3.4.1.2 Отрисовка списка при изменении каждые 100мс с долгой операцией каждые 500мс.

Долгая операция заключена в том, что каждые 500мс запускается действие по добавлению числа в массив из миллиона элементов. Несмотря на то, что этот массив не участвует в отрисовке, результаты, в сравнении с предыдущим экспериментом, изменились - среднее время операции осталось тем же, но появились визуальные просадки:

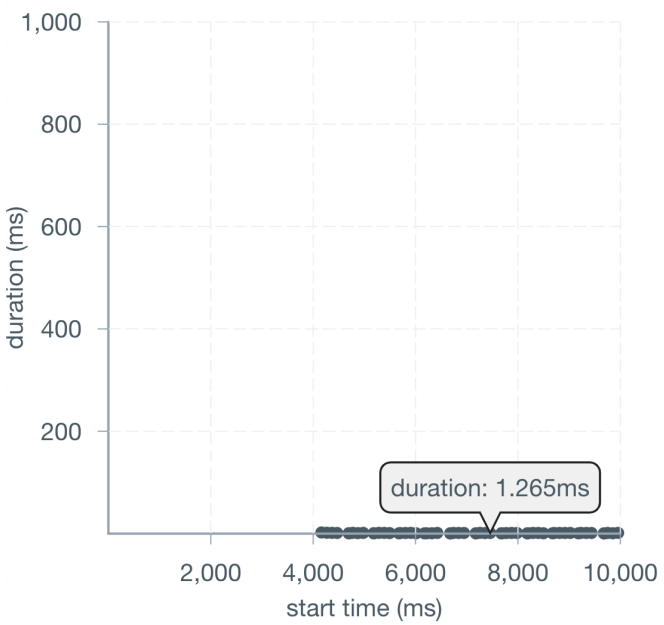


Рис.2 Время обновления списка redux с долгой операцией

Появились пробелы, когда приложение не обновлялось

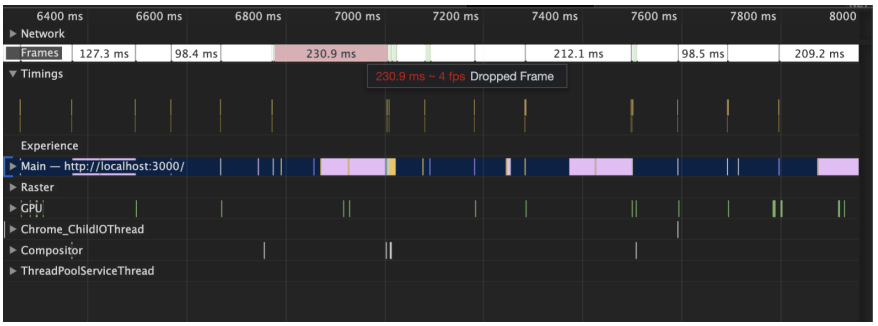


Рис.3 Время обновления списка redux с долгой операцией (devtools)

Это связано с тем, что при redux использует немутируемое состояние, которое при обновлении создает свою копию с полученными изменениями. Следовательно, при добавлении в массив из миллиона элементов еще одно число, redux создает новый массив из миллиона + 1 элемента. В связи с этим появляется просадка. Стоит отметить, что при разработке эксперимента в dev режиме, redux создает копию большого массива, даже если он не менялся вызванным действием. Делается это с целью поддержки time travelling - функционал redux devtools, позволяющий во время отладки приложения вернуться в конкретную точку состояния. Так как у redux единый стор, ему требуется создавать копию всего состояния каждый раз. Такой подход может ухудшить опыт разработки, в связи с долгой перерисовкой приложения.

3.4.2 Mobx - отрисовка списка

3.4.1.1 Отрисовка списка при изменении каждые 100мс.

Mobx при отрисовке показывает среднее время от 0.5 мс до 1 мс:

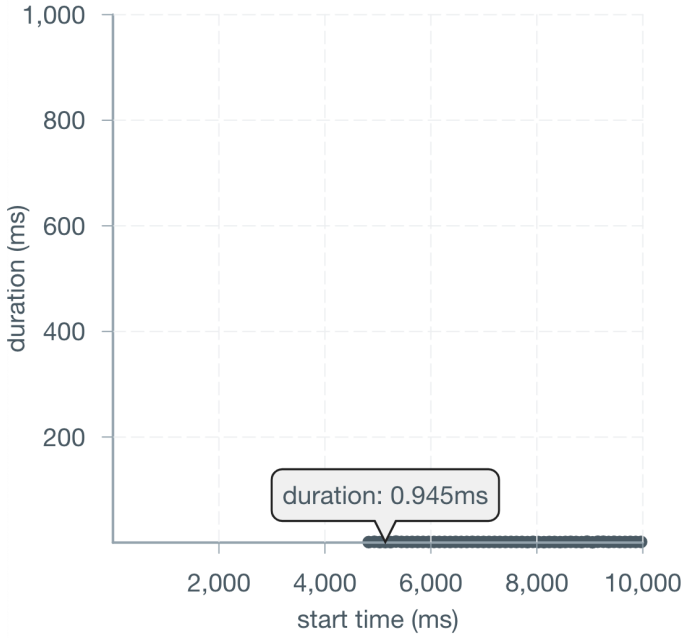


Рис.4 Время обновления списка mobx

3.4.1.2 Отрисовка списка при изменении каждые 100мс с долгой операцией каждые 500 мс.

