Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»

Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки / специальность: 03.04.01 Прикладные математика и физика

Направленность (профиль) подготовки: Математические и информационные технологии

Форма обучения: очная

Допущен к ГИА

Руководитель учебного подразделения

_____/_____

(подпись) *расшифровка подписи*

«___» _____ 2018 г.

РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА НА ЕДИНОЙ КОДОВОЙ БАЗЕ ДЛЯ WINDOWS-ПРИЛОЖЕНИЙ И ВЕБ-ПРИЛОЖЕНИЙ

(магистерская диссертация)

Обучающийся: Кошечкин Сергей Вениаминович

_____ (подпись обучающегося)

Научный руководитель: Кулага Андрей Александрович, к.ф.-м.н.

_____ (подпись научного руководителя)

Москва 2018

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ **2**

АННОТАЦИЯ **5**

ВВЕДЕНИЕ **6**

Актуальность темы исследования 6

Цель дипломной работы 6

Теоретическая значимость 6

Практическая значимость 6

Структура работы 7

ГЛАВА 1 Подходы для разработки на нескольких платформах **8**

Electron 8

Electron компоненты 9

Chromium 9

Node.js 11

V8 12

Процессы 14

Основной процесс 14

Процесс рендеринга 15

Progressive Web Applications 16

Safe 18

Progressive 18

Responsive 19

Connectivity independent 19

App-like 21

Fresh 22
Discoverable 22
Re-engageable 22
Installable 23
Linkable 23
Результаты 24
ГЛАВА 2 Архитектура Web-приложений 26
Model-View-Controller 26
Модель 26
View 27
Controller 28
Flux 28
Structure and Data Flow 30
ГЛАВА 3 ОПИСАНИЕ ПРИЛОЖЕНИЯ 33
Функциональные требования для приложения 33
Импорт из файла, используя Drag-and-Drop 33
Конвертация .mp4 → .gif 33
Предпросмотр 34
Экспорт результата на жесткий диск 34
Кросс-платформенность 34
Описание сборки приложения 34
Архитектура приложения 34
Описание запуска приложения 35
Результат 37
ЗАКЛЮЧЕНИЕ 39

СПИСОК ЛИТЕРАТУРЫ **41**

АББРЕВИАТУРЫ **43**

АННОТАЦИЯ

Данная работа посвящена разработке технологии создания кроссплатформенных приложений с динамическим формированием структуры и контента. Основные задачи включают в себя поиск и разработку оптимального паттерна проектирования мобильных приложений, на основе которого возможно построение архитектурного решения, позволяющего создание приложений на несколько платформ одновременно. Выбранное оригинальное решение основано на Electron.js с web-основой на Angular 6.

В работе приведен анализ существующих подходов к решению проблемы создания кроссплатформенных приложений, и, в конечном счете, предложена оригинальная архитектура разработанного приложения, предусматривающая создание нативных мобильных и десктопных кроссплатформенных приложений. Также проведен анализ существующих архитектурных решений для разработки Веб-приложений на основе этого

Результатами являются: разработанный подход создания нативных кросс-платформенных приложений с единой кодовой базой, созданное веб-приложение, которое позволяет конвертировать mp4 видео в GIF-изображения с возможностями экспорта и импорта используя Drag-and-Drop.

ВВЕДЕНИЕ

Актуальность темы исследования

Современное программное обеспечение должно запускаться на большом количестве различных платформ, у каждой из которых свои аппаратные особенности и возможности. Для того чтобы достичь максимального удобства для пользователя, многие компании разрабатывают продукт непосредственно под каждую платформу, тем самым увеличивая финансовые и временные затраты на разработку и поддержку в несколько раз.

Цель дипломной работы

Разработка пользовательского интерфейса на единой кодовой базе.

Теоретическая значимость

Приведен достаточно подробный анализ имеющихся решений и подходов, которые могут упростить и удешевить производство кросс-платформенных приложений.

На основе данных подходов выбран паттерн разработки, с помощью которого создано приложение являющееся доказательством и решением поставленной задачи.

Практическая значимость

На основе данных подходов выбран паттерн разработки, с помощью которого создано приложение являющееся доказательством и

решением поставленной задачи. Дополнительно это приложение автоматизирует процессы создания gif изображений из mp4 видео, что необходимо UI/UX команде Acronis.

Структура работы

Работа состоит из введения, трех глав и заключения. В первой главе дается общее представление о подходах разработки кросс-платформенных приложений. Вторая глава посвящена архитектуре Веб-приложений. В ней разбираются основные паттерны разработки, и выбирается самый оптимальный. В третьей главе рассказывается про реализованное приложение, реализованное на основе выводов из предыдущих глав.

ГЛАВА 1 Подходы для разработки на нескольких платформах

В данный момент существует несколько способов сделать мультиплатформенное приложение. Эти способы кардинально отличаются друг от друга. В данной работе было решено выбор сделать между Electron и PWA.

Electron

Electron - это платформа с открытым исходным кодом для разработки кросс-платформенных десктопных приложений с использованием JavaScript, HTML и CSS, как это определено официальной документацией Electron. Разработанная и поддерживаемая Github, она прошла долгий путь с 2015 года, когда то, что первоначально было разработано и называлось Atom Shell, было тогда названо Electron. В настоящее время популярность Electron процветает, а количество загрузок достигло 1,2 миллиона.

Когда речь идет о Electron, это значит говорить о трех его основных компонентах, на которых он основан: Node.js, Chromium и V8 - движок JavaScript. В следующих разделах мы обсудим Electron, его основные компоненты и структуру, чтобы дать более полное представление об изученном фреймворке и том, что он предоставляет приложениям.

Electron компоненты

Electron framework был реализован на основе трех основных компонентов: движка Node.js, Chromium и V8 JavaScript. Один экземпляр движка V8 используется как Node.js, так и Chromium.

Chromium

Chromium - это проект браузера с открытым исходным кодом, разработанный и поддерживаемый Chromium Project и являющийся основой браузера Google Chrome. Electron использует только библиотеку рендеринга от Chromium для поддержки окон.

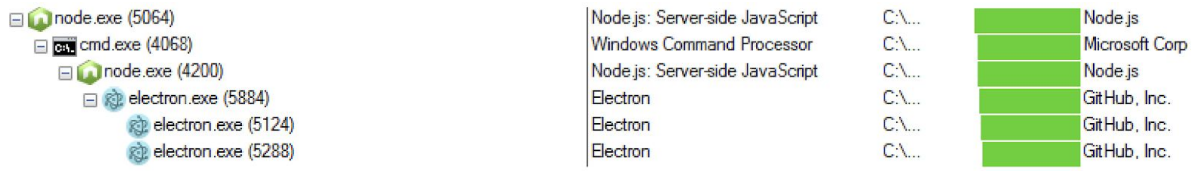
Модуль контента, используемый Electron, взят из репозитория Chromium, где он упакован. В данный момент находится в своем репозитории известном как **libchromiumcontent**.

Этот модуль содержит каждый фрагмент кода, необходимый для отображения страницы в многопроцессорном изолированном браузере. Однако модуль контента не включает функции Chrome, такие как автозаполнение, орфографию, расширения и другие, но реализует API, на которых эти функции могут быть построены.

Electron следует концепции многопроцессорной архитектуры, реализованной в Chromium, где один или несколько процессов визуализации запускаются из основного процесса. Каждый из процессов рендеринга может содержать один или несколько объектов рендеринга представления. Это аналогично браузеру Chrome и на управление вкладками в окне браузера. Каждый процесс визуализации имеет доступ к связи с его родительским процессом, будучи изолированным от других процессов рендеринга. Пример создания

процесса можно показать, запустив приложение «Быстрый старт» Electron и наблюдая созданные процессы через Process Monitor, как показано на рисунке.

Простое приложение, такое как «Быстрый старт», отображающее текст и версии Node.js, Chromium и Electron, создаст следующие процессы:



Безопасность является одной из самых важных целей для Chromium, но, следуя выполненным командам, очевидно, что по умолчанию Electron запускается в оболочке Chromium без включенной функции песочницы. Это также позволило использовать интеграцию с Node и использование тегов Webview, которые соответственно обеспечивают доступ к Node API и встраивание стороннего контента. Поэтому содержимое, отображаемое в приложении, имеет доступ к операционной системе и ее нативным API-интерфейсам для выполнения таких операций, как чтение и запись файлов.

