МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

Тема: Анализ производительности и оптимизация web-приложений на примере клиент-серверного приложения для отслеживания задач и планирования разработки ПО

Студентка гр. 4304 Лапцевич Д.А.

Руководитель Лисс А.А.

Санкт-Петербург
2019

ЗАДАНИЕ

НА НАУЧНО-ИССЛЕДОВАТЕЛЬСКУЮ РАБОТУ

Студентка Лапцевич Д.А.

Группа 4304

Тема работы: Анализ производительности и оптимизация web-приложений на примере клиент-серверного приложения для отслеживания задач и планирования разработки ПО

Дата сдачи отчета: 26.12.2019

Дата защиты отчета: 27.12.2019

Студентка Лапцевич Д.А.

Руководитель Лисс А.А.

АННОТАЦИЯ

В данной работе рассматриваются способы оптимизации веб-приложений. Изучаются инструменты для измерения производительности и результатов оптимизаций. Приведены основные уязвимости в производительности клиент-серверных приложений и основные источники расходов ресурсов

Даны общие понятия о тестировании производительности. Проведен анализ производительности. Изучена технология тестирования производительности по скорости загрузки сайта. Разработана методика разработки производительных сайтов на примере клиент-серверного приложения для отслеживания задач и планирования разработки ПО.

SUMMARY

This paper discusses ways to optimize web applications. Tools for measuring performance and optimization results are being studied. The main vulnerabilities in the performance of client-server applications and the main sources of resource consumption are described.

General concepts of performance testing are given. Performance analysis conducted. The technology of performance testing by the speed of loading the site was studied. A methodology for developing productive sites was developed using the example of a client-server application for tracking tasks and planning software development.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С развитием мировых технологий экспоненциально увеличивается выбор девайсов (компьютеров, ноутбуков, планшетов, смартфонов) с различной производительностью. Веб-приложения являются кроссплатформенными инструментами для пользователя.

Для обеспечения притока новых пользователей и сохранения пользователей уже использующих приложения необходимо создавать производительные приложения, которые могут работать эффективно как на машинах с высокой производительностью, так и с низкой.

В современном мобильном мире сложно представить свое существование без постоянного карманного помощника. Со стороны же производительности мобильные телефоны обладают одними из самых низких показателей среди современных девайсов. Поэтому очень важно обеспечить работу приложения с минимальными ресурсными затратами. Целью данной работы является тестирование и анализ производительности на примере веб-приложения для отслеживания задач и планирования разработки ПО, а также выявления наиболее оптимальных подходов реализации сайтов с высокой производительностью.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Архитектура клиент-серверного взаимодействия

Веб-приложения представляют собой тип программ, построенных по архитектуре “клиент-сервер”. Клиент-серверная модель представляет собой структуру приложения, которая распределяет задачи и нагрузки между поставщиками ресурсов и услуг, называемым серверами и теми, кто запрашивает эти услуги, называемыми клиентами. По сути клиенты и серверы представляют собой программное обеспечение. Как правило они расположены на различных вычислительных машинах, и обмениваются данными по вычислительной сети посредством сетевых протоколов, но иногда клиент и сервер могут находиться на одном компьютере. Хост сервера запускает одну или несколько серверных программ, которые распределяют свои ресурсы между клиентами. Клиент запрашивает содержимое сервера, но не передает ничего. Серверы ожидают запросы, а клиенты инициируют сеансы связи с ними.

Запросы клиента обрабатываются на сервере - там, где расположена База Данных и Система Управления Базой Данных (СУБД). Это дает преимущество в отсутствии пересылки больших объемов данных, а также запрос оптимизируется таким образом, что на него тратится минимум времени. Все это повышает быстродействие системы и снижает время ожидания результата запроса. При выполнении запросов сервером существенно повышается степень безопасности данных, поскольку правила целостности данных определяются в базе данных на сервере и являются едиными для всех приложений, использующих эту БД [1].

Функции клиента:

- инициализация запроса серверу;
- обработка результатов запросов, полученных от сервера;

- представление результатов запроса пользователю в форме интерфейса пользователя.

Функции сервера

- прием запросов от клиента;
- обработка запросов;
- выполнение запросов к Базе Данных и их оптимизации;
- отправка результатов запросов клиенту;
- обеспечение системы безопасности;
- обеспечение стабильности многопользовательского режима работы.

1.2. Производительность

Разработка программного обеспечения, развивалась годами, начиная с ручного тестирования, но в те времена требования были намного ниже, сайты были текстовые и загружались в течение нескольких минут. Поэтому у веб-разработчиков было гораздо меньше стимулов для предварительного тестирования. Но ставки выросли, как только электронная коммерция набрала обороты в мире веб-разработки. Поэтому тестирование стали проводить в среде разработки. Но с ростом приложений начали автоматизировать и тестирование.

Разработчики начали писать автоматизированные тесты. В конце концов, тестирование созрело до такой степени, что оно распространилось за пределы простых наборов тестовых модулей и интеграционных тестов в стиле воспроизведения. Организации начали строить все более изощренные, тонкие тестовые комплекты.

На сегодняшний день, ставки для работы приложений стали выше, чем когда-либо. Тесты работы приложения уже давно стали стандартом. Так как приложения стали слишком сложными для ручного тестирования, были создали тестовые фреймворки, с помощью которых можно было автоматизировать тестирование. И любой хороший код начинается с 9 написания тестов. Но это

не позволяет узнать, как будет вести себя приложение в дикой среде. Тестирование производительности веб-приложений исправляет это.