При операции в добавлении в большой массив элемента, mobx показывает те же результаты, что и без нее:

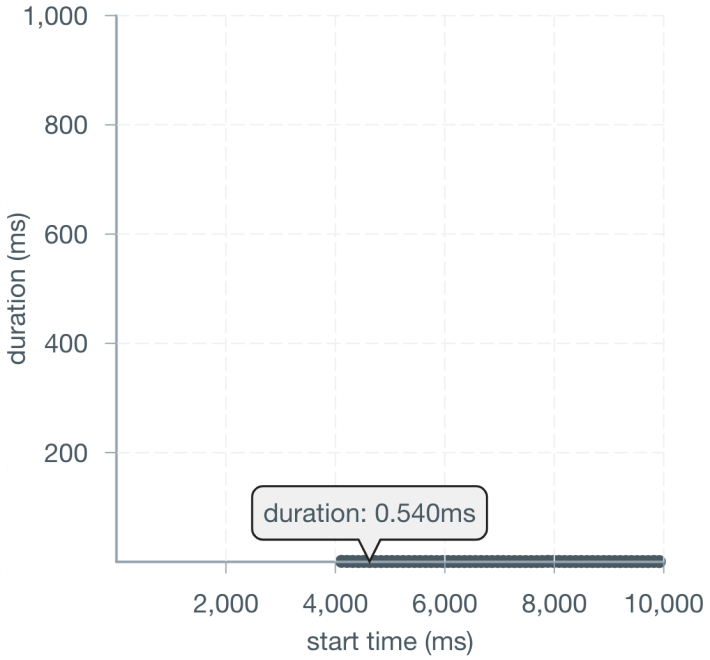


Рис.5 Время обновления списка mobx с долгой операцией

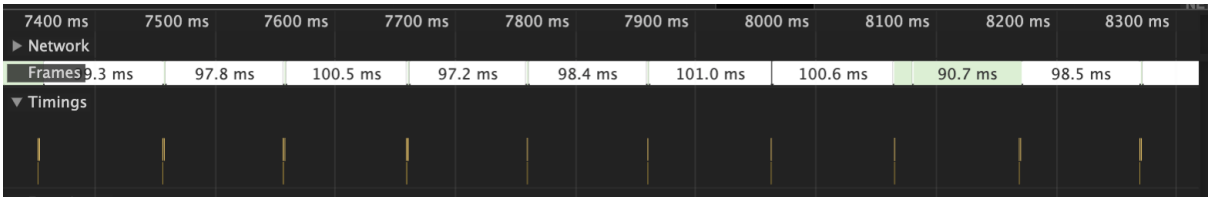


Рис.6 Время обновления списка mobx с долгой операцией(devtools)

Так как mobx не возвращает новый стейт, а мутирует предыдущий, он не имеет просадок, вызванных пересозданием состояния, как в redux.

3.4.3 Reatom - отрисовка списка

3.4.1.1 Отрисовка списка при изменении каждые 100мс

Reatom при отрисовке показывает среднее время от 0.5 мс до 1 мс:

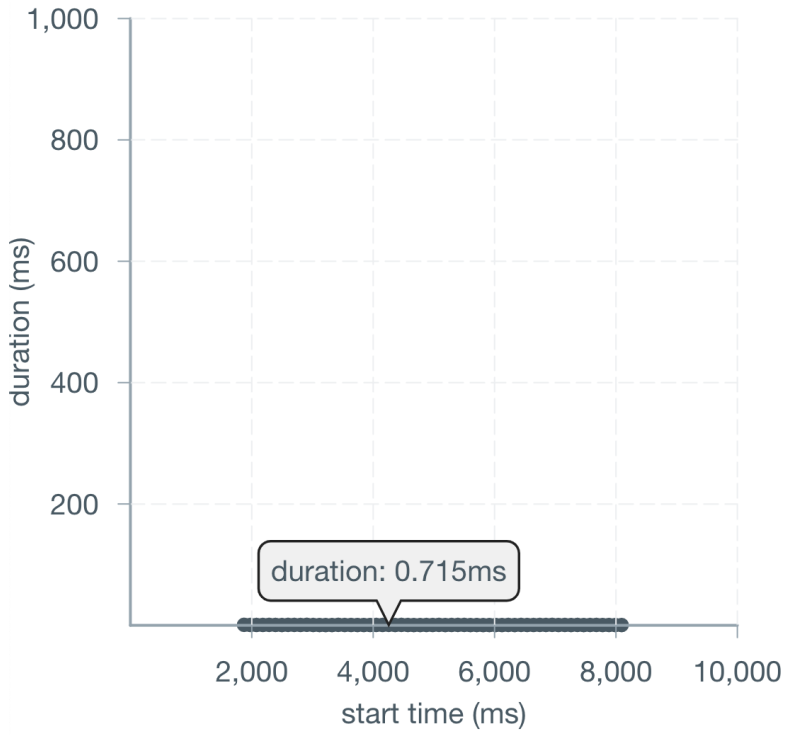


Рис.7 Время обновления списка reatom

3.4.1.2 Отрисовка списка при изменении каждые 100мс с долгой операцией каждые 500 мс.

Результаты reatom с добавлением долгой операции остались прежними:

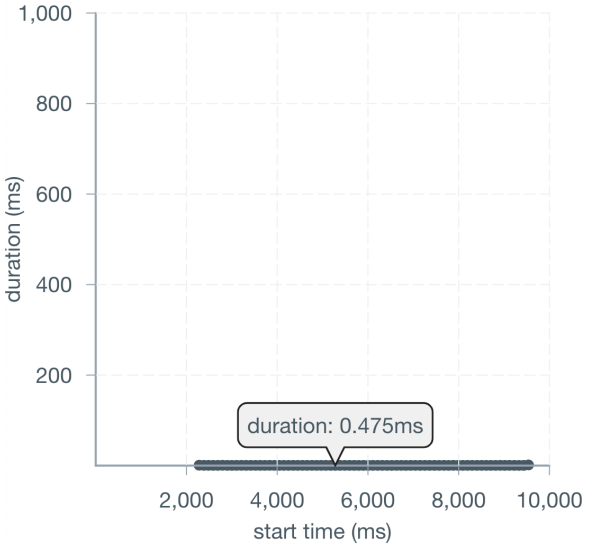


Рис.8 Время обновления списка reatom с долгой операцией

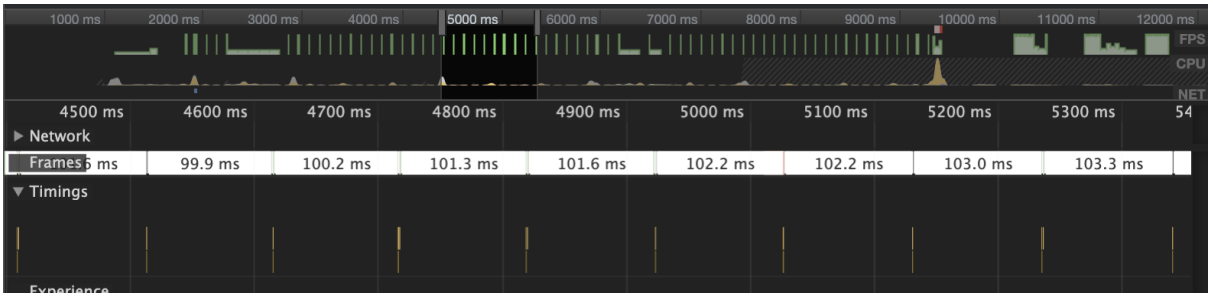


Рис.9 Время обновления списка reatom с долгой операцией

Несмотря на то, что reatom возвращает новый стейт при обновлении атома, изменение происходит отложенно: пока данные из атома не используются в компоненте, редьюсер функция не вызывается.

3.4.4 Redux - отрисовка матрицы

3.4.4.1 Отрисовка матрицы 100x100

Результаты замеров:

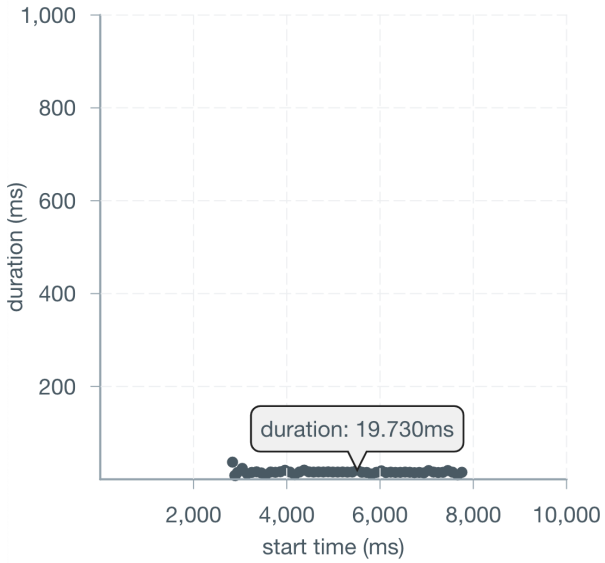


Рис.10 Время обновления матрицы 100x100 redux

В среднем обновление происходит за 15-20 мс. Скорость меньше, в сравнении со списком из первого эксперимента, так как элементов в 1000 раз больше, из чего следует, что в redux скорость выполнения операции обратно пропорциональна количеству данных в хранилище.

3.4.4.2 Отрисовка матрицы 200x200

При увеличении матрицы в 4 раза, удалось получить следующие метрики:

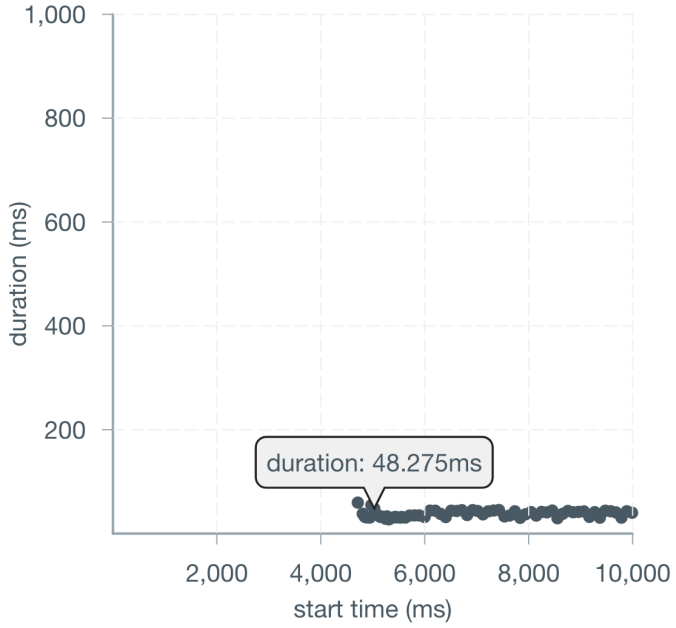


Рис.11 Время обновления матрицы 200x200 redux

Время на обновление состояния уменьшилось приблизительно в 2 раза.

3.4.4.3 Отрисовка матрицы 400x400

При увеличении матрицы еще в 4 раза, время на обновление увеличилось еще примерно в 3 раза ~130 мс, с периодическими просадками до 200 мс.

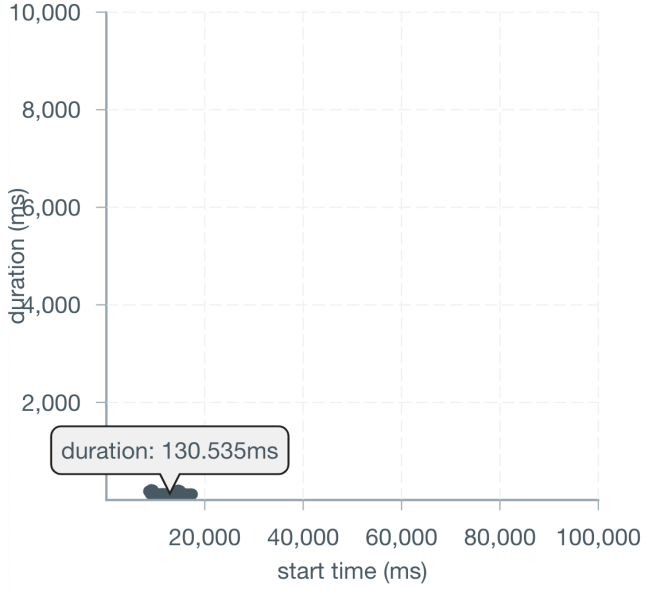


Рис.12 Время обновления матрицы 200x200 redux

3.4.5 Mobx - отрисовка матрицы

В ходе данного эксперимента будет рассмотрено 2 подхода реализации архитектуры:

1) с единым хранилищем, содержащим матрицу объектов, и компонента контейнера, подписанного на него;
2) С хранилищем(стором), которое при инициализации матрицы создает NxN под хранилищ, с организацией подписки каждого элемента матрицы на собственное хранилище;

В первом случае при каждом обновлении матрицы будет перерисовываться контейнер с NxN элементов, во втором случае контейнер отрисуется в первый раз, впоследствии перерисовка будет затрагивать только конкретный измененный элемент матрицы.

3.4.5.1 Отрисовка матрицы 100x100 (единый стор)

Среднее время обновления: 19 мс с единичной просадкой до 46 мс

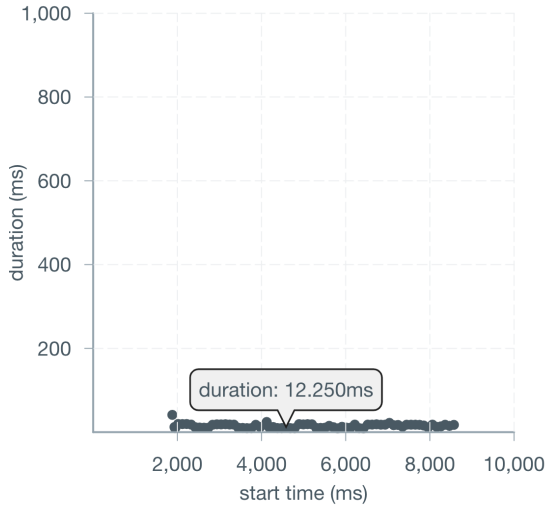


Рис.13 Время обновления матрицы 100x100 mobx

3.4.5.2 Отрисовка матрицы 100x100 (с динамическими сторами)

Результаты замеров: среднее время обновления - 0.5 - 1.5 мс с единичной просадкой до 6 мс

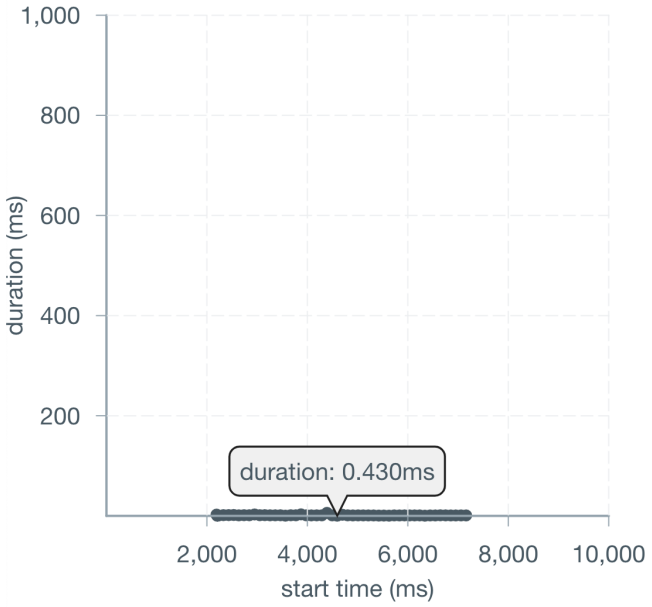


Рис.14 Время обновления матрицы 100x100 mobx

3.4.5.3 Отрисовка матрицы 200x200 (единый стор)

Время на обновление - от 34 до 71 мс

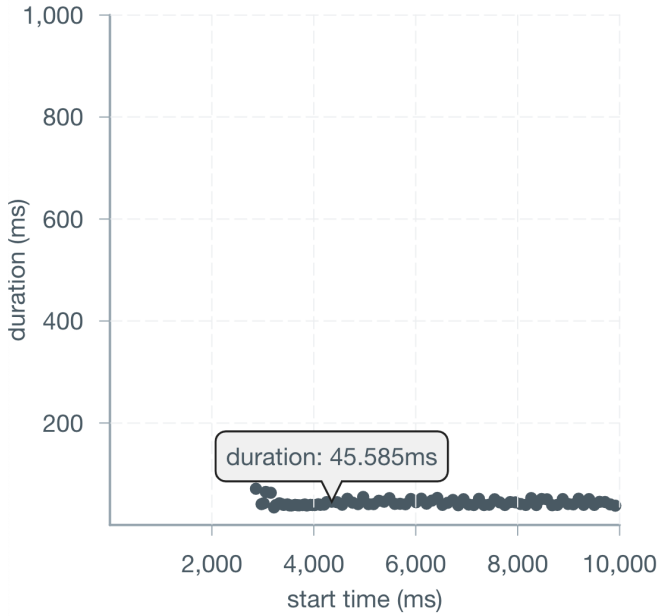


Рис.15 Время обновления матрицы 200x200 mobx

3.4.5.4 Отрисовка матрицы 200x200 (с динамическими сторами)