Electron, используя веб-технологии, сталкивается с такими же проблемами безопасности, как и любое веб-приложение, работающее внутри браузера Chrome, но воздействие атаки может быть гораздо сильнее. В изолированной среде, такая уязвимость как межсайтовый скриптинг встречается, но все же имеет большое влияние. Он может позволить злоумышленнику выполнить захват сеанса, украв куки-файл сеанса пользователя, выполнить вредоносное перенаправление, получить доступ к конфиденциальным данным и даже выполнить

действия от имени жертвы. Без наличия изолированной среды последствия могут быть гораздо больше.

Node.js

С популярностью JavaScript, стабильное создание Node.js в мире программного обеспечения было неизбежным. Преодоление границ запуска JavaScript только на стороне клиента проложило путь для серверных приложений, а также для мобильных и настольных приложений.

Node.js - это среда исполнения JavaScript с открытым исходным кодом, которая использует механизм V8 для анализа, компиляции и запуска JavaScript. Поскольку Node.js является асинхронным, он позволяет обрабатывать многие операции одновременно. Он поставляется со своим собственным менеджером пакетов npm, который содержит наибольшее количество библиотек с открытым исходным кодом в мире. В декабре 2015 года количество модулей несколько превышало 200 000. В настоящее время примерно 700 000 модулей, в среднем 700 добавленных новых модулей в день, доказывают свою растущую популярность и принятие сообществом.

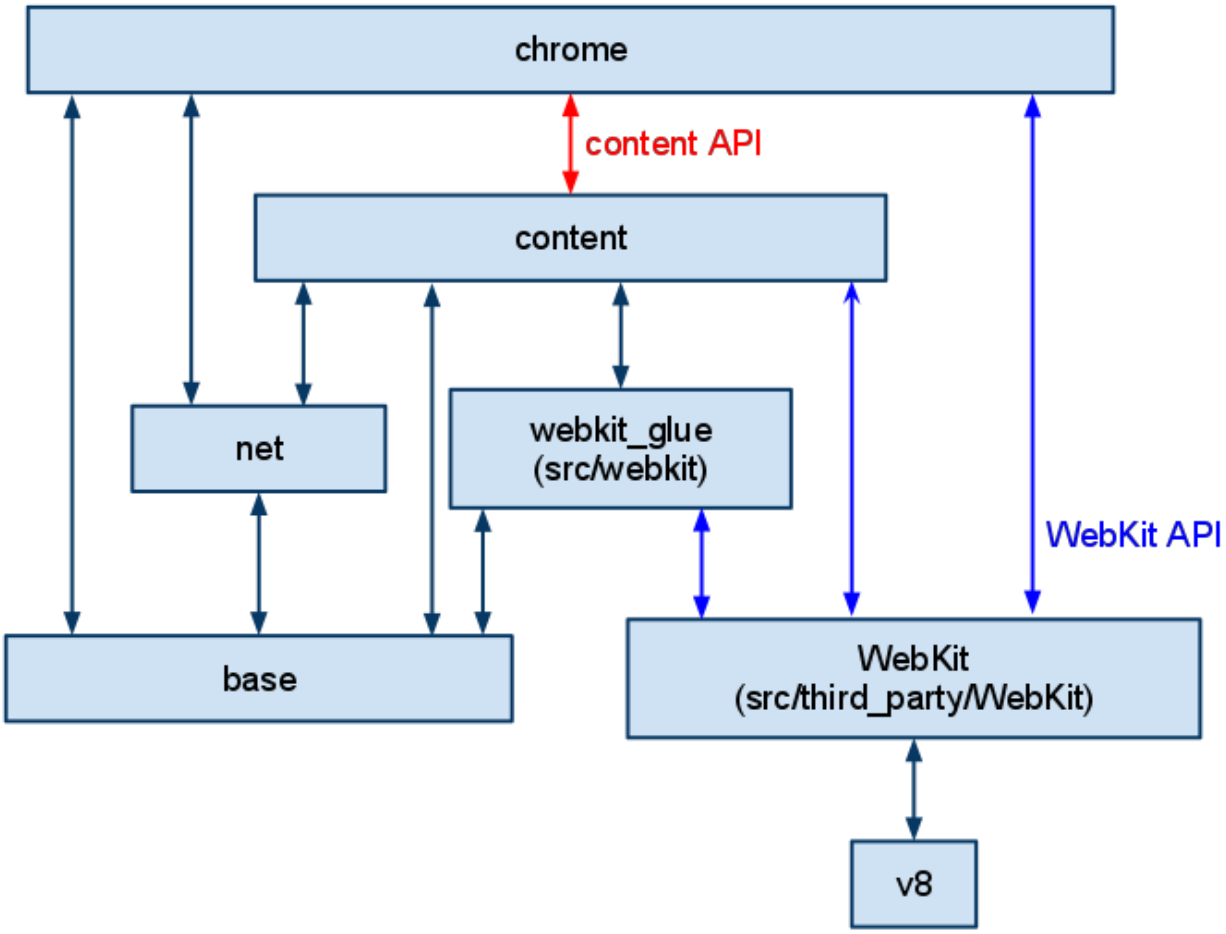
Electron сильно зависит от Node и возможностей, которые он обеспечивает для разработки. Он предоставляет приложение с интегрированным доступом к Node, чтобы использовать модули Node.js, которые затем могут быть использованы в приложении. Это включает в себя использование таких модулей, как файловая система, - для чтения/записи файлов, os - для использования связанных с операционной системой методов, например, для запроса информации о

ЦП, местонахождении домашнего каталога, имени хоста машины пользователя, количестве свободной памяти в системе, для получения IP-адресов и сетевых интерфейсов и т.д., path - для работы с файловыми и системными путями и многими другими модулями.

V8

V8 - механизм запуска JavaScript с открытым исходным кодом, используемый в браузере Chrome и в Node.js. Он делает это следующим образом: он принимает код JavaScript, анализирует его, создает абстрактное синтаксическое дерево (AST) и в конечном итоге приводит к генерации байт-кода, который будет обработан. V8 встроен в Node.js, который аналогичным образом написан на C++. Поскольку Node.js подключается к движку V8, он расширяет свои возможности, добавляя дополнительные функции, такие как чтение и запись файлов. В электронной структуре Chromium и Node.js используют один экземпляр V8.

Основным преимуществом использования V8 является компилятор JIT (Just in Time), который обеспечивает быстрое выполнение JavaScript и оптимизированный код на основе наблюдений, выполненных во время выполнения. Как упоминалось командой Electron, обновления версии V8 полагаются на версию V8, используемую для Chromium, которая затем исправлена для работы в Node, но это, вероятно, может измениться в соответствии с новостями, опубликованными фондом Node.js.

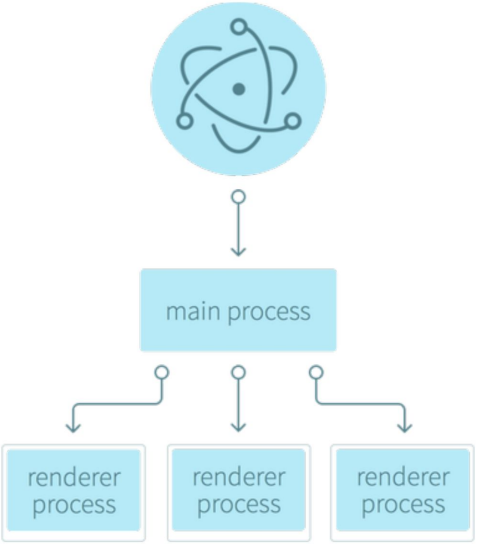


Процессы

Подобно Chromium, Electron использует архитектуру с несколькими процессами, в которой каждый из процессов работает одновременно и изолированно друг с другом. Он также реализует иерархию, в которой основной процесс может запускать один или несколько процессов рендеринга, в которых отображается веб-контент. В следующих двух подразделах «Основной процесс» и «Процесс рендеринга» будет даваться простой обзор каждого из них.