Тестирование производительности - это форма тестирования программного обеспечения, в котором основное внимание уделяется тому, как система работает под определенной нагрузкой. Тестирование производительности должно дать организациям диагностическую информацию, необходимую им для выявления и устранения узких мест.

Медленная работа приложений влияет на то, что платные пользователи уходят, а новых подписчиков становится меньше, что влияет на доходах.

Зачастую решать проблемы с производительностью оказывается очень затруднительно в связи с тем, что разработчикам сложно воспроизводить такого рода “баги”. Проблемы с производительностью напрямую не затрагивают поведение программного обеспечения. Скорее, они связаны с тем, как программное обеспечение реагирует на хаотичный мир сред, в которых запускается приложение. Поэтому необходимо производить тестирование производительности.

Обычное QA тестирование заключается в наблюдении, как приложение ведет себя с одним человеком.

Для измерения производительности необходима симуляция суровых условий, поэтому проводятся нагрузочное тестирование, стресс-тестирование и тестирование на выносливость.

Нагрузочное тестирование

В тестовой среде можно выбрать нагрузку для приложения, например, одновременное использование приложения тысячью пользователями при выполнении обычных операций и измерение поведения приложения. Такое тестирование не будет проводиться созданием реальных пользователей. Для этого создаете программное обеспечение, которое помогает имитировать нагрузку.

Стресс-тестирование

При стресс-тестировании приложению создаются неблагоприятные условия, чтобы увидеть, как он себя ведет в этих ситуациях. Это позволяет понять, в какой момент оно сломается и какие узкие места у приложения имеются. Каждое приложение имеет точку прерывания, поэтому после отказа на стресс-тесте можно вносить изменения, зная что искать и когда ожидать проблем, и сделать все возможное, чтобы в тот момент, когда приложение терпело неудачу, оно делало это изящно и разумно.

Тестирование на выносливость

При тестировании на выносливость применяется нагрузка на определенное время. Так же, как необходимо знать, как ведет себя приложение с большим количеством пользователей, необходимо знать, как он будет вести себя в течение нескольких недель или месяцев. И лучше это узнать в тестовой среде, чем в режиме реального времени.

Тестирование производительности лучше производить в процессе разработки приложения, так как есть вероятность обнаружить сбои, вызванные архитектурными решениями.

Перед началом тестирования производительности необходимо понять производственные условия работы приложения.

Необходимо выяснить нормальные и пиковые условия для подготовки к различным видам тестирования производительности. Тестирование на производительность будет включать нагрузочное тестирование, стресс-тестирование, тестирование на выносливость и другие. При тестировании нагрузки выбирается условия производства (например, количество пользователей, объем трафика и прочее). Затем имитируется эта нагрузку, чтобы увидеть, как система обрабатывает ее. Также можно увеличить

нагрузку до точки взлома, чтобы увидеть, в какой момент это происходит, и как эту проблему можно решить.

После определения условия тестирования приложения, первая задача будет заключаться в настройке среды для тестирования.

Необходимо симулировать производство тестирование не только на машине разработчика в меру своих возможностей. Это может означать более жесткие серверы, чем те, что стоят в компани.

После настройки тестового окружения необходимо выяснить, как имитировать условия производства для нагрузочных и стресс-тестов. При проведении тесты производительности, необходим запись результатов, пропуски и сбои для создания статистики.

Традиционная проверка при обычном тестировании довольно проста в некотором смысле. «Когда мы вводим X, программа должна вернуть Y.»

При тестировании производительности все гораздо сложнее. Так как нет истинных и ложных результатов, а скорее есть границы нормальной работы приложения и рекомендации. Например, можно сказать, что существует какоелибо допустимое максимальное время ответа приложения. Зачастую при начале проведения тестирования производительности можно и не знать, что ожидать, но в ходе работы необходимо протестировать производительность приложения как на клиенте (в браузере), так и на сервере.

По этой причине необходимо установить исходные данные. Проверяя, как приложение работает как есть, собирать данные, задав какие-то базовые предполагаемые данные и исправлять приложение до тех пор, пока оно не будет им соответствовать. И далее запускать тесты, изменяя исходные данные, создавая более неблагоприятные условия, исправлять приложение и так далее [2, 3].

1.3. Инструменты для тестирования производительности

1.3.1. Инструменты для тестирования производительности на стороне сервера

Apache Bench and Siege отлично подходят для тестов быстрой нагрузки с одного конечного пункта. Если необходимо просто получить представление о запросах в секунду для конечной точки, это отличные решения.

Locust.io - платформа тестирования нагрузки на стороне сервера. с открытым исходным кодом, которая позволяет выполнять сложные транзакции и может с легкостью генерировать высокий уровень параллелизма.

Bees with Machine Guns - авторы описывают, как «утилиту для создания большого количества экземпляров микро EC2 для нагрузочного тестирования веб-приложения.

Multi-Mechanize - это среда с открытым исходным кодом для тестирования производительности и нагрузки, которая запускает параллельные сценарии Python для генерации нагрузки (синтетических транзакций) в отношении удаленного сайта или службы. Она обычно используется для тестирования производительности сети и масштабируемости, но ее также можно использовать для создания рабочей нагрузки для любого удаленного API, доступного из Python.

