Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Н. П. ОГАРЁВА»
(ФГБОУ ВО «МГУ им. Н. П. Огарёва»)

Факультет математики и информационных технологий

Кафедра фундаментальной информатики

УТВЕРЖДАЮ
Зав. кафедрой
канд. физ.-мат. наук, доц.
_____А. Г. Смольянов
«____» _____ 2022 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

по теме:

ПРИМЕНЕНИЕ СИСТЕМ СБОРКИ ПРИ РАЗРАБОТКЕ ПРИЛОЖЕНИЙ JAVASCRIPT

Автор бакалаврской работы _____ С. А. Антонова
подпись, дата

Обозначение бакалаврской работы БР–02069964–02.03.02–01–22

Направление подготовки 02.03.02 Фундаментальная информатика и информационные технологии

Руководитель работы
канд. физ.-мат. наук, доц. _____ А. В. Попов
подпись, дата

Нормоконтролёр
канд. тех. наук, доц. _____ С. В. Гарина
подпись, дата

Саранск 2022

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Н. П. ОГАРЁВА»
(ФГБОУ ВО «МГУ им. Н. П. Огарёва»)

Факультет математики и информационных технологий

Кафедра фундаментальной информатики

УТВЕРЖДАЮ
Зав. кафедрой
канд. физ.-мат. наук, доц.
_____А. Г. Смольянов
«____» _____ 2022 г.

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРА)

Студент Антонова Светлана Андреевна, 402 группа

1 Тема Применение систем сборки при разработке приложений JavaScript

Утверждена приказом №10134-с от 29.12.2021

2 Срок представления работы к защите _____

3 Исходные данные для научного исследования: монографии, учебные пособия, научные статьи.

4 Содержание выпускной квалификационной работы

4.1 Поддержка языка JavaScript в браузере и на сервере

4.2 Разработка модульных сценариев JavaScript

4.3 Пример разработки и сборки модульного веб-приложения

5 Приложения

5.1 Приложение А Простая реализация

5.2 Приложение Б Применение модели MVC в приложении

5.3 Приложение В Применение модулей в приложении

Руководитель работы _____ А. В. Попов
подпись, дата

Задание принял к исполнению _____ С.А. Антонова
подпись, дата

РЕФЕРАТ

Выпускная квалификационная работа 91 с., 40 рис., 14 источн., 3 прил.

СИСТЕМЫ СБОРКИ, БРАУЗЕР, СЕРВЕР, FRONTEND, BACKEND, JAVASCRIPT, ПРИЛОЖЕНИЕ, МОДУЛИ

Объектом исследования являются системы сборки файлов JavaScript, использующиеся для уменьшения количества файлов и ускорения время загрузки страницы приложения.

Цель работы: разработать приложение на языке JavaScript с использованием различных систем сборки.

В процессе работы использовались: монографии, учебные пособия, научные статьи.

В результате проделанной работы было изучено взаимодействие клиентской (frontend) и серверной (backend) частей веб-приложений. Рассмотрены системы сборки, преобразующие модульные JavaScript-приложения для их эффективного использования в веб-браузере, разработано приложение на языке JavaScript.

Степень внедрения – частичная

Область применения – разработка веб-приложений

Эффективность (значимость) – данная работа направлена на выявление преимуществ и недостатков разработки приложений при помощи сборщиков модулей.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы. Современная жизнь немыслима без приложений, которые выполняются в среде веб-браузеров. Исторически сложилось так, что единственным языком программирования, который понимают браузеры, является JavaScript.

Веб постоянно расширяется, возникают все новые задачи, поэтому решения, написанные на JavaScript, с каждым годом становятся сложнее и объемнее. При этом возникает определенное противоречие в требованиях к JavaScript-коду, исполняемому в браузере.

С одной стороны, во время разработки исходный код должен быть максимально понятным человеку, его должно быть легко сопровождать, исправлять и дополнять. Для этого код необходимо структурировать, по возможности распределять по независимым модулям или функциональным компонентам, функции из которых можно импортировать. Из названий переменных и функций должен быть ясен их смысл. Общий размер исходных файлов JavaScript при разработке не имеет особого значения.