Основной процесс

В Electron основным процессом является процесс, который создает и управляет окнами браузера. Основной процесс запускается, когда выполняется файл, помеченный как точка входа в файле зависимостей. Имея контроль над созданием и управлением окнами браузера, основной процесс имеет право создавать несколько процессов рендеринга, как показано на рисунке.



Помимо управления окнами браузера, основной процесс также имеет доступ ко всем основным процессам Electron API, которые позволяют создавать меню, определять сочетания клавиш, запускать безрамные окна, загружать файлы управления и получать доступ к модулям Node.js.

Для связи с процессом рендеринга используется модуль IPCmain. Этот модуль способен обрабатывать асинхронные и синхронные сообщения, отправленные и полученные от средства визуализации.

Процесс рендеринга

Каждая страница в Electron работает в своем отдельном процессе, который называется процессом рендеринга. Несколько процессов рендеринга могут быть связаны с основным процессом. Основным процессом является то, что запускает и останавливает приложение, а файл зависимостей - это строка «main»: «main.js», обозначающая точку входа в приложение. Рисунок иллюстрирует создание окна браузера в main.js, где index.html загружается по указанному пути, как только приложение будет готово. Когда экземпляр webContents загружается в окно браузера, создается новый процесс визуализации.

```
const {app, BrowserWindow} = require('electron')
const url = require('url')
const path = require('path')

let win

function createWindow() {
      win = new BrowserWindow({width: 800, height: 600})
      win.loadURL(url.format ({
            pathname: path.join(__dirname, 'index.html'),
            protocol: 'file:',
            slashes: true
      }))
}

app.on('ready', createWindow)
```

Еще одна интересная нотация состоит в том, что в одном окне браузера может быть один или несколько экземпляров webContent, поэтому из одного окна могут быть созданы несколько процессов визуализации. Такая ситуация может возникнуть, например, когда webviews используются для отображения контента.

По аналогии с основным процессом, который имеет модуль для отправки сообщений в процесс рендеринга, есть модуль ipcRenderer. Он

может быть использован процессом визуализации для отправки и получения сообщений из основного процесса. Например, это позволяет процессу визуализации записывать в консоль, что было бы невозможно в противном случае.

Progressive Web Applications

PWA - это новый подход разработки Web приложений, предложенный Google. В основе него лежат обычные и всем известные технологии: HTML, CSS и Javascript. Но добавляя новые технологии поверх этой основы и следуя 10 основным концептам предложенными Google можно сделать передовое прогрессивное приложение.

- **Safe** – Использует HTTPS для предотвращения атаки “человек посередине” и целостности контента.
- **Progressive** – Работает для всех пользователей вне зависимости от выбора браузера в виду того, что приложение построено на основе прогрессивных улучшений.
- **Responsive** – Размещение на любом форм-факторе: десктоп, смартфон, планшет, часы, и любое другое еще не существующее.
- **Connectivity–independent** – Используя технологию Service Worker, дает доступ к возможностям приложения при отсутствии или при медленном интернет соединении.
- **App-like** – Ощущается как нативное приложение для пользователя с взаимодействиями в стиле приложения и навигации, потому что они построены на модели оболочки приложения.
- **Fresh** – Постоянно последней версии, благодаря процессу обновления, использующему SW.

- **Discoverable** – Является идентифицируемым как «приложение» благодаря манифесту W3C и регистрации Service Worker.
- **Re-engageable** – Делает возврат пользователей простым, используя Push уведомления.
- **Installable** – Разрешает пользователям сохранять приложение на рабочий стол без сложностей связанных с использованием магазина приложений.
- **Linkable** – Добавляет возможность делиться приложением с помощью обычной ссылки, не используя сложные процессы установки.

Данные концепты PWA будут подробнее разобраны в следующих секциях.

Safe

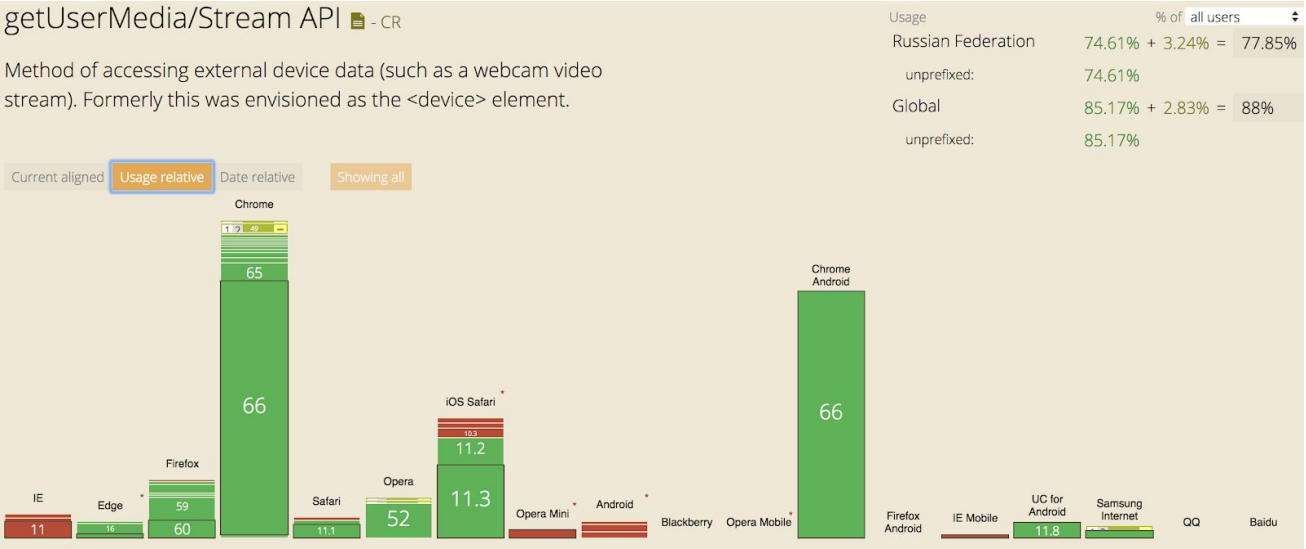
Поскольку прогрессивные веб-приложения могут передавать конфиденциальные данные через нативное API и Service Worker, рекомендуется использовать прогрессивные веб-приложения через HTTPS-соединение.

Progressive

Прогрессивность – это техника, используемая для улучшения контента и возможностей приложения, насколько это позволяет браузер или интернет-соединение, но основные функции или, по крайней мере, правильное сообщение об ошибке должны быть доступны для всех.

Ядром прогрессивного веб-приложения является новое API браузера Service Worker, который пока недоступен во всех браузерах, но приложение все равно может работать даже без него.

Другим примером нового API браузера является доступ к камере на устройстве через API getUserMedia. Чтобы все люди могли использовать эту функциональность приложения, необходимо охватить все конкретные префиксы API, используемые разными браузерами.



Responsive

Отзывчивый дизайн означает, что пользовательский интерфейс отображается таким образом, который соответствует размеру устройства. Приложение должно иметь гибкий дизайн и работать с каждым размером экрана. Отзывчивый дизайн очень важен, так как прогрессивные веб-приложения нацелены на мобильные устройства. Реагирующий дизайн может быть достигнут с использованием концепций плавающей сетки, гибких изображений и медиа запросов CSS3.