Siege - эта утилита тестирования и тестирования производительности HTTP была разработана, чтобы позволить веб-разработчикам измерять код под нагрузкой, чтобы увидеть, как он будет себя вести при нагрузке в Интернете. Siege поддерживает базовую аутентификацию, файлы cookie, протоколы HTTP и HTTPS и позволяет пользователю набирать веб-сервер с настраиваемым количеством имитируемых веб-браузеров.

Apache Bench - инструмент для тестирования HTTP-сервера Apache, чтобы получить представление о том, как работает Apache.

Httperf - этот инструмент измеряет производительность веб-сервера и обеспечивает гибкую генерацию разнообразных рабочих нагрузок HTTP и производительности сервера.

производительности сервера.

JMeter - для проверки производительности как на статических, так и на динамических ресурсах (файлы, сервлеты, скрипты Perl, объекты Java, базы данных и запросы, FTP-серверы и т. д.). Также его можно использовать для симуляции большой нагрузки на сервер, сеть или объект, чтобы проверить приложение прочность или проанализировать общую производительность при разных типах нагрузки [4].

1.3.2. Инструменты для тестирования производительности на стороне клиента

Современные приложения тратят больше времени на стороне клиента, чем на сервере.

Google PageSpeed Insights - служба, которая анализирует содержимое веб-страницы и генерирует рекомендации, для более быстрой загрузки страниц. Сокращение времени загрузки страницы снижает частоту отказов и увеличивает коэффициент конверсии.

Sitespeed.io - инструмент для оценки производительности на клиентской стороне из реальных браузеров. Этот инструмент с открытым исходным кодом анализирует скорость и производительность веб-сайта на основе лучших практик и показателей времени.

Разработчики не всегда могут изменить приложения для оптимизации производительности на стороне клиента. Google инвестировал в создание ngx_pagespeed и mod_pagespeed в качестве расширений веб-серверов для автоматизации повышения производительности без необходимости изменения кода.

Google ngx_pagespeed серверный модуль nginx с открытым исходным кодом и Google mod_pagespeed сервер HTTP с открытым исходным кодом Apache ускоряют работу сайта и сокращают время загрузки страницы, применяет лучшие методы веб-производительности для страниц и связанных с ними файлов (CSS, JavaScript, изображений), не требуя изменения существующего контента или рабочего процесса.

WebPagetest.org - обеспечивает глубокое понимание производительности клиентской стороны в различных реальных браузерах. Эта утилита проверяет веб-страницу в любом браузере, из любого места, по любому сетевому условию [5].

2. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ВЕБ-ПРИЛОЖЕНИЙ

2.1. Показатели производительности веб-приложений

Исследование производительности веб-приложение можно проводить по следующим показателям:

1. время на редирект;
2. продолжительность соединения;
3. время работы сервера;
4. время до первого байта;
5. время первой отрисовки;
6. загрузка DOM (содержимого страницы - контента);
7. время до интерактива (когда пользователь может начать взаимодействовать со страницей);
8. время полной загрузки.

Когда выполняется запрос для страницы, Front-end и Server-side компоненты затрачивают определенное количество времени для выполнения своих операций. Поскольку эти операции по существу последовательны, их суммарное время можно считать общим временем загрузки страницы.

Рассмотрим подробнее каждый из пунктов.

Время на редирект

Время на редирект- это время, затраченное на перенаправление URL-адресов до загрузки последней HTML-страницы.

Общие переадресации включают:

- перенаправление URL-адреса, начинающегося не с www, на www (например, example.com на www.example.com);
- перенаправление на защищенный URL (например, http: // в https: //), - перенаправление для установки файлов cookie;
- перенаправление на мобильную версию сайта.

Некоторые сайты могут также выполнять цепочку из нескольких переадресаций (например, сначала на www, а затем на безопасный URL-адрес). Время на редирект - это общее количество времени, потраченное на перенаправление, оно равно 0, если не было переадресации.

Переадресация состоит из времени с начала произведения измерения до момента, когда мы начнем запрос последней HTML-страницы (когда мы получим первый ответ 200 OK). До этого времени экран браузера пуст. Поэтому очень важно следить за сокращением времени на редирект страницы, сводя к минимуму переадресации.

Продолжительность соединения

Как только все переадресации завершены, измеряется продолжительность соединения. Это время, потраченное на подключение к серверу, для выполнения запроса на страницу.

С технической точки зрения, эта продолжительность представляет собой комбинацию заблокированного времени, времени DNS, времени соединения и времени отправки запроса (а не просто времени соединения).

Продолжительность подключения состоит из всего этого, включая время отправки в окончательном запросе HTML-страницы (первый ответ 200 OK).

Все это время экран браузера также остается пустым. Этому могут способствовать различные причины, в том числе медленная связь между тестовым сервером и сайтом или медленное время отклика с сайта.

Время работы сервера

Как только соединение будет завершено и будет выполнен запрос, серверу необходимо сформировать ответ для страницы. Время, затрачиваемое на генерацию ответа, называется временем работы сервера.

Время работы сервера состоит из времени ожидания в запросе страницы.

Существует ряд причин, по которым продолжительность работы сервера может быть медленной. Даже после того, как оптимизирована

производительность на стороне Front-end, необходима оптимизация и серверной части. Это означает оптимизацию генерации страницы сервером (время до первого байта). Как правило, это время должно быть в пределах одной секунды (или как можно меньше).

