Разработка прогрессивного web-приложения для системы управления push-уведомлениями

Ю.Ю. Гавриленко, Д.Ф. Саада, Е.А. Ильюшин, Д.Е. Намиот

***Аннотация*— В данной статье представлена система управления push-уведомлениями. В ее основу положены результаты проекта, включающего в себя разработку сервера приложения для системы управления push-уведомлениями. Push-уведомления (серверные уведомления) в последнее время используются все чаще, в том числе, и как замена для SMS (MMS) уведомлений. Такого рода уведомления могут использоваться как в мобильных приложениях на разнообразных платформах, так и в веб-приложениях, работающих в современных мобильных операционных системах. В данной работе была использована технология браузерных push-уведомлений - web push. В статье проведен подробный обзор технологий и архитектур, с помощью которых были разработаны серверная и клиентская часть web-приложения, обозначены преимущества этих технологий. Подробно рассмотрена архитектура приложения и схемы базы данных. Особое внимание уделено технологии Progressive Web Application (PWA), которая продвигает разработку web-приложения, являющегося некоторым гибридом веб-сайта и зависимого от платформы (нативного) приложения. Эта технология была анонсирована в 2015 году компанией Google и быстро завоевала популярность своими очевидными преимуществами перед нативными приложениями и обычными web-приложениями. Также подробно описаны сервисные инструменты (фреймворки, библиотеки) с открытым кодом, использовавшиеся при создании приложения.**

***Ключевые слова*— JavaScript, Web Push API, PWA, RESTful API**

I. Введение

Технология push - способ распространения информации в сети таким образом, что данные поступают от отправителя к получателю на основе установленных параметров. Эту технологию еще называют серверными уведомлениями, поскольку здесь информация формируется на сервере и доставляется клиенту по инициативе сервера. В этом и состоит ее отличие от технологии pull, где клиент инициирует запрос новой информации. Push-уведомления могут появляться на экране любого устройства, если есть возможность вывода на экран данных, принятых из сети Интернет. При этом пользователю не обязательно поддерживать приложение постоянно работающим, чтобы получать уведомления. Push-оповещения могут быть браузерными, мобильными, а также существовать в связке «компьютер-приложение».

Описываемая в работе система была спроектирована и реализована в рамках курсового проекта, выполненного в лаборатории Открытых информационных технологий факультета ВМК МГУ имени М.В. Ломоносова.

Применение технологии push является востребованным и выгодным благодаря множеству преимуществ. На данный момент push-уведомления поддерживаются всеми современными операционными системами и браузерами, а компании, разрабатывающие эти продукты, имеют свои собственные серверы push-уведомлений, которые являются посредником между сервером автора приложения и конечным пользователем. Такой подход позволяет обеспечить безопасность и уменьшить потребление энергии на устройстве пользователя, так как благодаря интеграции операционной системы и сервера push-уведомлений отпадает необходимость в периодическом опросе каждого сервера поставщика контента на предмет новых данных. Также, технология push явно выигрывает у таких технологий, как SMS и MMS за счет того, что эта технология является бесплатной. Минусами данной технологии является необходимость установки приложения, через которое будет поддерживаться функционал push-уведомлений, либо браузера в случае web push технологии. Естественно, что для гарантии актуальности информации необходима поддержка постоянного интернет соединения.

Существует множество компаний, которые предоставляют услуги по настройке и рассылке push-уведомлений из пользовательского приложения или с сайта. Также, применение этой технологии рассматривалось во многих работах, выполнявшихся в лаборатории ОИТ [1].

Веб-приложение - это клиент-серверное приложение, в котором взаимодействие клиента с сервером осуществляется при помощи браузера, а за серверную часть приложения отвечает веб-сервер. Такой подход был выбран как наиболее предпочтительный в связи с

Статья получена 20 августа 2018.
Ю.Ю. Гавриленко – МГУ имени М.В. Ломоносова (e-mail: yulyasayshello@gmail.com)
Д.Ф. Саада – МГУ имени М.В. Ломоносова (e-mail: danielbitesdog@gmail.com)
Е.А. Ильюшин – МГУ имени М.В. Ломоносова (e-mail: eugene.ilyushin@gmail.com)
Д.Е. Намиот – МГУ имени М.В. Ломоносова (e-mail: dnamiot@gmail.com)

некоторыми особенностями веб-приложений, например, независимости от платформы. Браузеры, поддерживающие технологию push-оповещений, имеют собственный push-сервис - систему для обработки сообщений и их доставки нужному получателю. Поскольку система push-уведомлений реализуется в виде web-приложения, была использована технология браузерных push-уведомлений - web push.