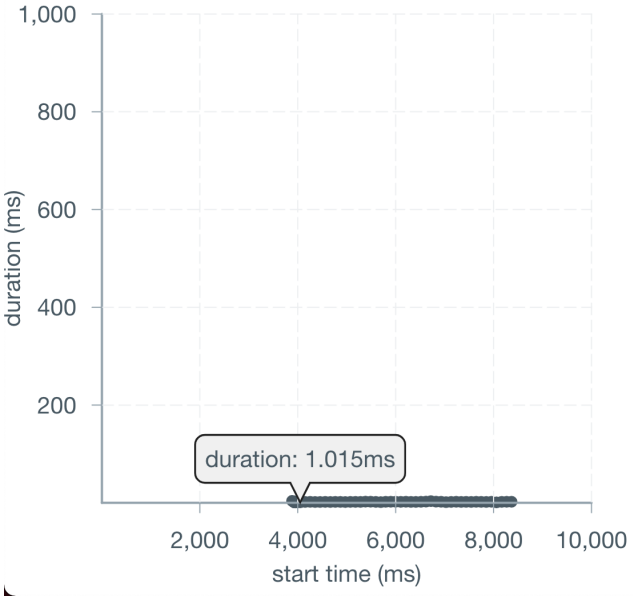


Рис.16 Время обновления матрицы 200x200 mobx

Среднее время обновления - 1 мс

3.4.5.5 Отрисовка матрицы 400x400 (единый стор)

Среднее время: 160 мс с просадками до 280 мс

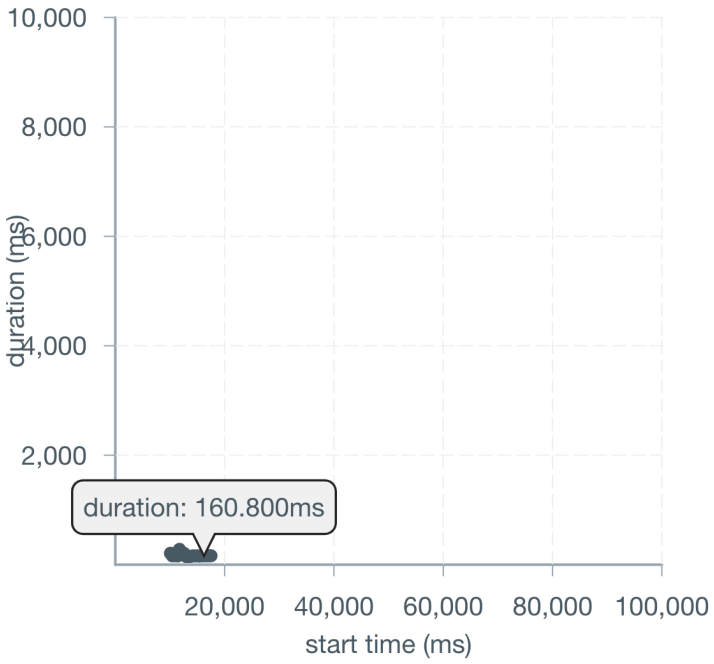


Рис.17 Время обновления матрицы 400x400 mobx

3.4.5.6 Отрисовка матрицы 400x400 (с динамическими сторами)

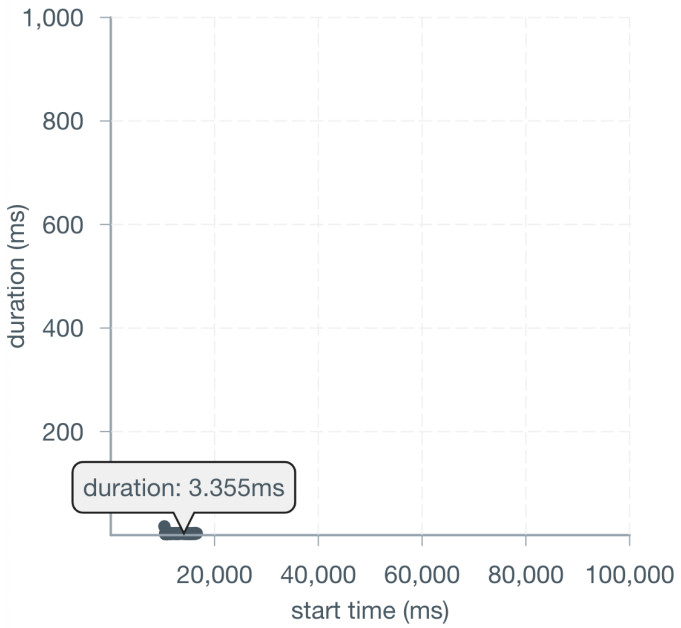


Рис.18 Время обновления матрицы 400x400 mobx

Среднее время - 3.5 мс
Такая разница по времени обновления между двумя подходами обусловлена тем, что процесс обновления данных вызывает уведомление подписчиков. В случае с единым стором существует observable двумерный массив. Согласно документации, элементы observable коллекции рекурсивно становятся observable. компонент, который отрисовывает всю матрицу подписывается в последнем эксперименте на более чем 160 000 обновлений. это нагружает транзакцию замедляет обновление состояния. Во втором варианте создается 160 000 компонентов и сторов, связанных один к одному - это позволяет производит транзакцию в разы быстрее.

3.4.6 Reatom - отрисовка матрицы

В ходе данного эксперимента будет рассмотрено 2 подхода реализации архитектуры:

3) с единым атомом, содержащим матрицу объектов, и компонента контейнера, подписанного на него;
4) С атомом, которое при инициализации матрицы декларирует NxN динамических атомов, и обновление каждого атома по собственному динамически объявленному action-у, с организацией подписки каждого элемента матрицы на уникальный элемент хранилища;

В первом случае при каждом обновлении матрицы будет перерисовываться контейнер с NxN элементов, во втором случае контейнер отрендерится в первый раз, впоследствии перерисовка будет затрагивать только конкретный измененный элемент матрицы.