Причины медленной работы серверной стороны по существу могут быть сгруппированы в две категории:

- неэффективный код или SQL;
- узкие места / медленный сервер.

Поскольку каждый сайт имеет уникальную платформу и настройку, решение этих проблем для каждого сайта своя. Возможно, одному сайту необходимо оптимизировать серверный код, а другому может потребоваться более мощный сервер. Также важны бюджетные ограничения, таким образом оптимизация кода на стороне сервера для незначительного увеличения скорости может быть более доступной, чем обновление серверов для увеличения скорости.

Время до первого байта (TTFB)

Время до первого байта (TTFB) - это общее количество времени, затрачиваемого на получение первого байта ответа после выполнения запроса. Это сумма «Длительность редиректа + «Длительность соединения + «Длительность работы сервера». Этот показатель является одним из ключевых показателей производительности сети.

Некоторые способы улучшения TTFB включают в себя: оптимизацию кода приложения, реализацию кэширования, точную настройку конфигурации веб-сервера или обновление серверного оборудования.

Время первой отрисовки

Время первой отрисовки - это первый момент, когда браузер делает какой-либо рендеринг на странице. В зависимости от структуры страницы эта

может быть просто цвет фона (включая белый), или это может быть большая часть отображаемой страницы.

Это время имеет важное значение, поскольку до этого момента браузер будет показывать только пустую страницу, и это изменение дает пользователю указание, что страница загружается. Однако мы не знаем, на скольких страницах произошла первая отрисовка, поэтому наличие ранней первой отрисовки не обязательно.

Загрузка DOM

Загрузка DOM - это момент, когда браузер завершил загрузку и анализ HTML, и была создана DOM (Document Object Model). DOM - это то, как 18 браузер структурирует HTML, чтобы он мог его отображать. Загрузка DOM очень близка ко времени загрузки содержимого DOM.

Время до первого интерактива

Или время загрузки содержимого DOM (DOM загружено или DOM готово для краткости) - это точка, в которой DOM готов (т. е. DOM interactive), и нет таблиц стилей, блокирующих выполнение JavaScript.

Если нет таблиц стилей, блокирующих выполнение JavaScript, и нет синтаксического анализатора, блокирующего JavaScript, то это будет то же самое, что и интерактивное время DOM.

Многие фреймворки JavaScript используют это событие в качестве отправной точки для начала выполнения своего кода.

Поскольку это событие часто используется JavaScript в качестве отправной точки, а задержки в этом событии означают задержки в рендеринге, важно убедиться, что порядок стилей и сценариев оптимизирован и что разбор JavaScript отложен.

Время полной загрузки

Время загрузки - это когда обработка страницы завершена, и все ресурсы на странице (изображения, CSS и т. д.) завершили загрузку. Это также то же самое время, когда DOM готов и происходит событие JavaScript window.onload.

Необходимо обратить внимание, что также может быть JavaScript, который инициирует последующие запросы для получения большего количества ресурсов, следовательно для результатов для нас предпочтительнее время полной загрузки. [6]

Ниже приведен краткий обзор ключевых показателей эффективности:

- менее 100 миллисекунд воспринимается как мгновенный ответ;
- от 100 мс до 300 мс - наблюдается незначительная задержка;
- одна секунда - это предел для того, чтобы поток мысли пользователя оставался непрерывным;
- пользователи ожидают, что сайт загрузится через 2 секунды;
- через 3 секунды 40% посетителей покинут ваш сайт;
- 10 секунд - это ограничение для поддержания внимания пользователя.

2.2. Стоимость JavaScript

Рассмотрим следующий пример страницы:

```
<!DOCTYPE html>
 <html>
  <head>
   <link rel="stylesheet" href="/styles.css">
   <script src="/app.js" async></script>
  </head>
  <body>
   <my-app>
    <picture slot="hero-image">
      <source srcset="img@desktop.png, img@desktop-2x.png 2x" media="(min-width:
990px)">
      <source srcset="img@tablet.png, img@tablet-2x.png 2x" media="(min-width:
750px)">
```

```
    <img srcset="img@mobile.png, img@mobile-2x.png 2x" alt="I don't know why. It's
a perfectly cromunlent word!">
   </picture>
  </my-app>
 </body>
</html>
```

Браузер получает этот документ от сервера в ответ на GET запрос.

Сервер отправляет его как поток байтов, и когда браузер сталкивается с каждым из подресурсов, на который ссылается документ, он запрашивает их.

Чтобы эта страница была загружена, она должна стать интерактивной, то есть на нужно «Время до интерактива». Браузеры обрабатывают ввод пользователя, генерируя события DOM, которые прослушивает код приложения. Эта обработка ввода происходит в основном потоке документа, где выполняется JavaScript.

Существует ряд операций, которые могут проходить в других потоках, позволяя браузеру оставаться отзывчивым:

- Анализ HTML
- Разбор CSS
- Анализ и компиляция JavaScript (иногда)
- Некоторые задачи сборки мусора JS
- Декодировние изображений
- Преобразования и анимации CSS
- Прокрутка основного документа

Следующие же операции должны выполняться в основном потоке:

- Выполнение JavaScript
- Построение DOM
- Создание макета
- Обработка ввода

Выполнение скриптов задерживает интерактивность несколькими способами:

Если сценарий выполняется более 50 мс, то время до интерактива задерживается на все время, необходимое для загрузки, компиляции и выполнения JS.