Схема системы push-уведомлений включает в себя следующие составляющие [2]:

1. Приложение - пользовательская часть программы, которая взаимодействует с браузером для получения подписки на push-уведомления и обновленных push-событий;
2. Сервер приложения - backend сервис, генерирующий обновления push-событий для доставки через push-сервер;
3. Push-система, ответственная за доставку событий от сервера приложения к самому приложению;
4. Push-server - сервис, обрабатывающий события и доставляющий их нужному подписчику. Каждый браузер использует свой push сервис, в случае Google Chrome это Firebase Cloud Messaging;
5. Канал - пользовательский запрос временной информации, касающейся какой-либо тематики, который создает endpoint для отправки обновлений push событий;
6. Endpoint - уникальный URL, который используется для отправки push-уведомлений определенному подписчику;
7. Подписка - приложение, подписываемое на систему push-уведомлений, чтобы получать обновления, либо пользователь, который подписывает приложение на систему push-уведомлений, нажав на кнопку "Подписаться на уведомления";
8. Обновление подписки - событие, отправленное системе push-уведомлений, в результате которого push-сервис получает push-уведомление;
9. Push-уведомление - сообщение, отправленное от сервера приложения на приложение через push-сервис.

Это проиллюстрировано на рис. 1.

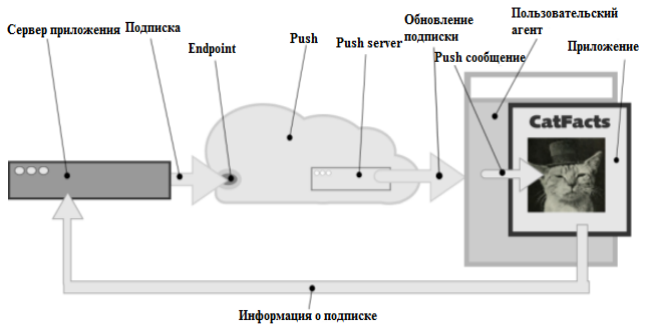

Рис. 1. Схема системы push уведомлений.

II. Обзор архитектуры приложения

Архитектура приложения во многом продиктована тем, как устроены браузерные push-оповещения на данный момент. У пользователя в браузере помимо основного веб-приложения регистрируется отдельная программа— *service worker*. Это специальный JavaScript файл, который может запускаться в фоновом режиме отдельно от самой веб-страницы. Он продолжает работать, даже когда сам браузер закрыт. За счет этого появляется возможность получать уведомления в том случае, когда веб-приложение не запущено у пользователя. После регистрации в системе, *service worker* подписывает браузер пользователя на получение уведомлений, после чего специальный объект подписки отправляется на сервер. Объект подписки содержит в себе всю необходимую информацию для отправки уведомления конкретному пользователю и состоит из уникального URL-адреса, который используется для отправки сообщения конкретному пользователю (endpoint), а также публичного ключа шифрования [3]. Публичный ключ шифрования — это один из двух серверных ключей шифрования, предоставленный веб-приложению.

Объекты подписки, которые сервер получает от клиента, должны сохраняться в базе данных для дальнейшего использования. Когда поступает запрос на отправку уведомления пользователю, сервер обращается к базе данных, получает соответствующий объект подписки и делает запрос к push-сервису, используя так называемый endpoint-адрес. В целях безопасности push-сервис браузера должен проверить, действительно ли сервер может отправлять уведомления пользователю, либо же объект подписки просто был украден. Для этого пользовательский сервер должен использовать протокол Voluntary Application Server Identification for Web Push (VAPID) и перед отправкой запроса подписывать своим приватным ключом отправляемые push-сервису данные. Последний, используя публичный ключ, который он получил и сохранил на шаге регистрации *service worker*'а, проверяет подлинность сервера и в случае положительного результата отправляет сообщение пользователю [4]. Как только сообщение было получено, активируется *service worker* и обработает его соответствующим образом (в нашем случае — отобразит уведомление).

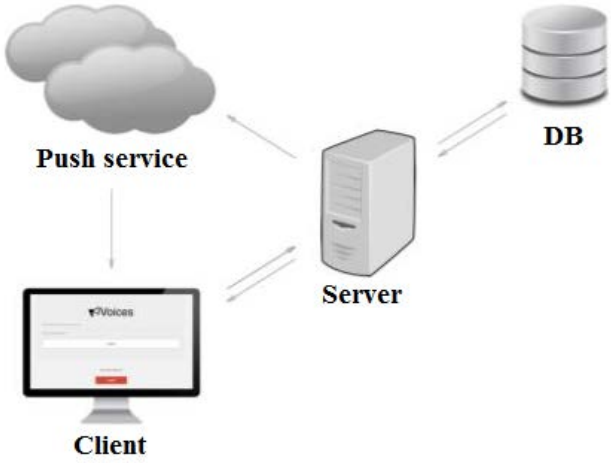

Рис.2. Схематичное изображение архитектуры приложения.

Таким образом, архитектура всего веб-приложения схематично выглядит так, как показано на рис. 2 и соответствует модели Клиент-Сервер.