3.4.6.1 Отрисовка матрицы 100x100 (единый атом)

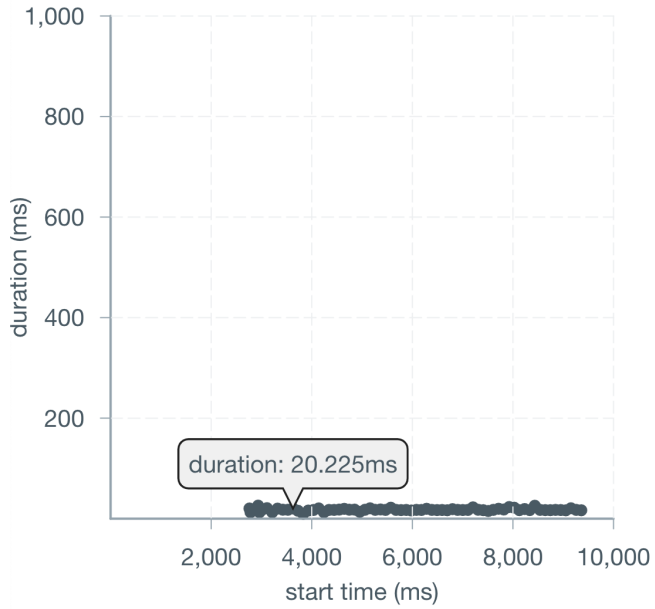


Рис.19 Время обновления матрицы 100x100 reatom

3.4.6.2 Отрисовка матрицы 100x100 (с динамическими атомами)

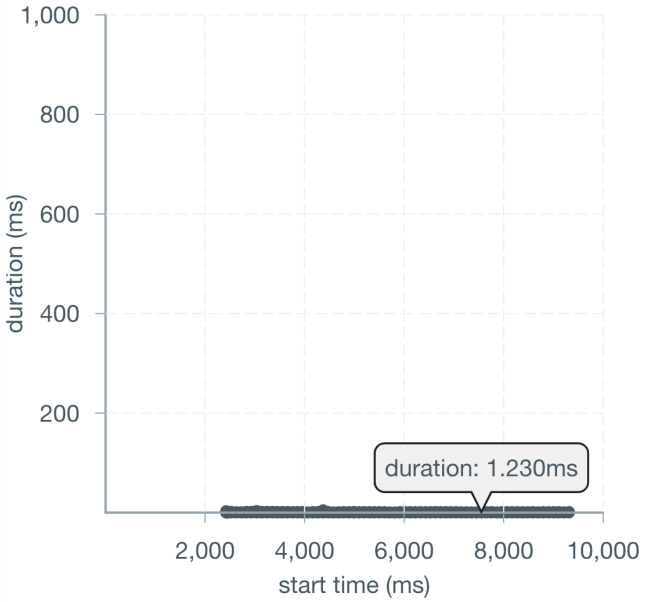


Рис.20 Время обновления матрицы 100x100 reatom

3.4.6.1 Отрисовка матрицы 200x200 (единый атом)

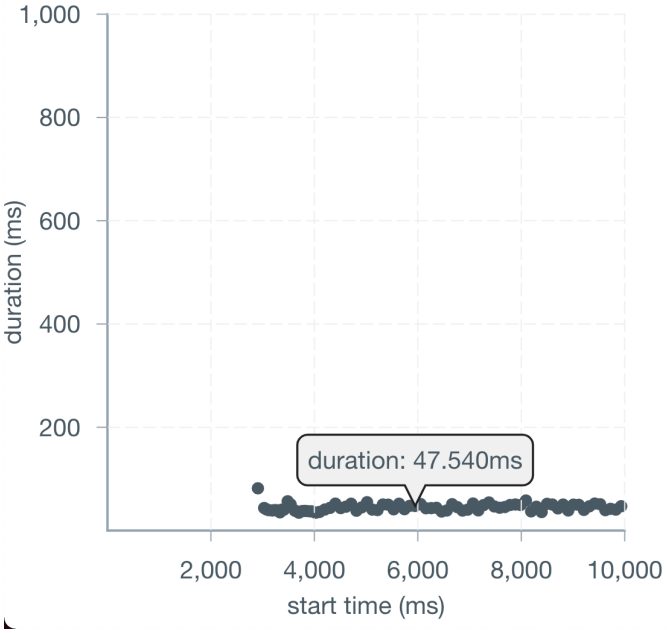


Рис.21 Время обновления матрицы 200x200 reatom

Среднее время обновления 45 мс

3.4.6.2 Отрисовка матрицы 200x200 (с динамическими атомами)

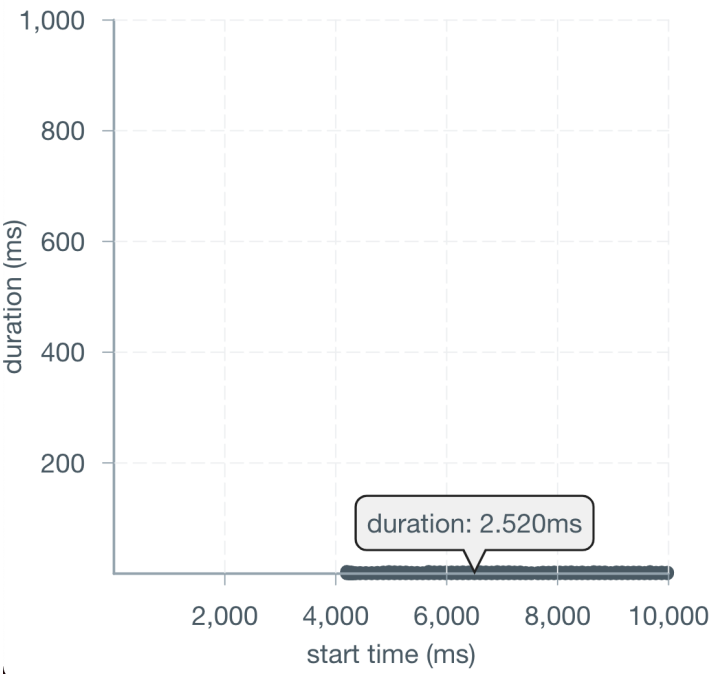


Рис.22 Время обновления матрицы 200x200 reatom

3.4.6.1 Отрисовка матрицы 400x400 (единый атом)