Любой DOM или пользовательский интерфейс, созданный в JS, недоступен для использования до запуска скрипта.

С другой стороны, изображения не блокируют основной поток, не блокируют взаимодействие при анализе или растрировании и не препятствуют тому, чтобы другие части пользовательского интерфейса оставались интерактивными. Поэтому, очень важно понимать, что 150 КБ JS будет задерживать время до интерактива из-за:

- Запроса кода, включая DNS, TCP, HTTP и накладные расходы на декомпрессию
- Парсинг и компиляцию функций верхнего уровня JS
- Выполнение скрипта

Исходя из проводимой статистики 45% мобильных подключений приходится на 2G по всему миру, а 75% соединений происходит на 2G или 3G.

Медианный пользователь находится в медленной сети.

В Google Chrome DevTools при включении тротлинга на медленном соединении 3G - иммитривется связь с пропускной способностью 400 мс RTT это скорость 400-600 кбит/с - таким образом можно принять эти показатели за базовые.

В современном мире разработки существует некий метрический показатель, в пределах которого должно находиться время до интерактива:

- TTI менее 5 секунд для первой нагрузки;
- TTI менее 2 секунд для последующих нагрузок.

Исходя из этого можно сделать следующий расчет: поиск DNS и авторизации TLS составляет около 1,6 секунды, а это значит, что для загрузки страницы остается около 3,4 секунд Затем можно вычислить, сколько данных

можно отправить по этой ссылке за 3.4 секунды: 400 Кбит/с = 50 КБ/с. Значит 50 КБ/с * 3,4 = 170 КБ.

Подводя итог, можно сказать, что в среднем бюджет для ресурсов, передаваемых по сети (CSS, JS, HTML и данные) должен составлять 170 КБ, если же разработчики используют js-framework, то бюджет снижается до 130 КБ, так как инфраструктура как правило занимает порядка 40 КБ.

Снизить стоимость передачи JavaScript по сети можно следующими способами (рисунок 1):

- Отправление только того кода, который нужен пользователю. В данной ситуации может быть очень полезно разделение кода.
- Минимизация (Uglify для ES5, babel-minify или uglify-es для ES2015)
- Сильное сжатие (с использованием Brotli ~ q11, Zopfli или gzip). Brotli превосходит gzip по степени сжатия. Компании CertSimple удалось сократить на 17% JS файлы, а LinkedIn сэкономить 4% на времени загрузки.
- Удаление неиспользуемого кода. Определить неиспользуемый код можно с помощью DevTools code coverage. Для удаления неиспользуемого кода можно использовать tree-shaking, Closure Compiler’s и библиотеки, такие как lodash-babel-plugin или ContextReplacementPlugin Webpack для библиотек, таких как Moment.js. Также рекомендуется использование babel-preset-env & browserlist в современных браузерах. Также необходимо проводить анализ Webpack bundle для выявления возможностей удалить неиспользуемые зависимости.
- Кэширование JS файлов, для минимизирования количества сетевых запросов. Также необходимо определить оптимальные сроки жизни для скриптов (max-age) и токенов проверки поставки (ETag), чтобы избежать передачи неизмененных байтов. Кэширование с помощью Service

Worker’ов может сделать сеть приложений устойчивой и предоставит быстрый доступ к таким функциям, как кеш-код V8.

Рис. 1. Рекомендации по сокращению количества JavaScript

Парсинг/Компиляция

После загрузки одно из самых больших - это время, когда JS-движок парсит/компилирует JavaScript код.

В панели Chrome DevTools Performance > Bottom-Up. С включенной статистикой Call Runtime V8 (рисунок 2) мы можем увидеть время, потраченное на такие фазы, как парсинг и компиляция.

На рисунке 3 приведено суммарное соотношение затраченного времени на разбор полученных данных.

Большое время парсинга/компиляции кода могут сильно влиять на то, как скоро пользователь сможет взаимодействовать с сайтом. Чем больше JavaScript кода отправляется, тем больше времени необходимо для его парсинга и компиляция, то есть до момента, когда сайт станет интерактивным.

В сравнении с JavaScript также множество затрат на обработку изображений с эквивалентным размером (так как их еще нужно декодировать),

но в среднем требуется больше времени для обработки javascript кода, чем для загрузки изображений.