Connectivity independent

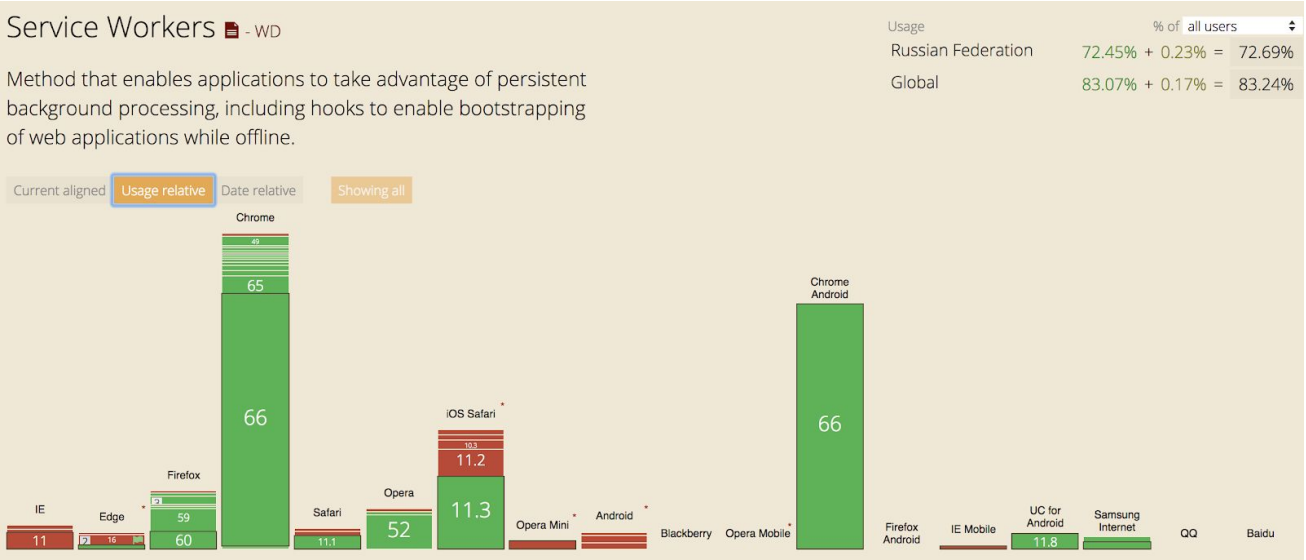
Технология Service Worker позволяет вашему приложению работать без подключения к Интернету. Service Worker - клиентский прокси-сервер, написанный на JavaScript, который помогает управлять кэшом разработчику и описывать стратегию реагирования на запросы ресурсов. Предварительно кэшируя основные ресурсы, приложение может устранить зависимость от сети, обеспечивая мгновенный и надежный UX для ваших пользователей.

Некоторые приложения по-прежнему в большой степени зависят от динамического контента онлайн от серверов, но даже эти приложения могут кэшировать элементы пользовательского интерфейса и быстрее запускать и даже отображать некоторые кэшированные данные из предыдущего взаимодействия.

Чтобы использовать Service Worker, разработчик должен создать файл JavaScript, который будет зарегистрирован как Service Worker и прослушать как минимум два события, “установить” и “извлечь” и реализовать основную логику.

С зарегистрированным Service Worker все файлы приложения могут быть сохранены в кэше браузера и доступны для пользователя, когда он вернется. Ответы с данными и переменными приложения могут быть сохранены в localStorage или IndexDB, где это возможно.

Service Worker - совершенно новая технология и еще не поддерживается всеми веб-браузерами, но уже имеет достаточную поддержку, чтобы использовать во внутренней разработке.



App-like

Концепцию дизайна, называемую архитектурой App-shell, рекомендуется использовать при построении пользовательского интерфейса Progressive web application. Эта концепция отделяет приложение от оболочки и содержимого.

Shell - это, по сути, все статические части приложения, необходимые для показа контента. Когда приложение кэшируется, необходимо кэшировать всю оболочку на устройстве. Кэшированная оболочка приложения гарантирует, что даже если нет подключения к Интернету, и приложение с кэшированным динамическим содержимым не загрузит хотя бы знакомый пользовательский интерфейс, и пользователю не будет предоставлен пустой экран или сообщение об ошибке соединения по умолчанию.

Содержимое отображается внутри оболочки. Сам контент также может быть кэширован, но он сильно зависит от типа приложения.

Fresh

Содержимое приложения и кэшированный контент всегда будут загружаться из локального хранилища после его кэширования. Однако браузер проверяет, был ли изменен файл Service Worker с последнего посещения, и если это так, новая версия Service Worker устанавливается для загрузки следующей страницы. Этот новый Service Worker может снова кэшировать все ресурсы и удалять старые кэшированные данные.

Discoverable

Использование Web Manifest позволяет разработчику предоставлять различную информацию о веб-приложении и дополнительно описывать, как оно должно отображаться. Web Manifest также позволяет устанавливать приложение на главный экран и открываться в отдельном окне браузера.

Web Manifest - это файл JSON, заданный в заголовке HTML-документа index страницы приложения.

Re-engageable

Прогрессивное приложение может использовать push-уведомления благодаря Service Worker, API Push и Notifications API. Благодаря этим API-интерфейсам веб-приложение может получать сообщения с сервера. Эти сообщения могут отображаться как уведомления пользователю и сообщать ему об обновлениях или запланированных событиях. Это упрощает повторное взаимодействие с пользователем и возвращает его в приложение.

Installable

Любой веб-сайт можно сохранить на главном экране Android/Windows/Mac с Chrome, используя кнопку «Добавить на главный экран» в меню. К сожалению, это не очень удобно.

Однако Chrome позволяет запрашивать у пользователя всплывающее окно предлагая ему сохранить приложение на главном экране. Это всплывающее окно недоступно разработчику, но браузер автоматически отобразит всплывающее окно, если приложение выполнит три следующих требования.

Веб-манифест должен быть представлен и соответствовать спецификации. Приложение должно иметь действительный и установленный Service Worker, а приложение должно обслуживаться через HTTPS.

Если пользователь подтверждает диалог, приложение немедленно сохраняется на главном экране и может быть удобно доступно в следующий раз, когда пользователь захочет его открыть.

Linkable

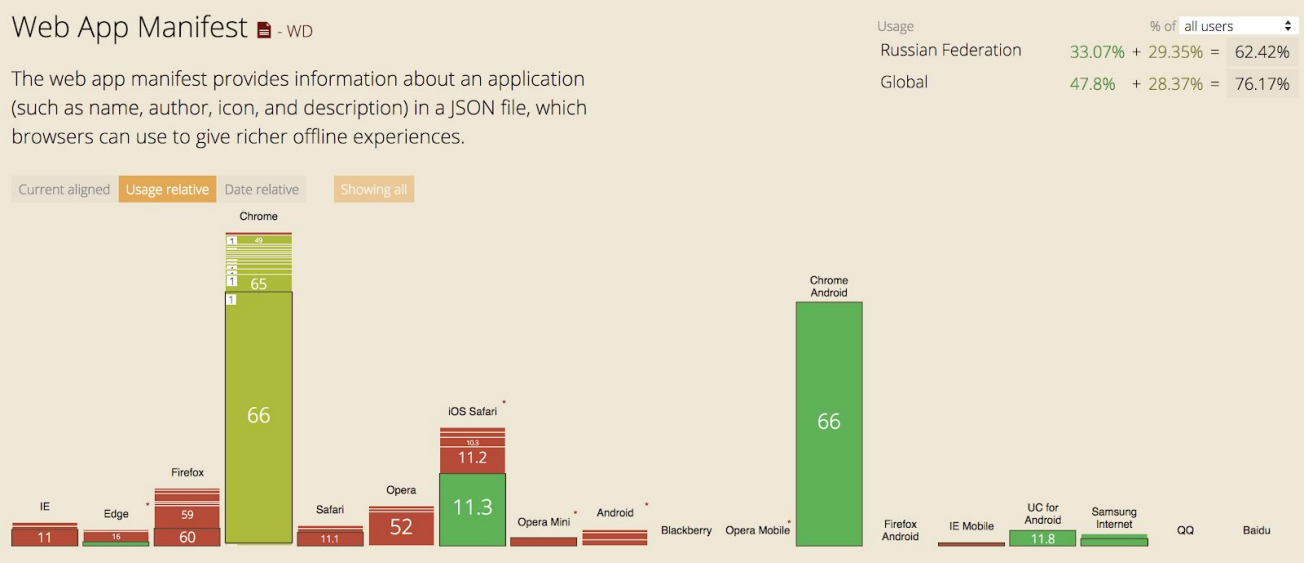
Поскольку прогрессивные веб-приложения доступны для всех, у кого есть браузер, они могут передаваться напрямую через URL. Также контент и отдельные страницы приложения должны быть связаны друг с другом, чтобы обеспечить возможность совместного использования и возможность повторного открытия приложения на том же месте.