С другой стороны, во время выполнения JavaScript-код должен загружаться в браузер максимально быстро, то есть его размер должен быть минимально возможным. Структурированность кода и его понятность человеку во время выполнения не нужна. Также желательно загружать сценарии одним файлом, ведь каждое обращение к серверу за ресурсами требует отдельного HTTP-запроса и занимает дополнительное время.

Таким образом, возникает задача преобразования программ на JavaScript с целью их минификации и объединения кода из разных модулей в один файл с учетом взаимосвязей модулей друг с другом. Эта процедура, по сути похожая на компиляцию исполняемого кода из исходного текста программы, называется *сборкой веб-приложения*.

Сегодня существуют множество систем сборки проектов (например, Parcel, Webpack и Browserify), которые также называются *бандлерами*. Суть работы этих инструментов состоит в том, что они берут JavaScript-код,

хранящийся во множестве файлов, и упаковывают его в один или несколько файлов, определённым образом упорядочивая и подготавливая к работе в браузерах. Они позволяют применять препроцессоры HTML-разметки и CSS-стилей, а также умеют делить код на фрагменты, загружающиеся в браузер по необходимости. Возможности сборщиков на этом не ограничиваются, фактически они помогают организовывать процесс разработки.

Цель данной работы – разработать модульное приложение на языке JavaScript с использованием разных сборщиков проектов.

Достижение поставленной цели обусловило решение следующих **задач** исследования:

- изучить взаимодействие клиентской и серверной частей веб-приложений;
- проанализировать особенности языка JavaScript;
- исследовать исполнение сценариев JavaScript в браузере и на сервере с помощью Node.js;
- рассмотреть стандарты определения модулей;
- рассмотреть популярные системы сборки и разобрать принципы их работы.

1 Поддержка языка JavaScript в браузере и на сервере

1.1 Двухкомпонентная структура веб-приложений

Средой исполнения веб-приложения, в которой работает пользователь, является веб-браузер, запущенный на локальной машине. Особенностью данных приложений является их двухкомпонентность. Части приложения четко разделены на клиентский (*frontend*) и серверный (*backend*) программный код.

Клиент и сервер взаимодействую друг с другом в сети Интернет или в любой другой компьютерной сети при помощи высокоуровневого протокола HTTP (Hypertext Transfer Protocol) (рис. 1).

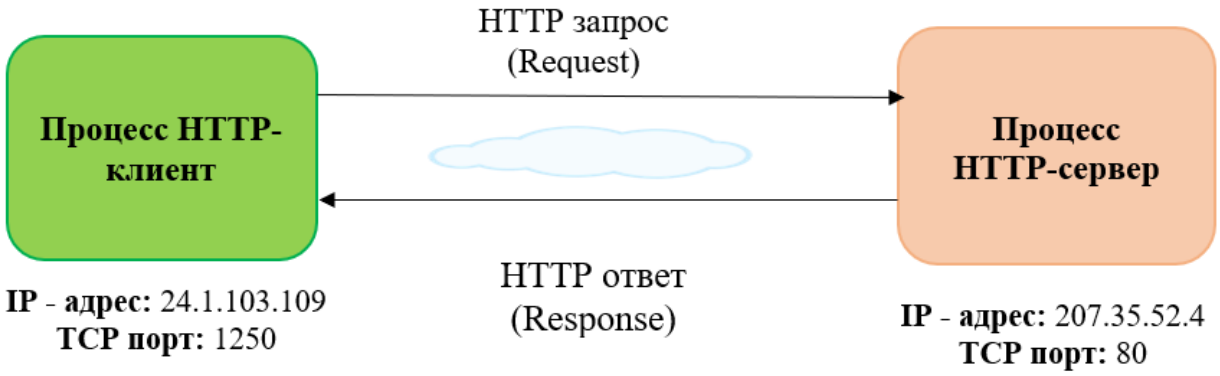

Рисунок 1 – Схема клиент-серверного приложения

Сообщения, которые посылают клиенты серверу, называются *HTTP-запросами*. Запросы имеют специальные методы, которые говорят серверу о том, как обрабатывать сообщение.

Сообщения, которые посылает сервер клиентам, называются *HTTP-ответами*, кроме полезной информации еще и специальные коды состояния, которые позволяют браузеру узнать то, как сервер обработал его запрос.