III. Средства реализации сервера

Для реализации серверной части системы управления push-уведомлениями выбран язык JavaScript. Разработка серверных приложений на языке JavaScript стала возможной благодаря *Node.js* (далее просто *Node*) — асинхронной событийно-ориентированной программной платформе для языка JavaScript, которая основана на движке V8, транслирующем JavaScript в машинный код [5]. В *Node* используется механизм *event loop*, который определяет, какое действие должно произойти при наступлении некоего события. Любое асинхронное действие должно иметь ассоциированную с ним функцию обратного вызова (*callback*). После того, как асинхронное действие начало выполняться, поток исполнения не дожидается его завершения и продолжает идти дальше. Когда асинхронное действие завершит работу, вызовется *callback*, которому в качестве аргумента будет передан результат действия [6]. Такой подход в значительной степени улучшает быстродействие сервера за счет неблокирующего ввода/вывода, а также упрощает разработку сервера.

В качестве архитектурного стиля построения API был выбран подход REST (Representational State Transfer) — передача состояния представления [7]. Основная идея заключается в том, чтобы информации из запроса было достаточно, чтобы корректно на него ответить, не храня на сервере никаких данных о состоянии представления. При этом адрес запроса обычно указывает на ресурс, с которым проводится действие, а метод HTTP запроса указывает, что именно с ним необходимо сделать.

Существует ряд требований, выполнение которых обязательно для всех RESTful API:

1) Модель клиент-сервер;
2) Отсутствие состояния - в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится;
3) Кэширование - клиенты, а также промежуточные узлы, могут выполнять кэширование ответов сервера;
4) Единообразие интерфейса - наличие унифицированного интерфейса является фундаментальным требованием дизайна RESTful API;
5) Слои - клиент обычно не способен точно определить, взаимодействует ли он напрямую с сервером или же с промежуточным узлом;
6) Код по требованию - REST может позволить расширить функциональность клиента за счёт загрузки кода с сервера в виде апплетов или сценариев.

Для построения API на основе *Node.js* используется фреймворк *Express.js* — гибкий веб-фреймворк, предоставляющий обширный набор функций для мобильных и веб-приложений. Ключевой особенностью данного фреймворка являются промежуточные обработчики (middleware). Промежуточный обработчик — это функция, которая имеет доступ к объекту запроса (request), объекту ответа (response) и к следующей функции промежуточной обработки (next). Промежуточный обработчик может выполнять любой код, изменять объекты запроса и ответа, завершать цикл “запрос-ответ”, а также вызывать следующую функцию промежуточной обработки.

Для отправки push-оповещений удобно использовать модуль web-push. Данный модуль берет на себя работу с Web Push протоколом, который описывает взаимодействие пользовательского сервера и push-сервиса [8]. Помимо этого, модуль web-push самостоятельно шифрует данные в соответствии со спецификацией Message Encryption for Web Push spec, предоставляет удобный API для отправки сообщений, а также включает в себя консольную утилиту, позволяющую генерировать уже рассмотренные ранее ключи шифрования [9].

Для реализации данного функционала регистрации, аутентификации и авторизации самым популярным решением в среде *Node.js* является использование middleware для фреймворка Express под названием *Passport.js*. Данный модуль является чрезвычайно гибким в настройке и может использовать различные стратегии аутентификации, которые свободно могут быть разработаны и добавлены сторонними разработчиками [10]. Учитывая выбор REST в качестве архитектурного стиля построения API, вполне логичным кажется реализация аутентификации при помощи JSON Web Token’ов.

JSON Web Token (JWT) — это открытый стандарт для создания токенов доступа, основанный на JSON формате. Как правило, он используется для передачи данных авторизации в клиент-серверных приложениях. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности. Токен JWT состоит из трех частей: заголовок (header), полезная нагрузка (payload) и подпись или данные шифрования. Первые два элемента — это JSON объекты определенной структуры. Третий элемент вычисляется на основании первых и зависит от выбранного алгоритма (в случае использования не подписанного JWT может быть опущен).

Рассмотрим структурно процесс аутентификации и авторизации пользователя с использованием технологии JWT:

1) Пользователь при помощи заданного набора данных (например, пара логин-пароль) проходит аутентификацию на сервере;
2) В случае успешной аутентификации, сервер отправляет пользователю JSON Web Token;
3) В последующих запросах к серверу клиент прикрепляет выданный ему токен;
4) Сервер осуществляет валидацию токена из запроса

и, в зависимости от результата, осуществляет авторизацию пользователя.

Из соображений безопасности, в базе данных нельзя хранить пароли пользователей. В связи с этим, используется модуль для *Node.js* под названием *bcrypt*. Он позволяет легко создавать хэши паролей, которые далее сохраняются в базе данных.