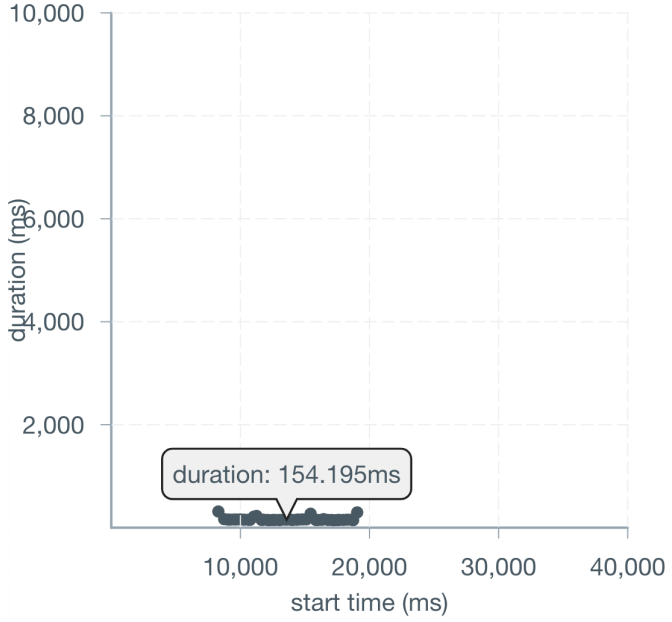


Рис.23 Время обновления матрицы 400x400 reatom

3.4.6.2 Отрисовка матрицы 400x400 (с динамическими атомами)

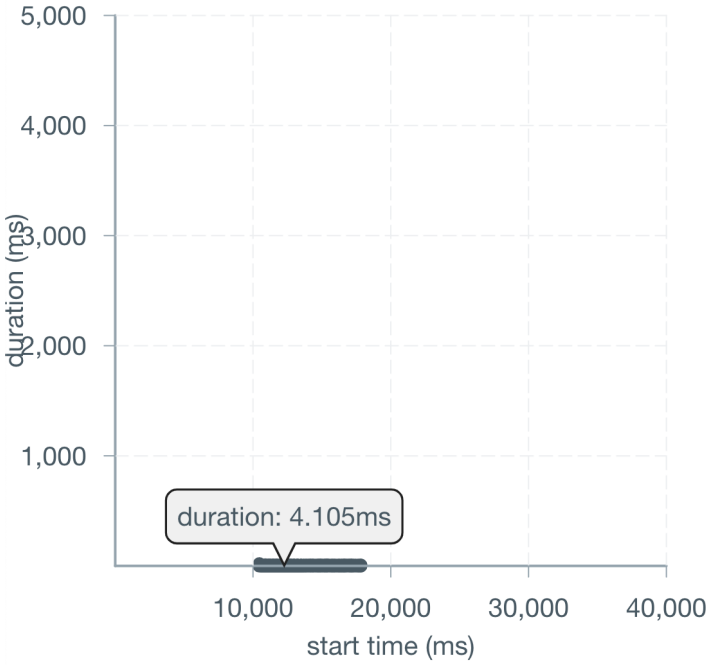


Рис.24 Время обновления матрицы 400x400 reatom

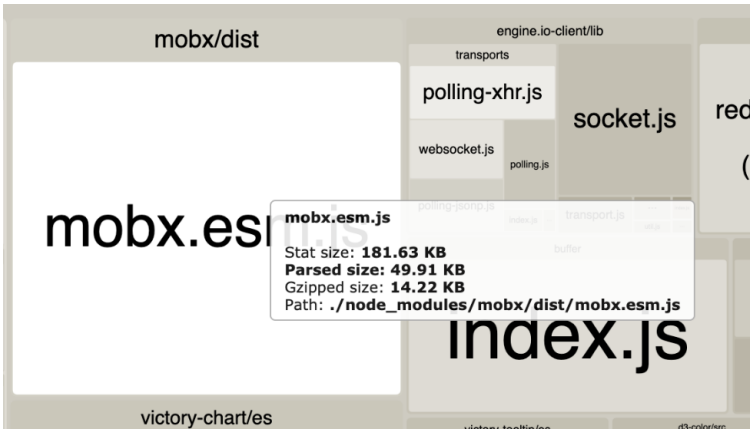
3.4.7 Анализ размера бандла

В ходе эксперимента был произведен расчет размера устанавливаемых зависимостей:

1) Для redux размер gzip составил 8.02 килобайт

Рис.25 Размер бандла redux

2) для mobx размер составил 15.87 килобайт



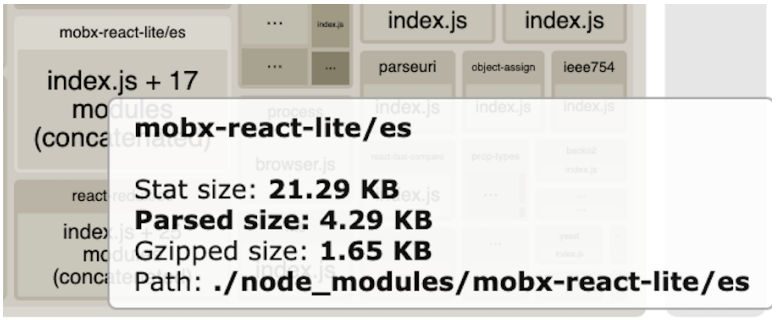
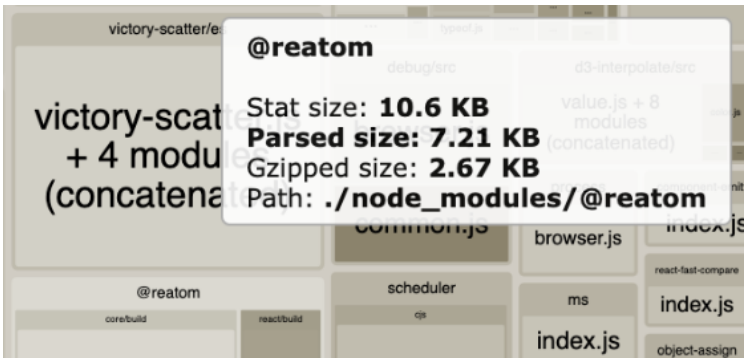


Рис.26 Размер бандла mobx

3) Для reatom размер составил 2.67 килобайт



4) Рис.27 Размер бандла reatom

3.4.8 Наличие api для отладки приложения

У всех redux и mobx имеется собственное api. Reatom не обладает собственным инструментом, но в него интегрируется средство отладки redux.