Для установления соединения между двумя процессами необходимо знать два параметра: IP-адрес сервера (идентифицирует узел сети) и TCP-порт (число, обозначающее номер порта).

После установления TCP-соединения между процессами клиента и сервера, клиент по поднятому каналу отправляет на сервер HTTP-запрос, в котором указано, какую информацию и в каком виде он хочет получить от

сервера и ждет ответа. Сервер в ответ отсылает браузеру сообщение, в котором содержится нужная информация в определенном формате (например, HTML или JSON).

Взаимодействие клиента с сервером всегда начинает клиент, сервер лишь отвечает клиенту. Клиентское и серверное программное обеспечение обычно установлено на разных машинах, но также они могут работать и на одном компьютере.

В HTTP определено несколько методов, которые указывают, какое действие нужно выполнить для запрашиваемого ресурса на сервере [1]:

- GET – запрос представления ресурса. Запросы с использованием этого метода должны только извлекать данные.
- HEAD – запрос аналогичен методу GET, но без тела ответа.
- POST – запрос для отправки сущностей к определённому ресурсу.
- PUT – замена текущего представления ресурса данными запроса.
- DELETE – удаление указанного ресурса.
- CONNECT – установка "туннеля" к серверу, определённому по ресурсу.
- OPTIONS – описание параметров соединения с ресурсом.
- TRACE – вызов возвращаемого тестового сообщения с ресурса.
- PATCH – метод для частичного изменения ресурса.

  Код ответа (состояния) HTTP показывает, был ли успешно выполнен определённый HTTP-запрос. Коды сгруппированы в 5 классов:

  1. Информационные сообщения (100 – 199).
  2. Запрос выполнен успешно (200 – 299).
  3. Сообщения о перенаправлении на другой ресурс (300 – 399).
  4. Ошибки на стороне клиента (400 – 499).
  5. Ошибки на стороне сервера (500 – 599).

1.1.1 Frontend

Frontend – это публичная часть веб-приложений и сайтов, с которой пользователь может взаимодействовать и контактировать напрямую. По сути, frontend – это всё то, что видит пользователь при открытии веб-страницы.

Для открытия веб-страницы в браузере необходимо в адресной строке браузера набрать определенный *URL* (Uniform Resource Locator), состоящий из нескольких частей, некоторые из которых являются обязательными, а некоторые – факультативными.

Рассмотрим наиболее важные части URL на примере [1]:

http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2

http:// – протокол, отображает, какой протокол браузер должен использовать. Обычно это протокол HTTP или его безопасная версия HTTPS. Интернет требует эти два протокола, но браузеры часто могут использовать и другие протоколы, например mailto: (чтобы открыть почтовый клиент) или ftp: для запуска передачи файлов;

www.example.com – доменное имя веб-сервера. В качестве альтернативы может быть использован и IP-адрес;

:80 – номер порта, для доступа к ресурсам на веб-сервере. Обычно подразумевается, что веб-сервер использует стандартные порты HTTP-протокола (80 для HTTP и 443 для HTTPS) для доступа к своим ресурсам. Порт – это факультативная составная часть URL;

/path/to/myfile.html – адрес ресурса на веб-сервере;

?key1=value1&key2=value2 – дополнительные параметры, которые браузер сообщает веб-серверу. Эти параметры – список пар ключ/значение, которые разделены символом &.

Схема работы Frontend изображена на рисунке 2.

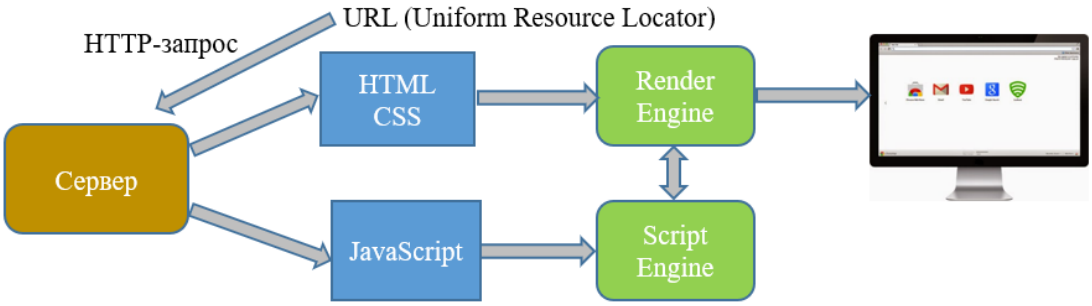

Рисунок 2 – Схема работы Frontend

После ввода URL и нажатия клавиши «Enter» браузер разбирает введенную нами строку, определяет с помощью серверов DNS на какой IP-адрес и порт нужно обратиться, т.е. браузер выступает в роли клиента.