Прогрессивные веб-приложения - это не новые технологии или рамки, это новый подход к созданию веб-приложений, которые ведут себя как

нативные. Google прилагает большие усилия в предоставлении учебных пособий и поддержки этого метода, а также с большей поддержкой браузеров и платформ это может быть будущее мобильной сети. Приложения также могут воспользоваться преимуществами технологий, упомянутых выше, даже на рабочем столе, где также очень полезны быстрые и автономные доступные веб-приложения. Особенно для кэширования использующего Service Worker.

Результаты

В работе был выбран Electron.js ввиду того, что у PWA плохая поддержка десктопных браузеров (появилась только в Chrome 67). А так же Electron.js зарекомендовал себя в многих успешных продуктах таких как: Skype, Slack, Atom, Github Desktop, а в быстро развивающейся сфере фронтенд разработки, зачастую это является одним из ключевых факторов (размер сообщества).



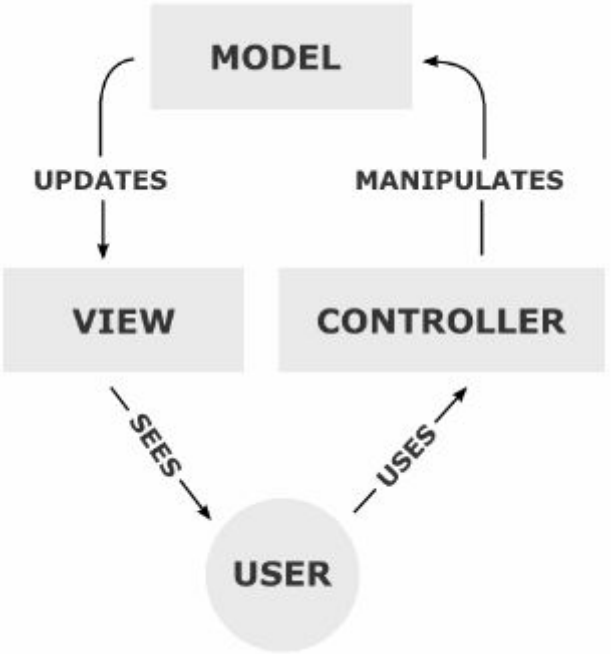
Также стоит заметить, что на момент написания работы только у 76% пользователей был доступ к функциям PWA, из них 62% - пользователи мобильных браузеров.

ГЛАВА 2 Архитектура Web-приложений

Model-View-Controller

MVC - схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

MVC делит интерактивное приложение на три различные области: обработка, вывод и ввод. Модель выполняет роль обработки, представление обеспечивает визуальную обратную связь, а контроллер обрабатывает вход или взаимодействие пользователя.



Модель

Модель представляет собой невизуальный объект, содержащий все данные и поведение, отличные от того, которое используется для его

представления пользователю. Модель представляет данные и состояние приложения и логики домена, которые ее манипулируют. Модель предоставляет интерфейс для других компонентов для доступа к данным. Чтобы максимизировать инкапсуляцию данных и повторное использование кода, модель полностью не знает о других компонентах и не зависит от них.

Модель может быть как простой целочисленной переменной, так и сложной, как корпоративный веб-сервис, обслуживающий данные JSON или XML, и построенный с использованием многослойной или даже многоуровневой архитектуры, инкапсулирующей бизнес-правила и требования к постоянству.

View

Визуальное показ приложения пользователю выполняется с помощью представления. Представления не знают, какие данные отображаются пользователю, они только представляют его. Взгляды передают действия пользователя, такие как движение мыши, штрихи клавиатуры или жесты касания к контроллеру. Представления также предлагают способы управления контроллером, например, включение и выключение функций.

Представления обычно вложены, так что в представлении содержатся субпредставления внутри него. Вид сверху этой иерархии может отвечать за общие аспекты, такие как управление окном, в котором выполняется приложение. В случае с MVC это реализовано с помощью шаблона Компоновщик, а также реализовано множество современных

инструментов и интерфейсов пользовательского интерфейса используют этот паттерн.

Controller

Контроллер - это то, что связывает представления и модели в MVC. Контроллер принимает вход пользователя как события из представления и принимает интеллектуальные решения для представления. Этот механизм событий выполняется с помощью шаблона стратегии. Шаблон стратегии делает контроллеры изменчивыми и может использоваться, например, для обеспечения редактируемого и неизменяемого поведения в одной и той же форме.

Flux

Flux - это архитектура приложений, используемая Facebook для создания клиентских веб-приложений. Она дополняет составные компоненты представления React, используя однонаправленный поток данных. Это скорее шаблон, чем формальная структура, и вы можете сразу начать использовать Flux без большого количества нового кода.

Приложения Flux имеют три основные части: диспетчер (dispatcher), хранилище (store) и представление (view). Их не следует путать с Model-View-Controller. Контроллеры существуют в приложении Flux, но они являются представлением, которые часто встречаются в верхней части иерархии, которые извлекают данные из хранилищ и передают эти данные своим детям. Кроме того, создатели действий(action) - вспомогательные методы диспетчера - используются для поддержки семантического API, который описывает все изменения, которые

возможны в приложении. Полезно подумать о них как о четвертой части цикла обновления потока.

Flux избегает MVC в пользу однонаправленного потока данных. Когда пользователь взаимодействует с представлением React, представление передает действие через центральный диспетчер в различные хранилища, в которых хранятся данные и бизнес-логика приложения, которые обновляют все просмотренные виды. Это особенно хорошо работает с декларативным стилем программирования React, который позволяет хранилищу отправлять обновления без указания способа перехода между состояниями.

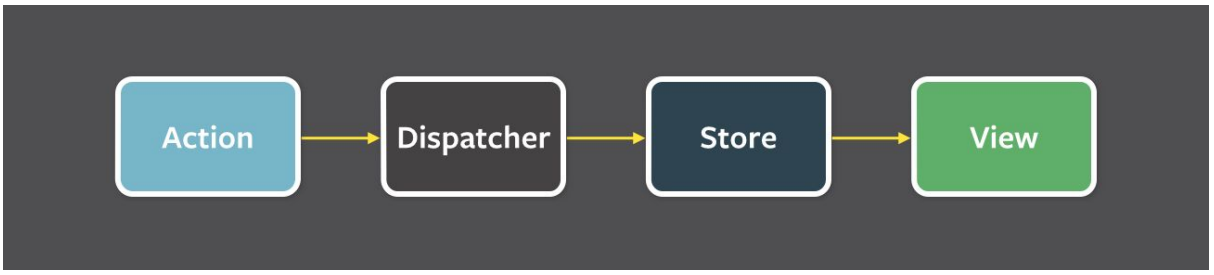
Мы изначально решили правильно обрабатывать производные данные: например, мы хотели показать счетчик непрочитанных сообщений, а в другом представлении показан список потоков, а непрочитанные - выделены. С MVC было сложно обрабатывать один поток, поскольку чтение обновляло модель потока, а затем также нужно было обновлять непрочитанную модель счета. Эти зависимости и каскадные обновления часто встречаются в большом приложении MVC, что приводит к запутанному переплетению потока данных и непредсказуемым результатам.

Управление инвертируется хранилищами: хранилища принимают обновления и согласовывают их по мере необходимости, а не в зависимости от чего-то внешнего, чтобы обновлять свои данные согласованным образом. Ничто за пределами хранилища не имеет никакого представления о том, как он управляет данными для своего домена, помогая сохранить четкое разделение проблем. В хранилищах нет прямых методов настройки, таких как setAsRead(), но вместо этого

есть только один способ получения новых данных в их автономный мир - обратный вызов, который они регистрируют в диспетчере.