Для генерации уникальных идентификаторов используется модуль *uuid* и четвертая версия генерации идентификаторов. Universally unique identifier (UUID) — это стандарт идентификации, используемый в создании программного обеспечения, стандартизированный Open Software Foundation.

IV. Структура базы данных

Для формирования структуры базы данных используется *Node.js* модуль *mongoose*. *Mongoose* — это Object Document Mapper (ODM) для документо-ориентированной СУБД MongoDB. С его помощью задается схема документов в коллекции, осуществляется валидация данных, а также создаются специальные функции для некоторых действий с базой данных [11].

Для поддержки авторизации необходимо создать схему пользователя. Каждый пользователь обязательно должен иметь имя и пароль. Также необходимо создать схему для хранения каналов рассылки. Каждый канал должен иметь название, уникальный идентификатор для создания ссылки на этот канал и имя пользователя, который владеет данным каналом. В схему канала рассылки также можно включить список сообщений, отправленных в нем, так как каждое сообщение относится только к одному каналу.

Визуально можно изобразить общую структуру базы данных так, как показано на рис. 3.

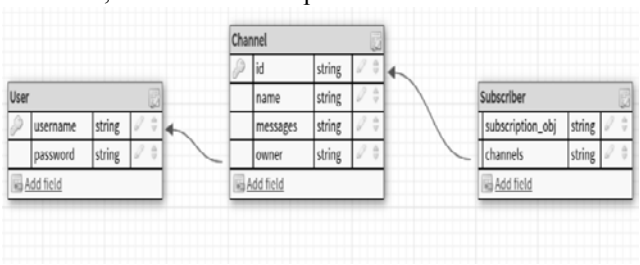

Рис. 3 Структура базы данных.

Схема выше соответствует реляционной модели, однако мы используем документо-ориентированную СУБД, поэтому можем пользоваться ее преимуществами и не заводить отдельные коллекции для истории сообщений или списка каналов рассылки, на которые подписан пользователь; мы можем просто хранить в объекте массив данных.

V. Реализация сервера

Была определена следующая структура сервера:

```
/
| — /api
|   — /controllers
|   |   authController.js
|   |   pushController.js
|   — /models
|   |   userModel.js
|   |   channelModel.js
|   |   subscriberModule.js
|   — /routes
|   |   apiRoutes.js
| — /config
|   database.js
|   passport.js
|   vapid.js
| server.js
```

Файл *database.js* содержит в себе адрес для подключения к базе данных. Файл *passport.js* задает метод извлечения информации из заголовка запроса, а также функцию *verify (payload, done)*, которая далее будет использоваться модулем для верификации. В *vapid.js* хранятся публичный и приватный серверный ключи для использования протокола Web Push. Оба ключа сгенерированы специальной утилитой заранее, однако при помощи метода *webpush.generateVAPIDKeys()* их можно менять.

Файл *server.js* является точкой входа программы. В данном файле подключается ряд необходимых модулей и файлов. Далее осуществляется подключение к базе данных. Адрес БД импортируется из конфигурационного файла. Следующим шагом необходимо задействовать ряд функций промежуточной обработки. Учитывая архитектуру приложения в целом, нам нужно включить совместное использование ресурсов между разными источниками (Cross-origin resource sharing, CORS), так что первая middleware-функция добавляет необходимые заголовки в объект ответа сервера. После этого используется модуль *body-parser*, инициализируется модуль passport, для которого сразу же выбирается стратегия passport-jwt, которая описывает аутентификацию при помощи JSON Web Token’ов. Последними шагами конфигурации являются вызов метода *webpush.setVapidDetails()* для передачи VAPID-ключей модулю web-push и задание файла маршрутизации для путей, начинающихся на “/api”. На этом конфигурация завершена и сервер начинает прослушивать выбранный порт на предмет подключений.

Рассмотрим далее каталог *api*. Его подкаталог *models* содержит три файла, каждый из которых задает схему для хранения соответствующих документов (пользователя, подписчика и канала) в базе данных. Подкаталог *controllers* содержит в себе два скрипта. Модуль *authController* предоставляет набор функций для осуществления регистрации и аутентификации пользователей. Модуль *pushController* обрабатывает все остальные запросы: получение списка каналов, создание канала, осуществление подписки на канал, отправка сообщения и так далее.