Далее по TCP-протоколу устанавливается связь с сервером и ему посылается GET-запрос. Backend-сервер обрабатывает запрос, обращается к базе данных и посылает ответ на запрос обратно frontend-клиенту. Сервер может вернуть HTML-разметку, которая загружается в браузер, она поможет наполнить сайт необходимой информацией и расположить ее в нужных частях страницы.

Затем происходит загрузка CSS-стилей для HTML-разметки, после этого с помощью движка Render Engine результат верстки CSS и HTML выводится на экран. С помощью того же движка происходит загрузка JavaScript, в результате этого сайт становится интерактивным.

1.1.2 Backend

Backend – программно-аппаратная часть сервиса. Это набор средств, с помощью которых происходит реализация логики веб-сайта или приложения, то есть выполняется скрытая от наших глаз работа вне компьютера и браузера.

Схема работы Backend изображена на рисунке 3.

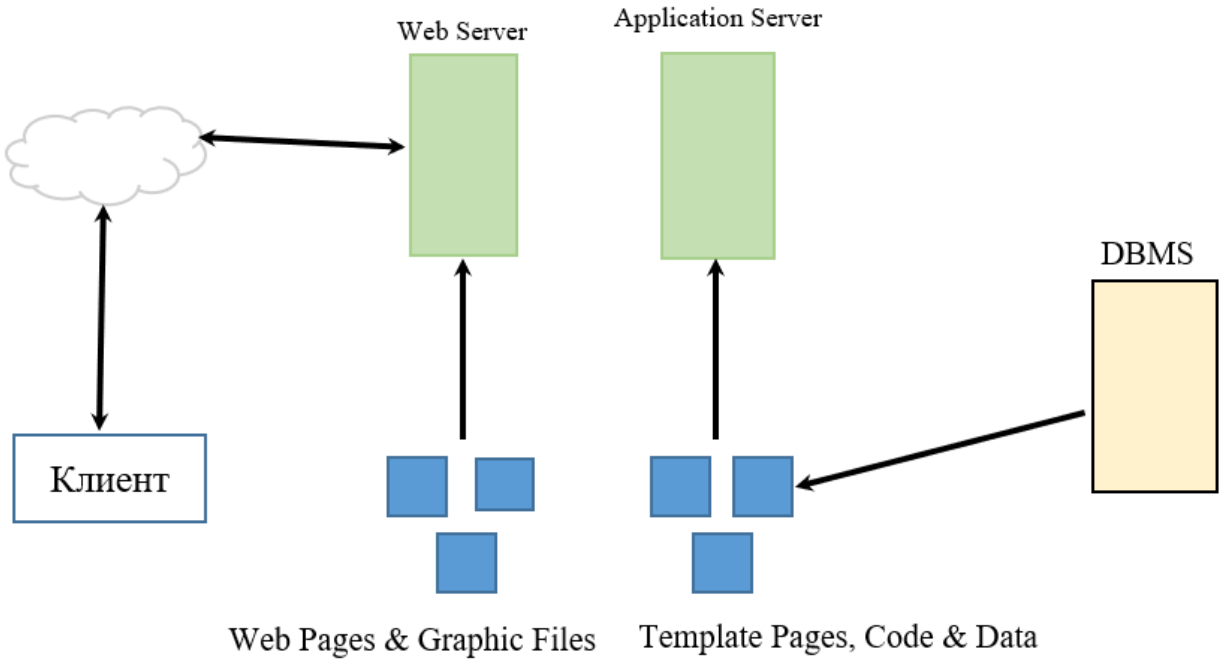

Рисунок 3 – Схема работы Backend

Теперь разберемся, что происходит на стороне сервера. Там должен быть запущен и постоянно работать процесс веб-сервера, слушающий определенный порт (80 для протокола HTTP или 443 для протокола HTTPS), на который поступают запросы от клиентов. В файловой системе сервера размещаются статические страницы с HTML-разметкой, представленные в виде файлов. Если к веб-серверу поступает GET-запрос на получение файла по определенному пути, и этот файл существует на сервере, то веб-сервер вернет клиенту его содержимое.