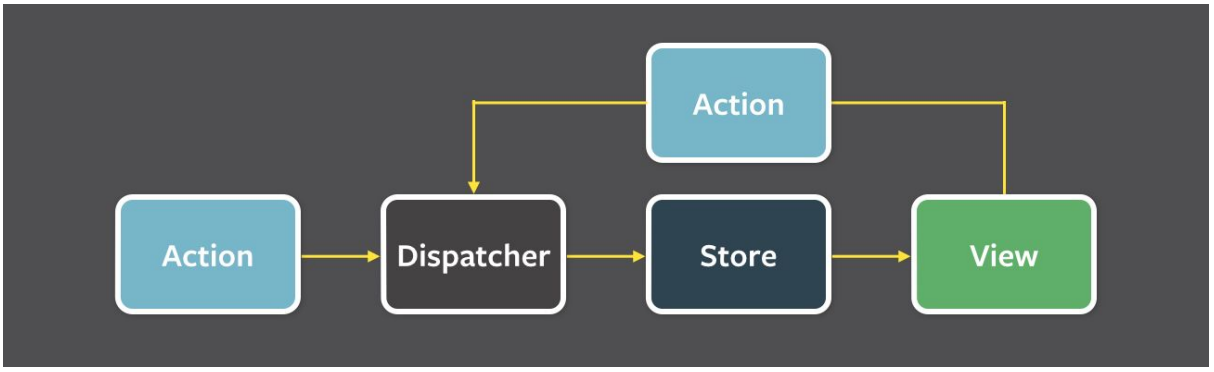
Структура и поток данных

Данные в приложении Flux поступают в одном направлении:



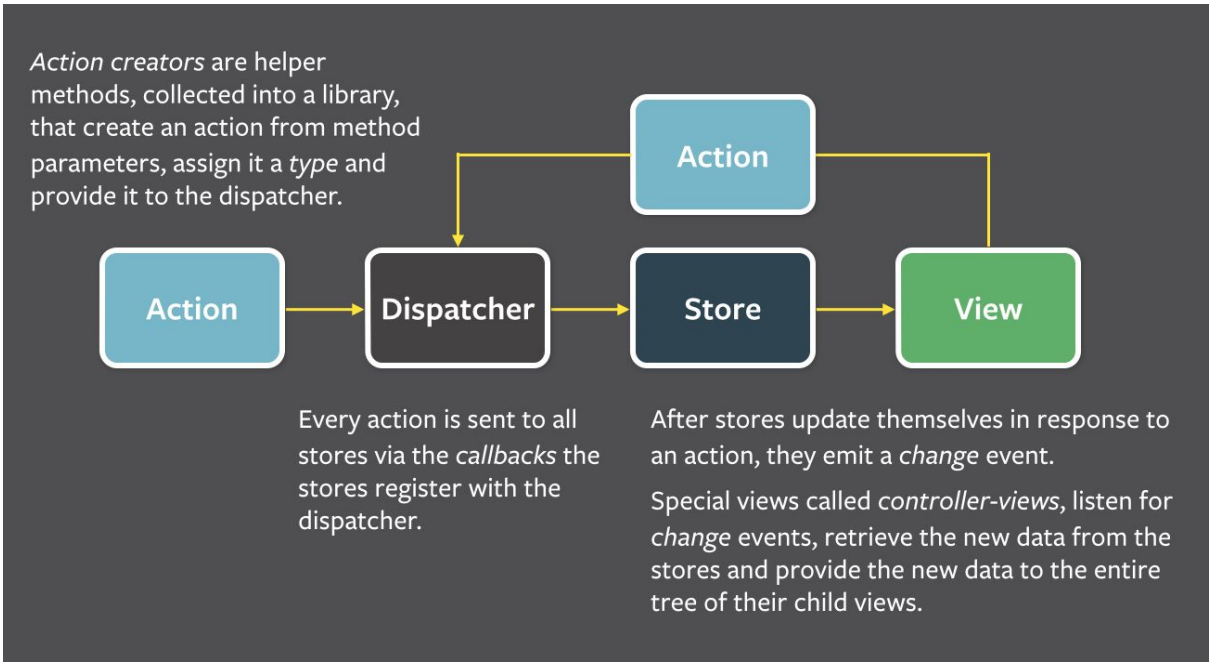
Однонаправленный поток данных является центральным для Flux, и приведенная выше диаграмма должна быть основной ментальной моделью для программиста работающего с Flux. Диспетчер, хранилища и представления являются независимыми узлами с различными входами и выходами. Действия - это простые объекты, содержащие новые данные и свойство идентификационного типа.

Представления могут приводить к распространению нового действия через систему в ответ на взаимодействие пользователя:



Все данные проходят через диспетчер в качестве центрального концентратора. Действия предоставляются диспетчеру в методе

создания действия и чаще всего происходят из пользовательских взаимодействий с представлениями. Затем диспетчер вызывает обратные вызовы, которые были зарегистрированы в хранилищах, отправляя действия во все хранилища. В пределах своих зарегистрированных обратных вызовов хранилища отвечают на то, какие действия имеют отношение к состоянию, которое они поддерживают. Затем хранилища выдают событие изменения, чтобы предупредить контроллеры-представления о том, что произошло изменение состояния данных. Контроллеры просматривают эти события и извлекают данные из хранилищ в обработчике событий. Контроллеры-представления вызывают свой собственный метод setState(), вызывая повторный рендеринг самих себя и всех их потомков в дереве компонентов.



Эта структура позволяет нам легко рассуждать о нашем приложении таким образом, который напоминает функциональное реактивное программирование или, более конкретно, программирование потоками

данных или программирование на основе потока, где данные проходят через приложение **в одном направлении**. Состояние приложения существует только в хранилищах, позволяя различным частям приложения оставаться сильно раздельными. Если между хранилищами существуют зависимости, они хранятся в строгой иерархии с синхронными обновлениями, управляемыми диспетчером.

Мы обнаружили, что двусторонние привязки данных привели к каскадным обновлениям, когда изменение одного объекта привело к изменению другого объекта, что также могло вызвать дополнительные обновления. По мере роста приложений эти каскадные обновления затрудняли предсказать, что изменится в результате взаимодействия с пользователем. Когда обновления могут изменять только данные за один раунд, система в целом становится более предсказуемой.

ГЛАВА 3 ОПИСАНИЕ ПРИЛОЖЕНИЯ

Эта глава посвящена представлению разработанного работоспособного приложения для нескольких платформ. Для доказательства разработанного подхода, было написано приложение. Было решено автоматизировать для UI/UX команды Acronis'а конвертацию видео в анимированные GIF изображение. Команда выдвинула следующие функциональные требования, разобранные в следующем разделе.

Функциональные требования для приложения

Импорт из файла, используя Drag-and-Drop

Должна быть поддержка Drag-and-drop – способ оперирования элементами интерфейса в интерфейсах пользователя при помощи манипулятора мыши или сенсорного экрана.

Способ реализуется путём «захвата» (нажатием и удержанием главной (первой, чаще левой) кнопки мыши) отображаемого на экране компьютера объекта, программно доступного для подобной операции, и перемещении его в другое место (для изменения расположения) либо «бросания» его на другой элемент (для вызова соответствующего, предусмотренного программой, действия). По отношению к окнам (также способным к перемещению подобным способом) данный термин обычно не употребляется.

Конвертация .mp4 → .gif

Приложение должно уметь конвертировать данные из видеопотока в формат анимированных изображений.

Предпросмотр

Необходимо иметь возможность предпросмотреть анимированное изображение.

Экспорт результата на жесткий диск

Должна быть поддержка сохранения картинки из памяти приложения на жесткий диск.

Кросс-платформенность

Приложение должно уметь запускаться на Windows/MacOS/Linux и большинстве браузеров, на данных платформах.