В папке *routes* содержится файл *apiRoutes.js*, который отображает программный интерфейс приложения. На рис. 4 приведен код этого файла.

```
var passport = require('passport');
var express = require('express');
var router = express.Router();
var authServer = require('../controllers/authController');
var pushServer = require('../controllers/pushController');

router.post('/register', authServer.register);

router.post('/', authServer.login);

router.get('/channels', passport.authenticate('jwt',
                                        { session: false }),
                                       pushServer.listChannels);

router.post('/channels', passport.authenticate('jwt',
                                         { session: false }),
                                         pushServer.createChannel);

router.get('/channel/:id', pushServer.getChannelHistory);

router.put('/channel/:id', pushServer.subscribe);

router.post('/channel/:id', passport.authenticate('jwt',
                                          { session: false }),
                                          pushServer.push);

router.delete('/channel/:id', passport.authenticate('jwt',
                                           { session: false }),
                                           pushServer.deleteChannel);


module.exports = router;
```

Рис. 4 - код файла *apiRoutes.js*

В строках 1-5 подключаются необходимые модули. Строки 7-19 задают для каждого пути необходимую функцию и наглядно демонстрируют результат работы. Зайдя на сайт, пользователь может либо отправить в POST-запросе данные для авторизации, либо перейти на страницу регистрации и отправить оттуда POST-запрос с данными для регистрации. Пройдя этап аутентификации, пользователь может получить список своих каналов при помощи GET-запроса, либо же создать новый канал при помощи POST-запроса. Открыв страницу некоего конкретного канала рассылки, пользователь может подписаться на него при помощи PUT-запроса. Методы GET и POST позволяют посмотреть историю сообщений и отправить уведомление подписчиками канала соответственно. Все действия, кроме регистрации, входа в систему и подписки на канал, требуют аутентификации пользователя.

VI. Технология Progressive Web Application

Progressive Web Application (PWA) - web-приложение, которое является "гибридом" web-сайта и нативного (зависимого от платформы) приложения. Технология PWA была анонсирована в 2015 году компанией Google и со временем завоевала должное внимание своими очевидными преимуществами перед нативными приложениями и обычными web-приложениями. Сейчас PWA называют основным трендом web-разработки 2018 года.

Прогрессивное web-приложение представляет собой web-страницу, которая создана по определенному стандарту. Благодаря поддержке браузера, PWA можно установить на домашний экран смартфона. В дальнейшем, это приложение будет функционировать, как нативное: благодаря загрузке информации в кэш и поддержке service worker'ов, использование PWA-приложения выглядит также, как и использование нативного приложения.

Выбор подхода PWA позволяет сделать приложение наиболее удобным для использования, лаконичным, быстродействующим и наиболее соответствующим современным идеям в написании приложений. Эта технология является наиболее прогрессивной в настоящее время и обладает большим количеством преимуществ, которые, в том числе, позволяют как можно более удобно поддерживать технологию push уведомлений в приложении. В построении приложения преследовалась цель наложения как можно меньшего количества условий на пользователя, поэтому тот факт, что веб-приложение не нужно специально загружать и устанавливать на устройство, сыграл в пользу выбора такой концепции.

Прогрессивность web-приложения означает то, что оно сможет работать на любых платформах при любых условиях. Приложения, созданные согласно концепции PWA, обладают следующими качествами:

1) Независимость от платформы использования и адаптивность: PWA можно использовать на любом гаджете, независимо от размера экрана и других спецификаций устройства, достаточно наличия последней версии популярного браузера;
2) Если приложение соответствует критериям PWA, браузер предлагает установить его на домашний экран. Это происходит при повторном посещении страницы, то есть, в том случае, когда пользователь проявил интерес к приложению;
3) Мгновенная установка: все компоненты, которые требуют загрузки, уже загружены в кеш при первом посещении страницы;
4) Обновления происходят моментально и не требуют длительной загрузки;
5) Независимость от магазина приложений: это делает установку гораздо более простым процессом для пользователя, а разработку лишает необходимости следовать правилам магазина приложений, в котором планируется распространять продукт (в частности, лишает необходимости платить 30% от объема продаж магазину приложений);
6) Независимость от платформы позволяет разработчику не тратить ресурсы на поддержку работы приложения на разных платформах: в случае PWA достаточно разработать одну версию, которая отображается одинаково на всех устройствах;
7) PWA работает независимо от наличия подключения к сети;
8) Приложение не занимает места в памяти телефона, поскольку эффективно использует возможности браузера;
9) PWA идентифицируется как "приложение" в поисковых системах.

Для того чтобы создать приложение прогрессивным, необходимо соблюдать определенные требования при разработке. Последние версии браузеров автоматически проверяют web приложение на соответствие требованиям PWA и принимают дальнейшие действия для соответствующего отображения данных

приложений. Google предоставила список требований, соответствуя которым разрабатываемое приложение становится прогрессивным. Среди таких требований представлены следующие [12]:

1) Приложение обслуживается через HTTPS;
2) Страницы адаптивны на планшетах и мобильных устройствах;
3) Все URL приложения загружаются в состоянии offline;
4) Предоставлены метаданные для добавления приложения на домашнюю страницу;
5) Первая загрузка производится быстро даже через 3G;
6) Приложение является кросс-браузерным;
7) Переходы на страницы должны происходить моментально;
8) У каждой страницы есть URL.