Если клиент запрашивает информацию, которую динамически нужно извлечь из базы данных, то для этого нужен сервер приложений, которому начальный сервер должен перебросить запрос. При этом прикладной сервер запускает программу (скрипт), которая обращается к базе данных и получает нужную информацию. Далее на основе полученных данных формируется ответ в виде HTML-страницы, которая возвращается по протоколу HTTP веб-серверу, после чего последний возвращает страницу клиенту.

1.2 Особенности языка JavaScript

Стиль написания программ на том или ином языке определяется набором идей и понятий, объединяемых термином «парадигма». Принято выделять следующие основные парадигмы [2]:

- *Процедурное программирование*, когда большая задача разбивается с помощью процедур на меньшие и это происходит это до тех пор, пока решение всех конкретных процедур не окажется тривиальным.

- *Объектно-ориентированное программирование*, при котором данные и логика их обработки структурируется в единицах, называемых классами.

- *Функциональное программирование*, кода процесс выполнения трактуется как вычисление значений функций в математическом понимании последних.

JavaScript относится к категории *многопарадигменных языков*. Он поддерживает возможность писать процедурный, классово-ориентированный и функциональный код, причем эти решения могут приниматься на уровне отдельных строк.

Один из фундаментальных принципов JavaScript – это обеспечение *обратной совместимости кода*. Это значит, что если JavaScript-код в определенный момент был признан допустимым, то в дальнейшем во внешнем окружении не могут произойти изменения, из-за которых этот код станет недопустимым. Другими словами, если код работал в браузере в 2010 году, то и через 10 лет он должен так же работать в новых браузерах.

Напротив, требование *прямой совместимости кода* означает, что включение новой языковой возможности в программу не нарушит ее работоспособности, если она будет запущена в старой среде исполнения.

JavaScript не обладает прямой совместимостью, поэтому всегда есть угроза того, что код, являющийся корректным для современного браузера, не сработает в устаревшем. Например, приведет к аварийному завершению программы такая ситуация: запуск программы, использующей новые возможности ES2019, в ядре от 2016 года.

Проблема нового и несовместимого синтаксиса решается *транспиляцией*. Этот термин описывает преобразование специальной программой исходного кода программы из одной формы в другую. Транспилятор JavaScript преобразует синтаксис новой версии языка к равносильному по функциональности старому синтаксису.

Таким образом, разработчики поручают транспилятору всю работу по построению совместимого кода, который может функционировать в среде со старым ядром JavaScript.

Кроме отнесения к той или иной парадигме, языки программирования часто делят на интерпретируемые и компилируемые. Настоящая причина, по которой важно четко понимать, является ли JavaScript тем или иным, связана с природой обработки ошибок в языке.

Интерпретируемые языки сценариев обычно выполняются строка за строкой. Исходный проход всей программы с обработкой кода до начала исполнения не применяется (рис. 4).

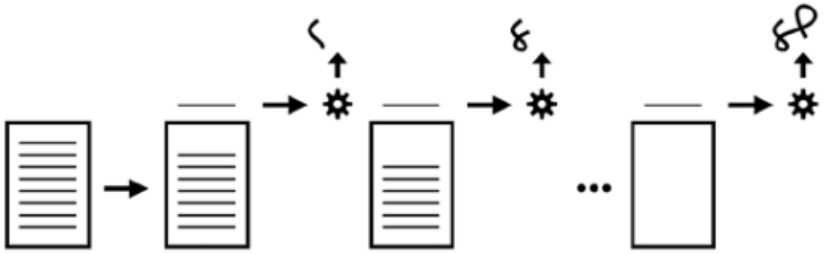

Рисунок 4 – Выполнение программ на интерпретируемом языке

Таким образом, в интерпретируемых языках ошибка в строке 6 программы не будет найдена до того момента, пока не будут выполнены строки с 1 по 5. Например, ошибка в строке 6 может возникать из-за условия времени выполнения или присутствия некорректной команды в этой строке.