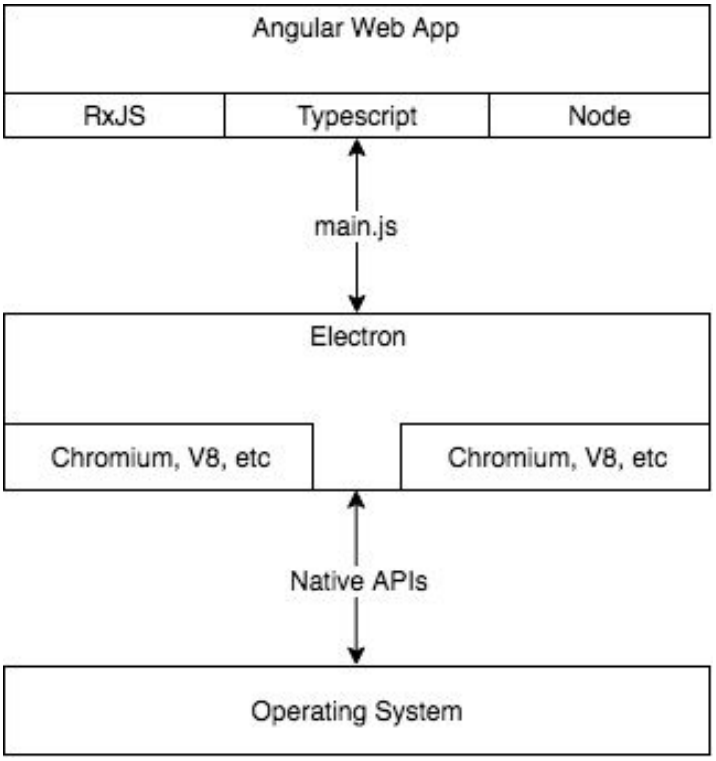
Описание сборки приложения

Приложение собирается с помощью angular cli - интерфейс командной строки, который поставляется вместе с фреймворком Angular. В который входят генерация начального приложения, в котором уже настроены конфиги для сборки с помощью Webpack. Также в него входят настроенное окружение для тестирования приложений, на основе Karma и Protractor. Весь код пишется на Typescript - типизированный язык, который компилируется в обычный Javascript.

Архитектура приложения

Приложение состоит из двух основных частей: Веб-приложение и Electron приложение, которое отвечает за генерацию файлов, которые будут нативно запускаться в требуемой среде: .exe, .dmg.

Для генерации GIF изображений используется JS библиотека **gifshot**, созданная Yahoo и умеющая создавать GIF из медиа потоков: списка изображений, видео, веб-камеры, и работать в несколько потоков.



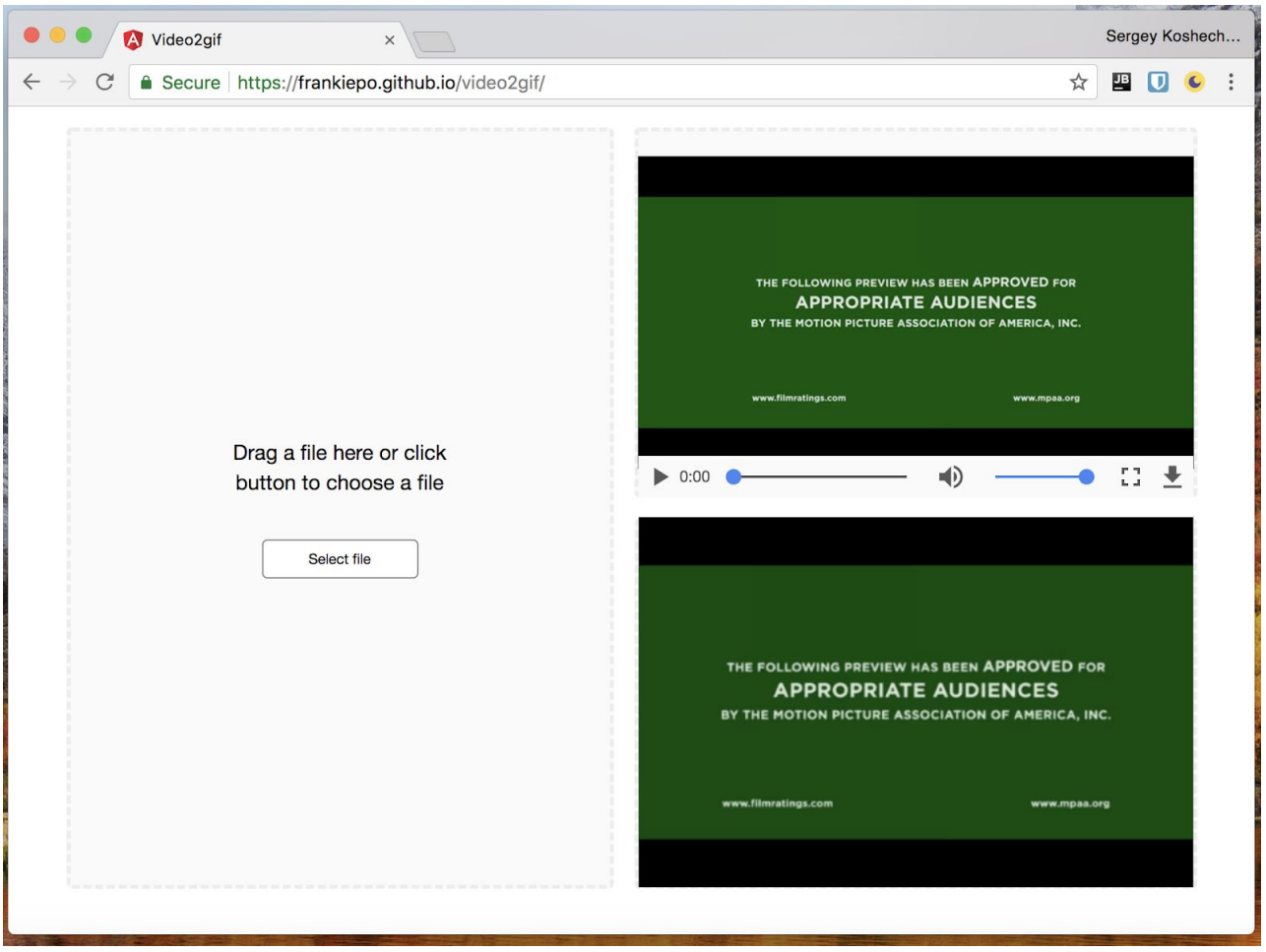
Описание запуска приложения

Для сборки нативного приложения используется скрипт, написанный на Javascript, являющийся настройкой окна движка Chromium, который находится под капотом Electron.