Redux debug extension в сравнении с mobx debug extension обладает более богатым функционалом, позволяющим возвращать приложение к определенному состоянию с помощью time travelling. Также доступен logger, который описывает, как менялось состояние приложения, в зависимости от действия.

Наиболее функциональным по отладке приложения можно выявить redux и reatom, потому что они используют один и тот же инструмент.

3.4.9 Популярность фреймворка

На момент написания работы, согласно статистике, количество еженедельных скачиваний библиотек составляет

- 5 974 116 у redux
- 632 080 у mobx
- 944 у reatom

При запросе в google страниц с использование названия библиотеки:

- 42 200 000 у redux
- 795 000 у mobx
- 7 150 у reatom

Из этого можно сделать вывод, что наиболее популярным фреймворком является redux, наименее - reatom.

3.5 Анализ результатов эксперимента

Метрики, указанные в эксперименте, собирались посредством web performance api, одинаковым методом для каждого кейса. Скрипты запускались локально.

Эксперимент с отрисовкой списка показал, что при отсутствии эффектов, выполняемых в фоне, все три фреймворка показывают похожие результаты. При добавлении операции вставки в большой массив, redux показывает худший результат в сравнении с mobx и reatom в связи с отсутствием отложенной обработки действий и необходимостью создавать новый массив при каждом его изменении.

Обновление матрицы без оптимизаций быстрее всего происходило у redux, однако с динамическими сторами в mobx и атомами в reatom удалось сократить время обновления в десятки раз. В случае с mobx, паттерн с динамическими сторами описан в документации к фреймворку. Reatom не описывает показанный способ оптимизации, для его приведения в данной работе потребовалось дополнительное исследование.

Согласно интернет источникам, наиболее популярным средством разработки архитектуры веб приложений принято считать redux, а по функционалу в отладке - redux и reatom.

Самым большим по весу с большой разницей оказался mobx, далее идет redux, и самый малый объем кода представляет reatom.

Выводы:

Исходя из полученных результатов, среди JavaScript фреймворков, самую большую производительность показывают mobx и reatom. Какой из них выбрать, зависит от конкретной задачи.

Redux – показал наименьшую производительность в описанных экспериментах. Однако, согласно анализу, у данного фреймворка наибольшая популярность, по нему разработано большое число хороших практик и имеется подробная документация. Он имеет небольшой размер бандла и показывал схожий результат при выполнении простых задач работы с состоянием. Этот инструмент подойдет для сайтов, где не требуется работать с большими данными и есть необходимость в оптимизации поисковой выдачи.

Mobx – это фреймворк с документацией, покрывающей множество кейсов. Он обладает большим набором конфигурации подписок. этот фреймворк обладает самым большим размером в gzip, поэтому его следует использовать в случае, когда на проекте необходимо следить за большим объемом данных, либо, когда для приложения проблема поисковой выдачи не является первостепенной.

Reatom - самый молодой среди рассматриваемых фреймворков. У него нет такой же популярности, однако он показал один из наилучших результатов по перфомансу и имеет наименьший вес. Несмотря на то, что с ним можно использовать апи отладки redux-a, у данного фреймворка нет документации, описывающей редкие кейсы по оптимизации. Он подойдет в случае, когда есть потребность и в работе с большим объемом данных, и в оптимизации поисковой выдачи, есть вероятность, что для решения некоторых задач понадобится исследование.

4 Заключение

В ходе реализации курсового проекта были решены все поставленные задачи:

- проведен обзор зарубежных и отечественных научных источников, описывающих процесс организации бизнес логики веб-приложений и используемых инструментов;
- подобраны необходимые для анализа JavaScript фреймворки;
- проведен эксперимент и собраны необходимые метрики для каждого решения;
- проведен сравнительный анализ с использованием собранных в ходе эксперимента метрик.

Были сделаны следующие выводы:

- Mobx лидируют среди JavaScript state management фреймворков по перформансу, так как предоставляют возможность создания динамических хранилищ данных и мутирование состояния. Это не отменяет того факта, что имеются проблемы при отладке приложения и размером бандла в сравнении с другими решениями.
- Redux обладает наибольшей популярностью, подробно описанной документацией, функциональным инструментарием для отладки и небольшим размером. Тем не менее, иммутабельность стейта ограничивает его производительность, но позволяет работать данными небольших размеров.
- Reatom – инструмент, показавший схожий результат по перформанс тестированию, что и mobx, но обладает наименьшим размером бандла. Имеет интеграцию с redux api, но для решения сложных кейсов нет документации.

5 Список литературы

1. Джейми Аллен, Брайан Ханафи, Роланд Кун Реактивные шаблоны проектирования

2. Mobx [Электронный ресурс] URL: https://mobx.js.org/ (дата обращения: 12.05.2021).
3. Максимов Я. А., Гудкова Е. А. Машины состояний в современном javascript //Современные информационные технологии. – Пензенский государственный технологический университет (Пенза) – №. 31. – С. 42-45.

4. Redux [Электронный ресурс] URL: https://redux.js.org/ (дата обращения 12.05.2021).

5. Reatom [Электронный ресурс] URL: https://reatom.js.org/ (дата обращения 12.05.2021).

6. React [Электронный ресурс] URL: https://reactjs.org/ (дата обращения: 12.05.2021).

7. MDN Web docs [Электронный ресурс] URL: https://developer.mozilla.org/ (дата обращения: 12.05.2021).

6 Приложения

6.1 Код с экспериментами

Код с экспериментами и описание по его запуску можно найти в репозитории https://github.com/daibogh/stm-performance.