В разбираемых языках исходный код проходит через этап синтаксического разбора до начала выполнения. Поэтому недействительная команда в строке 6 будет перехвачена еще в фазе разбора, до выполнения какого-либо кода программы. Что касается других ошибок, о них нужно

знать заранее, еще до неполного выполнения, обреченного на неудачу (рис. 5).

Рисунок 5 – Разбор + компиляция + выполнение

Все компилируемые языки являются разбираемыми. JavaScript также относится к разбираемым языкам, так как его исходный код перед выполнением преобразуется в абстрактное синтаксическое дерево (Abstract Syntax Tree, AST). Поэтому «ранние ошибки» в скриптах JavaScript выявляются еще до начала выполнения кода.

Разобранный код JavaScript преобразуется в двоичную форму, и именно этот код в дальнейшем выполняется (рис. 6). Точнее говоря, данная компиляция производит двоичный байт-код, который затем передается виртуальной машине для выполнения. Ядра JavaScript могут применять к генерируемому коду многопроходную JIT-обработку/оптимизацию (Just-In-Time).

Опишем пошаговую последовательность выполнения исходного кода JavaScript:

1. Исходный код сначала транспилируется (например, утилитой Babel), затем упаковывается (например, бандлером Webpack) и в этой новой форме передается ядру JavaScript.
2. Ядро JavaScript преобразует полученный код в форму AST.
3. Ядро JavaScript преобразует AST в двоичное промежуточное представление (Intermediate Representation), которое дополнительно обрабатывается оптимизирующим компилятором JIT.
4. Виртуальная машина JavaScript выполняет программу.

Эти этапы наглядно представлены на рисунке 6:

Рисунок 6 – Разбор, компиляция и выполнение JavaScript-кода

1.3 Поддержка JavaScript в браузере на стороне клиента

Рассмотрим подробно процесс отображения веб-страницы в браузере (рис. 7).

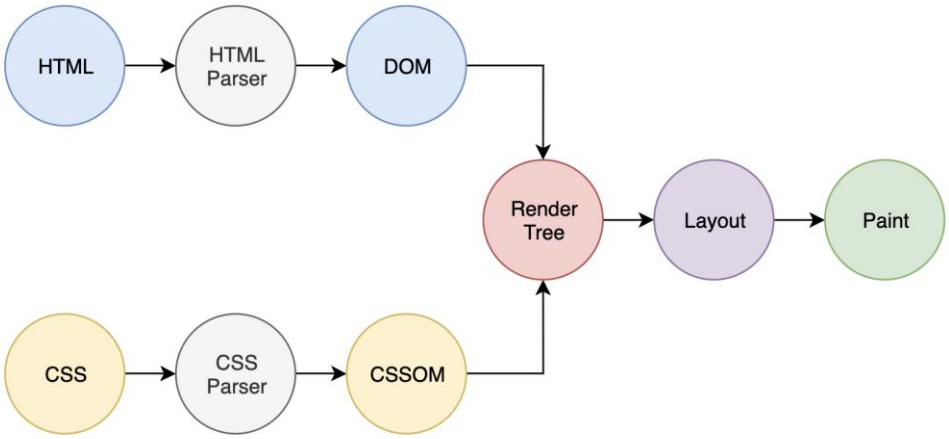

Рисунок 7 – Рендеринг веб-страницы

С сервера загружается HTML-код (текст с определенными тегами), затем он разбирается компонентом HTML Parser и на его основе в оперативной памяти формируется DOM (Document Object Model). Все это определяет структуру документа.

Затем идет подгрузка CSS-стилей, либо отдельно, либо внутри HTML-страницы, либо с других ресурсов. Далее стили разбираются компонентом CSS Parser и формируется CSSOM (CSS Object Model).

На основе DOM и CSSOM строится дерево рендеринга (render tree) – набор объектов для отрисовки. Данное дерево дублирует структуру DOM, но сюда не попадают невидимые элементы, а каждая строка текста представлена в нем в виде отдельного объекта. Каждый объект рендеринга содержит соответствующий ему объект DOM (или блок текста), и рассчитанный для этого объекта стиль. Проще говоря, дерево рендеринга описывает визуальное представление DOM.