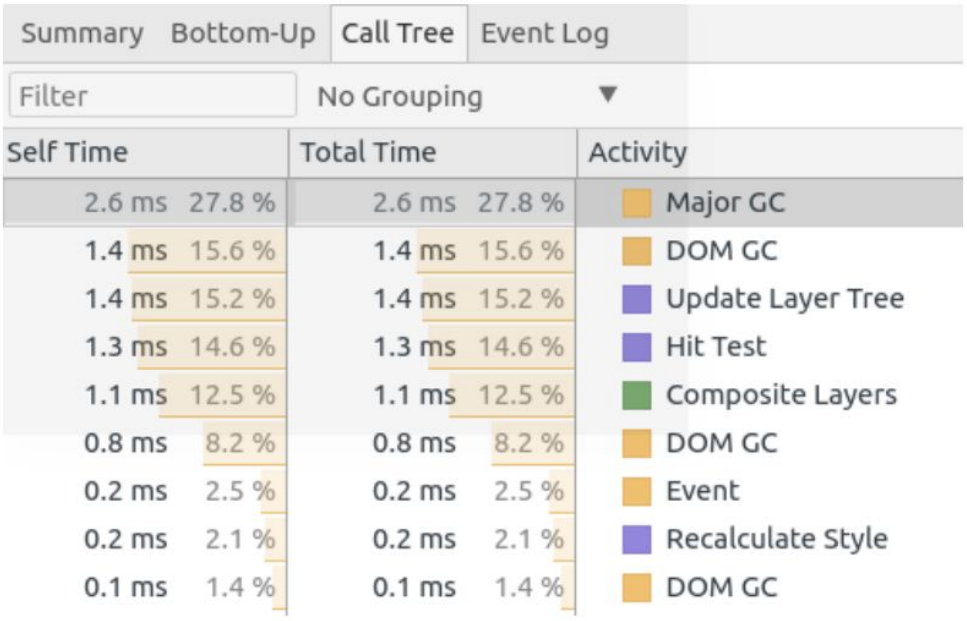

Рис. 2. Время на парсинг / компиляцию панели Performance вкладки The Bottom- Up/Call Tree для главной страницы сайта

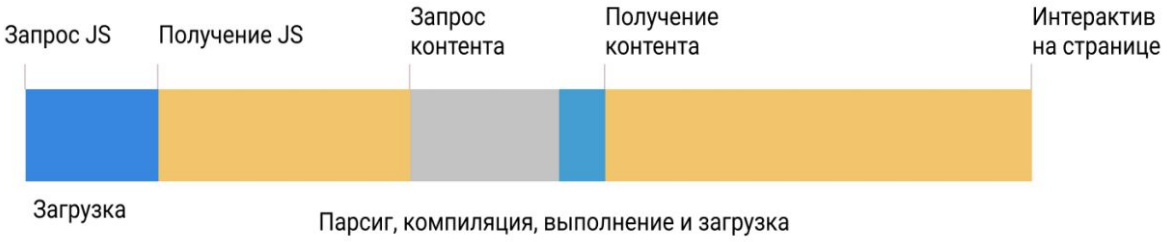

Рис. 3. Соотношение времени на загрузку и парсинг/компиляцию страницы

Очень важно понимать, что байты JavaScript и байты изображения имеют очень разные затраты. Обычно изображения не блокируют основной поток или не препятствуют взаимодействию с интерфейсом при декодировании и растрировании. Однако JS может задерживать интерактивность из-за временных затрат на парсинг, компиляцию и выполнение.

Долгий парсинг и компиляции очень важны, когда речь идет о средних мобильных телефонах. В среднем у пользователей могут быть телефоны с

медленными процессорами и графическими процессорами, без кеша L2 / L3, которые также могут иметь ограниченную память.

“Возможности сети и возможности устройства не всегда совпадают. Пользователь, у которого очень хорошее соединение не обязательно имеет лучший процессор для парсинга и компиляции JavaScript, отправленного на его устройство. И наоборот, плохое сетевое соединение, но быстрый процессор.” - Кристофер Бакстер, LinkedIn.

На рисунке 4 отмечена стоимость разбора ~ 1 МБ декомпрессированного (простого) JavaScript на низком и высоком уровне аппаратного обеспечения. Можно отметить разницу в 2-5 раз для парсинга / компиляции кода между самыми быстрыми телефонами на рынке и средними телефонами.