Для удобства, можно воспользоваться утилитой *Lighthouse*, запускаемой из *Chrome DevTools* или из командной строки [13]. Приняв URL приложения, она автоматически проверяет его на соответствие основным требованиям, после чего выводит отчет и оценку, на сколько процентов предоставленное приложение соответствует концепции PWA.

VII. Обзор библиотеки ReactJS

В качестве библиотеки для написания интерактивного интерфейса приложения было решено использовать *React JS*. Это постоянно развивающаяся библиотека с активным сообществом, разработанная Facebook. Основная идея библиотеки *React JS* состоит в работе с виртуальным DOM.

Объектная Модель Документа (DOM, Document Object Model) – программный интерфейс, который предоставляет структурированное представление документа в виде объектов и определяет то, как эта структура должна быть доступна из программ, которые могут изменять содержимое, стиль и структуру документа [14]. Согласно идее DOM, документ представляется в виде дерева, каждый узел которого образуется HTML-тегом и вложенными в него тегами. Изменение какого-либо элемента в DOM влечет за собой обновление дерева целиком, что является излишним процессом, если были изменены не все элементы, а в случае, когда элементов много, занимает достаточно много времени. Вместо того, чтобы изменять напрямую DOM напрямую, предлагается вносить изменения в его легковесную копию - Virtual DOM, и уже после этого применять изменения к реальному DOM: происходит сравнение дерева DOM с его виртуальной копией, определяется разница и перерисовка тех элементов, которые подверглись изменению. Этот подход значительно ускоряет процесс, поскольку в процесс обработки не включены все тяжеловесные части реального DOM.

React создает легковесное дерево из JavaScript объектов (элементов), которые описывают DOM-элементы, после этого генерирует HTML, который добавляется к необходимому DOM-элементу, что вызывает перерисовку страницы в браузере [15].

Основой React-приложения являются компоненты - небольшие функции, которые описывают определенные фрагменты пользовательского интерфейса. Компоненты можно использовать повторно, то есть на основе маленьких компонентов создавать более объемные [16]. Как и традиционные HTML элементы, на вход компоненты принимают список атрибутов (props), и выводят элементы. Элементы в *React* могут быть созданы при помощи JSX - JavaScript extension. По структуре код на JSX может напоминать код на HTML, однако, он таковым не является, поскольку обладает расширенными возможностями. На самом деле, JSX является синтаксическим сахаром для функции *React.createElement*. В качестве потомков в JSX можно использовать любой JavaScript код, предварительно обернув его в фигурные скобки - {} [17].

Жизненный цикл компонента в *React* состоит из следующих этапов [18, 19]:

1) Создание компонента: определение в нем шаблона, следуя которому будут создаваться новые элементы;
2) Применение функции рендеринга (*reactDOM.render, render*), передавая ей созданный компонент;
3) Инициализация экземпляра компонента и передача ему параметров, доступ к которым обеспечивается через *this.props*;
4) Вызов метода конструктора класса constructor();
5) Вызов функции жизненного цикла *componentWillMount()*, в которой определяется поведение до того, как компонент будет встроен в реальный DOM;
6) Обработка функции рендеринга и получение в результате узла виртуального DOM;
7) Встраивание (mounting) полученного узла в реальный DOM;
8) Вызов метода *componentDidMount()*, в котором определяются действия, происходящие уже после того, как узел выведен на экран;
9) Обновление вмонтированного в DOM компонента, если это необходимо, с помощью методов *componentWillReceiveProps(), shouldComponentUpdate(), componentWillUpdate(), render(), componentDidUpdate()* по аналогии с ранее представленными методами, связанными со встраиванием компонента в DOM;
10) Демонтирование: вызов метода *componentWillUnmount()* прямо перед его исключением из DOM браузера.

Для создания приложения использовалась утилита *Create React App*, которая была создана разработчиками из Facebook. Утилита автоматически настраивает все настройки и конфигурации приложения, устанавливает все необходимые модули, настраивает Webpack и создает шаблон структуры проекта.

VIII. Реализация интерфейса приложения

Разрабатываемое приложение представляет собой сервис для создания каналов рассылки push уведомлений и подписки на них. Администратор может создавать канал, назначать ему имя и описание, отправлять краткие уведомления подписчикам этого канала. Пользователь может перейти на канал по автоматически сгенерированной при его создании ссылке и подписаться на него, чтобы в дальнейшем получать push уведомления от создателя канала.

При разработке приложения преследовалась цель сделать функционал как можно более простым и удобным для пользователей. Используя уникальный идентификатор устройства, который генерируется при подписке пользователя на Push Service при разрешении отправки уведомлений, можно избавиться от необходимости обязательной регистрации пользователя, который планирует получать уведомления, поскольку получаемые сообщения прикрепляются не к конкретному пользователю, а к конкретному устройству. Для того чтобы создавать каналы, добавлять информацию о них и отправлять уведомления, необходимо зарегистрироваться. Также, для того, чтобы ограничить число нежелательных подписчиков на личные каналы, была введена процедура доступа к каналу по ссылке.