Для каждого элемента из дерева рендеринга рассчитывается его положение на странице – происходит разметка страницы (фаза Layout на схеме). Браузеры используют поточный метод (flow), при котором в большинстве случаев достаточно одного прохода для размещения всех

элементов. Далее выполняется фаза Paint – происходит отрисовка всего необходимого в браузере, в итоге формируется статическое изображение страницы.

Для придания странице интерактивности используется язык JavaScript – это единственный язык, который поддерживает браузер. Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не позволяющих это делать.

Возможности скриптов JavaScript зависят от окружения, в котором они запускаются. В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером:

- Изменение DOM и CSS-стилей.
- Реагирование на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
- Отправка сетевых запросов на удалённые сервера, скачивание и загрузка файлов.
- Получение и установка кук, вывод сообщений пользователю.
- Сохранение данных на стороне клиента.

Возможности JavaScript в браузере ограничены политиками безопасности:

- JavaScript внутри веб-страницы не имеет прямого доступа к системным функциям операционной системы не может читать или записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям операционной системы.
- Современные браузеры позволяют скриптам JavaScript работать с файлами, но с ограниченным доступом, предоставляемым только в случае, когда пользователь выполняет определённые действия, такие

как перетаскивание файла в окно браузера или его выбор с помощью тега `<input>`.

- Различные окна и вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).
- Это называется «Политика одинакового источника» (Same Origin Policy). Чтобы обойти данное ограничение, обе страницы должны согласиться с этим и содержать JavaScript-код, который специальным образом обменивается данными.
- JavaScript не может работать с локальными СУБД.

Для работы с JavaScript внутри браузера мы должны подключить скрипт напрямую в HTML-файл. Самый простой способ – написать команды внутрь тега `<script>`, расположенного в теле страницы. Тег `<script>` всегда закрывается при помощи `</script>`, даже если он не содержит кода и ссылается на файл скрипта, иначе часть страницы будет обработана как скрипт. Пример подобного кода:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1>Первый заголовок</h1>
  <script>
    console.log ('Hello, world');
  </script>
</body>
</html>
```

Когда движок Script Engine доходит до тега `<script>`, он сначала читает и выполняет код, а только потом продолжает читать страницу дальше.

В браузере вывелся тег `<h1>`. Чтобы посмотреть результат работы тега `<script>`, нужно в браузере открыть инструменты разработчика и перейти на вкладку **Console** – там будет выведено наше приветствие (рис. 8).

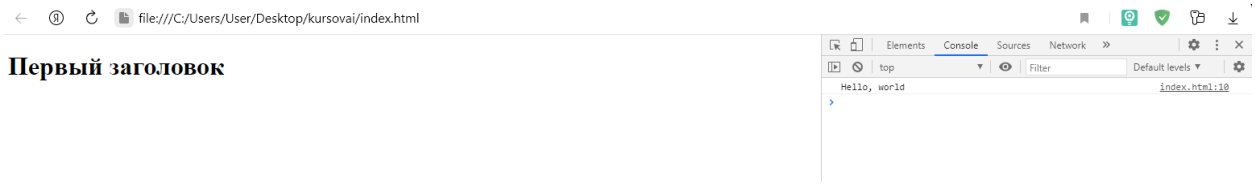

Рисунок 8 – Результат выполнения кода

Обычно JavaScript-код не пишут непосредственно в HTML-файле, а подключают отдельный файл со скриптом с помощью атрибута `src`:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
</head>
<body>
  <h1>Первый заголовок</h1>
  <script src="01_hello.js"></script>
</body>
</html>
```

В данном случае в качестве значения атрибута `src` указан путь к файлу, который впоследствии будет загружен и интерпретирован как сценарий JavaScript. Когда страница ссылается на другой URL и включает его в себя, браузер подгружает этот файл и включает его в страницу. Результат выполнения кода представлен ниже (рис 9).