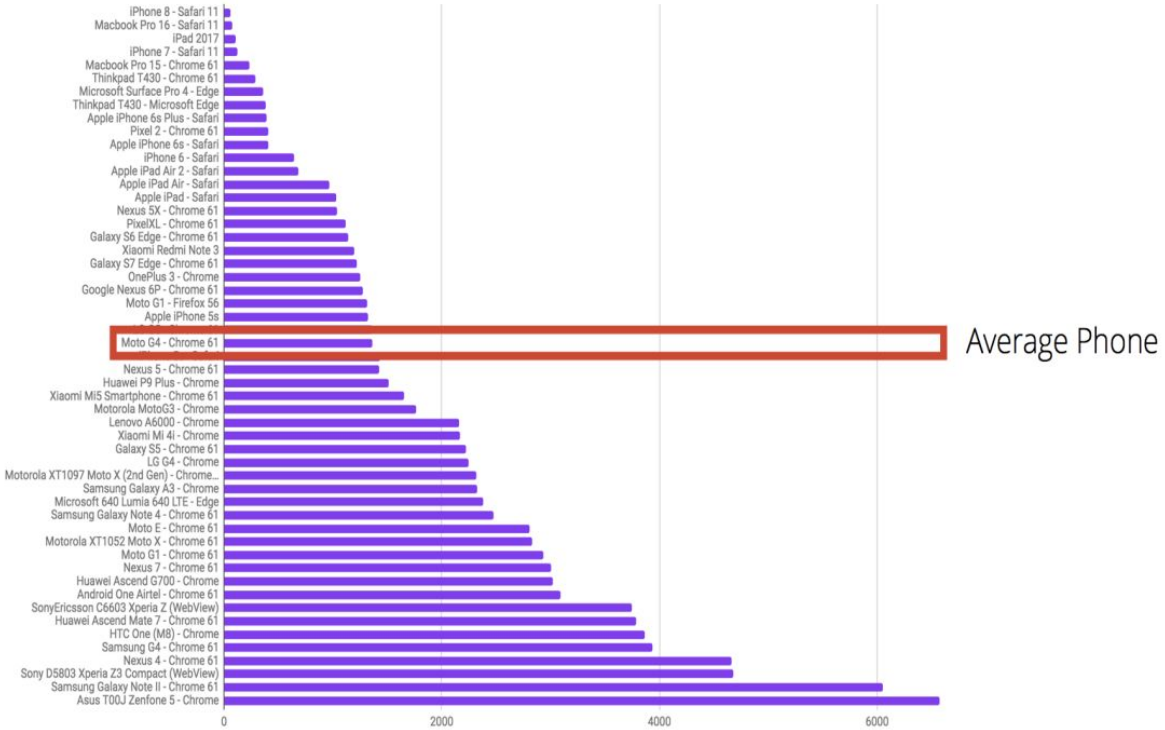

Рис. 4. Время парсинга 1 МБ bundle JavaScript (~ 250 КБ gzipped) на десктопный и мобильных устройствах разных классов по состоянию на 2018 год.

Рассматривая стоимость парсинга на примере декомпрессированных значений, например, ~ 250 КБ gzipped JS, необходимо иметь в виду, что в распакованном виде размер может быть до 1 Мбайт кода.

Правильнее всего проводить тестирования на среднем оборудовании, так как время парсинга и компиляции кода может быть в разы больше на смартфоне со средней производительностью, чем на последней версии iPhone. Аналитика может дать представление о классах мобильных устройств, с помощью которых пользователи обращаются к сайту. Это может предоставить возможности для понимания реальных ограничений ЦП/ГПУ, с которыми они работают.

В HTTP-архиве (более 500 тыс. Сайтов) для анализа загрузки JavaScript на мобильных устройствах, можно увидеть, что около 50% сайтам необходимо более 14 секунд до интерактивности (рисунок 5). Эти сайты тратят до 4 секунд лишь на парсинг и компиляцию JavaScript. Поэтому необходимо стремиться сократить время загрузки сайтов [7].

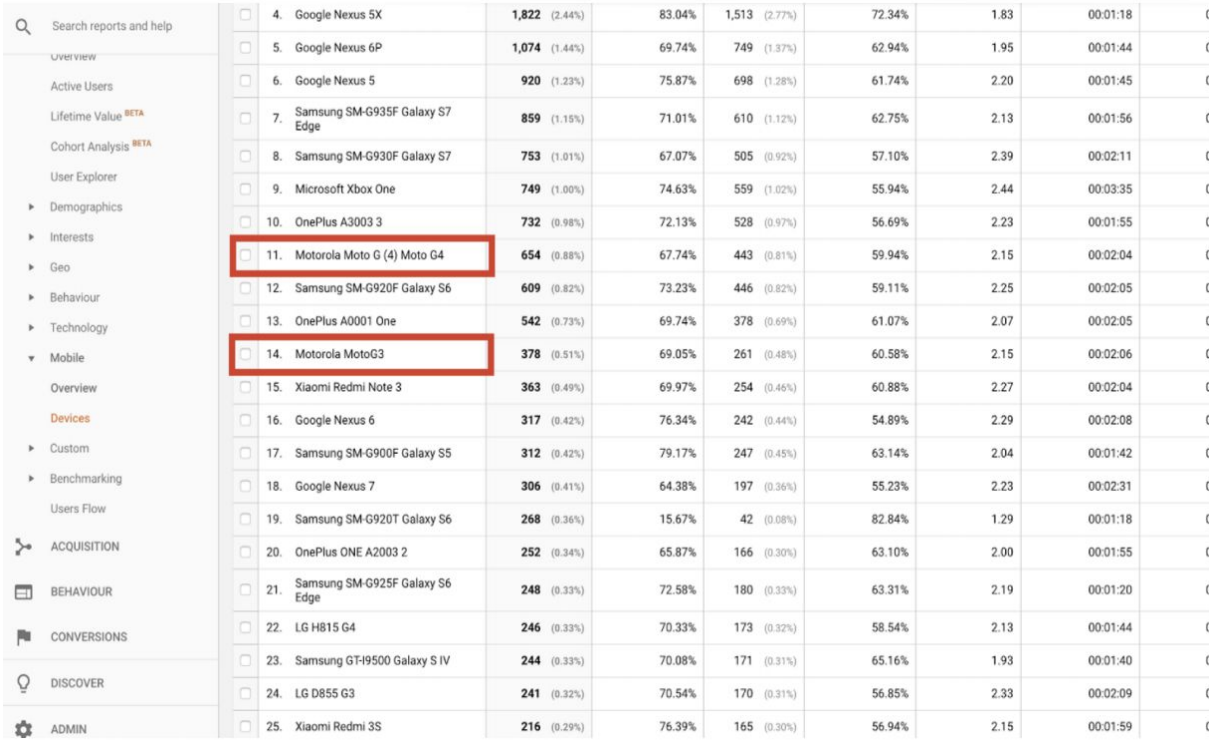

Рис. 5. Статистика времени до интерактива на мобильных устройствах на основе HTTP-архива

Удаление некритичного JavaScript может сократить время на передачу кода, парсинг и компиляцию, а также потенциальные затраты памяти.

Время исполнения

Это не просто парсинг и компиляция кода, которые влияют на стоимость. Выполнение JavaScript (запуск кода парсинг / компиляции) является одной из операций, которая выполняется в основном потоке. Длительное время выполнения также влияет на то, как скоро пользователь может взаимодействовать с сайтом.