Согласно идее PWA, при посещении web-страницы браузер предлагает добавить его на рабочий экран мобильного устройства для более удобного доступа. После добавления приложения на рабочий стол, для доступа к нему достаточно нажать на соответствующую иконку, переходить по ссылке для доступа не обязательно. Внешний вид главного экрана приложения представлен на рис. 5.

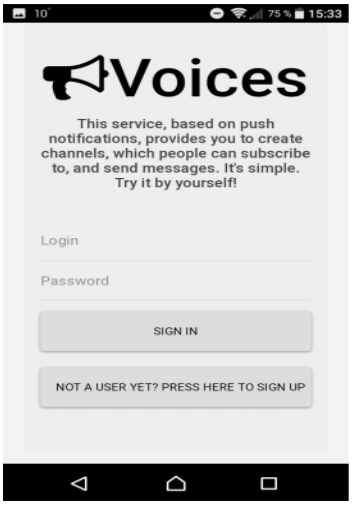

Рис. 5. Главный экран приложения

В случае, если пользователь не был ранее зарегистрирован на сервисе, ему предлагается пройти процедуру регистрации, после чего он получит возможность создавать каналы, давать им названия и описания, а также отправлять уведомления в рамках этих каналов. На рис. 6 представлен внешний вид приложения после того, как пользователь вошел в свой аккаунт, использовав логин и пароль.

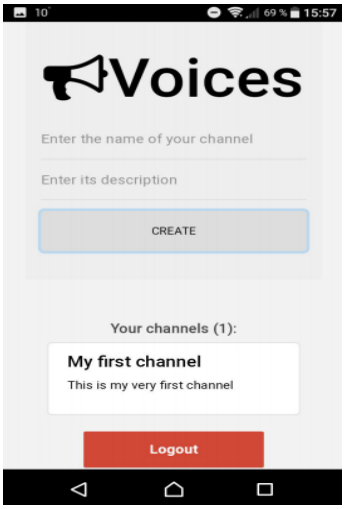

Рис. 6. Главный экран приложения после успешного входа в аккаунт

На главном экране отображается поле, в которое можно ввести имя и описание канала и создать его. Новому каналу будет соответствовать автоматически сгенерированная ссылка, перейдя по которой пользователи смогут ознакомиться со списком последних 10 уведомлений, отправленных с этого канала, а также подписаться на канал для получения уведомлений (рис. 7). Чуть ниже на главном экране расположен список ранее созданных администратором каналов. Кликнув на нужный из каналов, открывается страница, с которой администратор канала может отправлять уведомления (рис. 8).

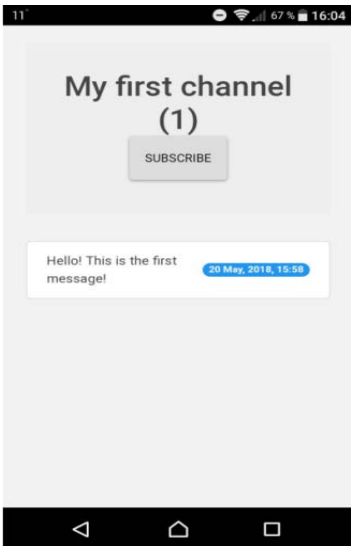

Рис. 7. Внешний вид страницы канала со списком последних отправленных сообщений и кнопкой «Subscribe»

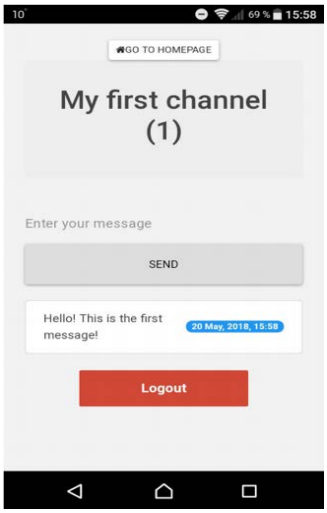

Рис. 8. Внешний вид страницы канала от лица администратора

В качестве основной библиотеки для создания интерфейса была использована библиотека для разработки пользовательских интерфейсов *React* для языка JavaScript.

IX. Заключение

В рамках данной работы было разработано web-приложение для передачи push-уведомлений. В качестве основных функций приложения можно рассматривать такие функции, как:

1) Осуществление регистрации в системе;
2) Создание тематических каналов, в рамках которых можно отправлять сообщения;
3) Подписка на интересующий канал без прохождения процедуры регистрации;
4) Получение push-уведомления о новых сообщениях в канале;
5) Просмотр истории отправленных сообщений.