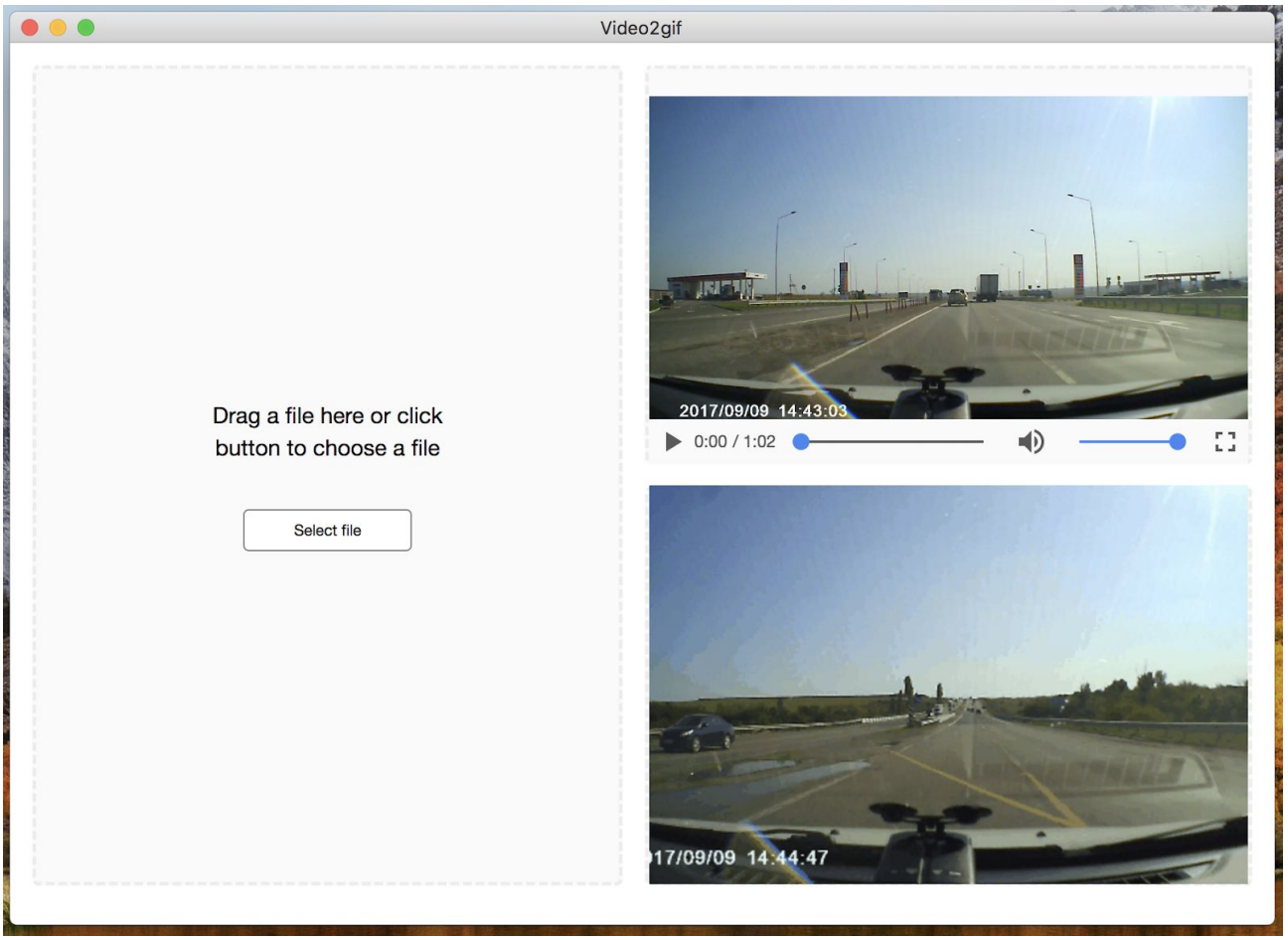
```
1   const { app, BrowserWindow } = require('electron');
2
3   let win;
4
5   function createWindow () {
6     // Create the browser window.
7     win = new BrowserWindow({
8       width: 900,
9       height: 650,
10      backgroundColor: '#ffffff',
11      icon: `file://${__dirname}/dist/assets/logo.png`
12    });
13
14
15    win.loadURL(`file://${__dirname}/dist/index.html`);
16
17    //// uncomment below to open the DevTools.
18    // win.webContents.openDevTools()
19
20    // Event when the window is closed.
21    win.on('closed', function () {
22      win = null
23    })
24  }
25
26  // Create window on electron intialization
27  app.on('ready', createWindow);
28
29  // Quit when all windows are closed.
30  app.on('window-all-closed', function () {
31
32    // On macOS specific close process
33    if (process.platform !== 'darwin') {
34      app.quit();
35    }
36  });
37
38  app.on('activate', function () {
39    // macOS specific close process
40    if (win === null) {
41      createWindow();
42    }
43  });
```

Результат

В результате разработан дизайн приложения, и сделано приложение, отвечающее всем функциональным требованиям поставленным выше. Собранное приложение было протестировано на MacOS и Windows. А его веб составляющая на всех последних версиях браузеров. В ходе проверки отличий заметно не было, и на всех платформах приложение отображалось одинаково.



Запуск приложения в браузере



Запуск приложения на MacOS.

ЗАКЛЮЧЕНИЕ

Этот магистерский проект был начат как исследование того, как проектировать и разрабатывать пользовательский интерфейс для кросс-платформенного веб-приложения. Я хотел выяснить какие основные проблемы и соображения дизайна для создания межплатформенных интерфейсов можно построить на текущих веб-технологиях и существующих методах. Дополнительная задача заключалась в том, чтобы выяснить, насколько такое приложение будет удобным. Наконец, был создан прототип кросс-платформенного веб-приложения основанный на функциональных требованиях.

Построенное приложение полностью удовлетворяло всем представленным к нему требованиям и имело все желаемые функции. Тестирование выявило минимальные проблемы с отображением на разных платформах. Кроме того, разработанный прототип был нацелен на мощные устройства и потребовал бы больших оптимизаций для работы на более простых. Однако, как доказательство концепции, прототип работал хорошо, и я смог проверить, как пользовательский интерфейс работал на разных устройствах.

Из-за отсутствия готового дизайна, и отсутствия у меня необходимой экспертизы для этого, чтобы приложение было готово к внедрению заказчику, требуется еще больше времени, однако общий результат положителен. Согласованность в пользовательском опыте и непоследовательная поддержка CSS на разных платформах является одной из основных проблем при создании межплатформенного пользовательского интерфейса.

Кросс-платформенная разработка веб-приложений является сложной областью, и во время процесса очень важно начинать с простой структуры, часто тестировать на реальных устройствах и использовать гибкие и прогрессивные методы повышения, чтобы обеспечить масштабируемость пользовательского интерфейса.

Несмотря на то, что родные приложения работают намного лучше, чем сайты, кросс-платформенное веб-приложение может стать лучшей стратегией для будущего развития. В сочетании с быстро развивающимися веб-технологиями, я считаю, что кросс-платформенные веб-приложения имеют большой потенциал в ближайшем будущем.

Растущее использование нескольких устройств ставит повышенные требования к доступу к информации и услугам в любом месте и на любом устройстве. Веб-браузеры развиваются, становясь дополнительными платформами приложений. HTML5, CSS и JavaScript широко используются для создания веб-приложений, и мы привыкли ежедневно пользоваться веб-приложениями. Все это делает Интернет уже проверенной платформой.

СПИСОК ЛИТЕРАТУРЫ

1. А.И.Дмитриенко. Информационная безопасность ERP-систем [Электронный ресурс]. — 2011. — URL: http://cyberleninka.ru/article/n/informatsionnaya-bezopasnost-erp-sistem-1.
2. Myeongju Ji Sungryong Kim Yongjin Park. Mobile device management system with portable devices // Consumer Electronics (ISCE), 2015 IEEE International Symposium on. — 2015.
3. Vendor Landscape: Mobile Device Management SuitesElectronic resource, InfoTech research group. — URL: http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/customers/info-iech-research-group-vendor-landscape-mdm-suites.pdf (online; accessed: 22.05.2018).
4. Leslie Liu Randy Moulic Dennis Shea. Cloud Service Portal for Mobile Device Management // e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on. — 2010.
5. Enterprise Mobility ManagementElectronic resource, VMworld 2014 conference coverage. — URL: http://searchmobilecomputing.techtarget.com/definition/enterprise-mobility-management-EMM (online; accessed: 6.03.2018).
6. WSO2 Enterprise Mobility ManagerElectronic resource, WSO2. — URL: https://docs.wso2.com/display/EMM220/WSO2+Enterprise+Mobility+Manager.
7. Third party App Integration (REST APIs)Electronic resource, Vtiger CRM. — URL: http://community.vtiger.com/help/vtigercrm/developers/third-party-app-integration.html.

8. Roberto Acerbis Aldo Bongio Marco Brambilla Stefano Butti. ModelDriven Development of Cross-Platform Mobile Applications with Web Ratio and IFML // 2nd ACM International Conference on Mobile Software Engineering and Systems. — 2015. — P. 170–171.
9. Епифанов А. Обзор кросс-платформенных решений для разработки мобильных приложений // CEE-SEC(R) 2016. 49
10. Jian Mao Yaoqi Xinshu Dong Yue Chen Ruilong Wang Zhenkai Liang. Unexpected Behaviours in HTML5 Mobile Apps using Difference in Execution Context.

АББРЕВИАТУРЫ

API - Application Programming Interface

CSS - Cascading Style Sheet

DOM - Document Object Model

GIF - Graphics Interchange Format

HTML - Hypertext Markup Language

IDE - Integrated Development Environment

NPM - Node.js Package Manager

JS - JavaScript

JSON - JavaScript Object Notation

MVC - Model View Controller

PWA - Progressive Web Application

RxJS - Reactive Extensions for JavaScript

SDK - Software Development Kit

SW - Service Worker

TS - TypeScript