Чтобы решить эту проблему, необходимо разбивать JavaScript код на небольшие фрагменты, чтобы избежать блокировки основного потока.

Для того, чтобы сохранить время парсинга/компиляции и времени передачи JavaScript кода, существуют паттерны, которые могут помочь в решении этих задач, например, route-based chunking или PRPL.

PRPL - это паттерн, который оптимизирует интерактивность посредством агрессивного кодового разделения и кеширования.

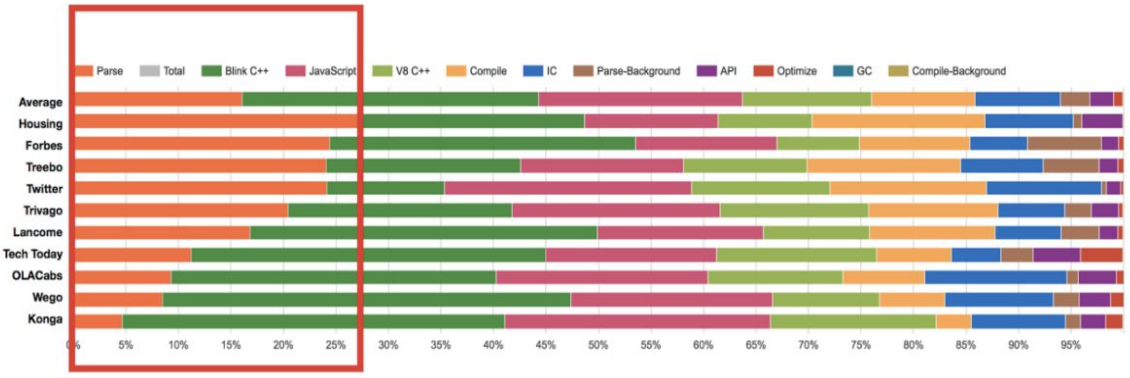

Рис. 6. Время загрузки популярных мобильных сайтов

На рисунке 6 приведенном выше отражено время загрузки популярных мобильных сайтов и прогрессивных веб-приложений с помощью V8 Runtime Call Stats. Как можно заметить, время парсинга (показано оранжевым цветом) занимает значительную часть времени, которое затрачивают многие из сайтов:

Wego (https://wego.com/), сайту, который использует PRPL, удается поддерживать низкое время на парсинг для своих маршрутов, быстро получая интерактивность. Многие из вышеперечисленных сайтов

Другие расходы

JavaScript может влиять на производительность страницы другими способами:

- Утечка памяти. Страницы могут часто приостанавливать свою работу, то есть терять свою интерактивность из-за сборщика мусора (garbage collector), что очень заметно в быстро реагирующих приложениях. Основным недостатком сборщиков мусора является недетерминированность, то есть сложно понять в какой момент может произойти сборка мусора. Когда браузер восстанавливает память, выполнение JS приостанавливается, поэтому браузер, часто собирающий мусор, может приостанавливать выполнение чаще, чем может быть удобно пользователям. Необходимо избегать утечек памяти и частых пауз сборщиков мусора, чтобы страницы не теряли свою интерактивность. Существует четыре самых распространенных вида утечек памяти JavaScript: случайные глобальные переменные, забытые коллбэки и таймеры, ссылки на удаленные из DOM элементы, замыкания [9].
- Во время запуска, JavaScript с долгим временем выполнения кода может блокировать основной поток, делая недоступными страницы. Разбивка кода на более мелкие части (с использованием requestAnimationFrame() или requestIdleCallback() для планирования) может минимизировать проблемы с откликом сайта [8].

ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены способы оптимизации веб-приложений. Изучены инструменты для измерения производительности и результатов оптимизаций. Приведены основные уязвимости в производительности клиент-серверных приложений и основные источники расходов ресурсов

Используя результаты, полученные в данной работе, можно приступать к разработке веб-приложения и к исследованию производительности на его основе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. О модели взаимодействия клиент-сервер простыми словами. Арзитектура клиент-сервер с примерами. URL: https://zametkinapolyah.ru/servera-i-protokoly/o-modeli-vzaimodejstviya-klient-server-prostymi-slovami-arxitektura-klient-server-s-primerami.html (дата обращения: 25.11.2019)
2. Fundamentals of Web Application Performance Testing. URL: https://msdn.microsoft.com/en-us/library/bb924356.aspx (дата обращения: 26.12.2019).
3. Fundamentals of Web Application Performance Testing. URL: https://stackify.com/fundamentals-web-applicationperformance-testing (дата обращения: 12.12.2019)
4. Web performance testing: Top 12 free and open source tools to consider. URL: https://techbeacon.com/web-performancetesting-top-12-free-open-source-tools-consider (дата обращения: 21.11.2019).
5. Client Side Performance Testing. URL: http://blog.dataart.com/client-side-performance-testing/ (дата обращения: 21.11.2019).
6. Understanding Resource Timing. URL: https://developers.google.com/web/tools/chrome-devtools/networkperformance/understanding-resource-timing (дата обращения: 21.11.2019).
7. HTTP archive. URL: http://httparchive.org/ (дата обращения: 21.11.2019).
8. Цена JavaScript. URL: https://habr.com/en/post/343562/ (дата обращения: 20.12.2019).
9. Garbage collector. URL: https://habrahabr.ru/post/309318/ (дата обращения: 20.12.2019).