В рамках дальнейшей разработки планируется добавить возможность просмотра статистики канала, ориентируясь на число переходов по уведомлениям; также планируется добавить больше информации о создателях (имя, организация, список созданных каналов) и каналах (имя создателя, дата создания, количество подписчиков, активность канала). В планах ввести возможность создания приватных каналов, доступ к которым может быть обеспечен по кодовому слову.

Сервер также может быть улучшен. В первую очередь, API нуждается в расширении функционала. В современных реалиях очень полезно иметь возможность не просто отправлять уведомления, но и получать статистику о результатах рассылки, хранить её на сервере и в дальнейшем анализировать. Вторым необходимым улучшением является создание системы создания и разбора очереди. При небольшом количестве активных соединений и не очень больших масштабах рассылки сервер будет справляться со своими задачами, однако если попытаться отправить уведомление по каналу рассылки, на который подписано, например, миллион пользователей, то сервер на длительное время перестанет отвечать на последующие запросы, что сделает работу приложения невозможной. Для этого необходима система очереди, благодаря которой сервер продолжал бы адекватно функционировать, а в свободное время разбирал бы очередь, отправляя уведомления.

Библиография

[1] Павлов В., Намиот Д. Анализ и разработка системы push-уведомлений с использованием технологий Google //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 3.
[2] Sending VAPID identified WebPush Notifications via Mozilla's Push Service [Электронный ресурс]. – URL: https://blog.mozilla.org/services/2016/08/23/sending-vapid-identified-webpush-notifications-via-mozillas-push-service
[3] Service Workers: an introduction [Электронный ресурс]. – Режим доступа: https://developers.google.com/web/fundamentals/primers/service-workers/
[4] The Web Push Protocol [Электронный ресурс]. – Режим доступа: https://developers.google.com/web/fundamentals/push-notifications/web-push-protocol
[5] About | Node.js [Электронный ресурс]. – Режим доступа: https://nodejs.org/en/about/
[6] What is Node.js? [Электронный ресурс]. – Режим доступа: http://book.mixu.net/node/ch2.html
[7] Roy, Fielding. Architectural Styles and the Design of Network-based Software Architectures / Fielding. Roy. : 2000
[8] Push API [Электронный ресурс]. – Режим доступа: https://www.w3.org/TR/push-api/#dfn-web-push-protocol
[9] web-push - npm [Электронный ресурс]. – Режим доступа: https://www.npmjs.com/package/web-push
[10] Passport.js documentation [Электронный ресурс]. – Режим доступа: http://www.passportjs.org/docs/
[11] UUID — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/UUID
[12] Progressive Web App Checklist | Web | Google Developers [Электронный ресурс]. – Режим доступа: https://developers.google.com/web/progressive-web-apps/checklist
[13] Lighthouse | Tools for Web Developers | Google Developers [Электронный ресурс]. – Режим доступа: https://developers.google.com/web/tools/lighthouse/
[14] Введение - DOM | MDN [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/DOM/DOM_Reference/Введение
[15] Что такое Virtual DOM? / Хабр [Электронный ресурс]. – Режим доступа: https:// habr.com/post/256965/
[16] Краткое руководство по React JS / Хабр [Электронный ресурс]. – Режим доступа: https://habr.com/post/248799/
[17] Introducing JSX - React [Электронный ресурс]. – Режим доступа: https://reactjs.org/ docs/introducing-jsx.html
[18] State and Lifecycle - React [Электронный ресурс]. – Режим доступа: https://reactjs.org/ docs/state-and-lifecycle.html
[19] Component Lifecycle - JS: React - Hexlet.io [Электронный ресурс]. – Режим доступа: https://ru.hexlet.io/courses/js-react/lessons/component-lifecycle/theory_uni

On the development of a Progressive Web Application for the push notifications management system

Yuliya Gavrilenko, Daniel Saada, Evgeniy Ilyushin, Dmitry Namiot

***Abstract*— This article presents a system for push notification management. It is based on the results of the project, which includes the development of an application server for the push notification systems. Push notifications (or server notifications) have recently been used increasingly as a replacement for SMS (MMS) notifications. Such notifications can be used both in mobile applications on a variety of platforms and in web applications that work in modern mobile operating systems. In this paper, we used the technology of browser push-notifications – so-called web push. The article provides a detailed overview of the technologies and architectures used to develop the server and client part of the web application and outlines also the advantages of these technologies. The architecture of the application and the database schema are discussed in detail. Particular attention is paid to the Progressive Web Application (PWA) technology, which promotes the development of a web application that is a hybrid of a website and a platform-dependent (native) application. This technology was announced in 2015 by Google and quickly gained popularity with its obvious advantages over native applications and conventional web applications. Also, the service tools (frameworks, libraries) with the open code, used at application creation are in detail described.**

***Keywords*— JavaScript, Web Push API, PWA, RESTful API**