Рисунок 9 – Результат выполнения кода

Благодаря JavaScript страницы выглядят более привлекательными в глазах пользователей. Скрипты JavaScript создают динамичные эффекты, всплывающие окна и другие объекты. Однако подключение объемных скриптов, содержащих много комментариев или других служебных символов, замедляет загрузку страницы.

Не затрагивая функциональность скрипта, код JavaScript можно сократить, упростив тем самым загрузку динамичных эффектов. Ведь если

этого не сделать, страница может подвисать до тех пор, пока полностью не загрузится код. Для уменьшения объёма кода и ускорения загрузки необходимо прибегнуть к *минимизации скрипта*, то есть удалению из кода всех несущественных символов (комментарии и незначащие пробелы, переносы строк, символы табуляции).

Обфускация – это альтернативный способ сокращения исходного кода. Так же, как минимизация, она удаляет пробельные символы и вырезает комментарии, но в дополнение обфускация изменяет сам код. Во время обфускации имена функций и переменных заменяются на более короткие, что делает код более компактным, но менее читабельным. Тем самым код уменьшается настолько, насколько это возможно. Одной из самых распространенных утилит для обфускации является Dojo Compressor.

1.4 Исполнение сценариев JavaScript на сервере

JavaScript работает не только в браузере, но и на локальной машине и сервере с помощью Node.js – программной платформы, созданной на основе движка V8 от Google, которая способна транслировать JavaScript в исполняемый код машины [3].

В 1996 году в компании Netscape были попытки создания серверного JavaScript (Server-side JavaScript – SSJS), однако технология не получила распространения. Платформу Node.js в 2009 году, после двухлетней экспериментальной деятельности в направлении серверных компонентов, создал Райан Дал. В ходе своих исследований он пришёл к выводу, что вместо традиционной модели параллелизма на основе потоков следует обратиться к событийно-ориентированным системам.

Данная модель была выделена благодаря своей простоте и низкому потреблению ресурсов. Главная цель системы Node.js – обеспечение максимально простого метода построения масштабируемых серверов в сети.

Основные особенности Node.js – скорость выполнения, связанная с неблокирующей архитектурой платформы, и простота в освоении и использовании.

В среде Node.js выполняется код, написанный на JavaScript. Это означает, что миллионы фронтенд-разработчиков, которые уже пользуются JavaScript в браузере, могут писать и серверный, и клиентский код на одном и том же языке программирования без необходимости изучать совершенно новый инструмент для перехода к серверной разработке. В браузере и на сервере используются одинаковые концепции языка.

Кроме того, в Node.js можно оперативно переходить на использование новых стандартов ECMAScript по мере их реализации на платформе. Для этого не нужно ждать, пока пользователи обновят браузеры, так как Node.js – это серверная среда, которую полностью контролирует разработчик.

Также на JavaScript часто пишут вспомогательные утилиты для автоматизации:

- *Линтеры* – программы, которые проверяют код на соответствие стандартам в соответствии с определённым набором правил. Правила описывают отступы, названия создаваемых сущностей, скобки, математические операции, длину строк и множество других аспектов. Главная задача линтера – сделать код единообразным, удобным для восприятия и самим программистом, и другими людьми, которые будут читать код.
- *Транспилеры* – интерпретаторы, преобразующие код программы, написанной на одном языке, в аналогичный код, но уже на другом языке программирования.

2 Разработка модульных сценариев JavaScript

2.1 Структуризация и модульность исходного кода

Вне зависимости от того, на каком языке программирования написана программа, она должна быть понятна человеку. Если писать команды непрерывным потоком, то программа быстро становится нечитабельной, поэтому код нужно разделять код на структурные единицы – *функции*.

Функции позволяют разбивать сложные задачи на более мелкие и простые, что кардинально снижает общую сложность программы. Кроме того, после объявления функции её можно вызывать много раз. Это позволяет избежать дублирования кода и сводит к минимуму вероятность возникновения ошибок при копировании/вставке кода.

Функции также могут использоваться и в других программах на том же языке, уменьшая объем кода, который нужно писать с нуля каждый раз.

За счет того, что переменные в функциях локализуются, появляется возможность написать отдельную функцию, не заботясь при этом об остальных частях кода. В конечном итоге может оказаться так, что функций, имеющих уже собственный алгоритм, становится тоже очень много и все это находится в одном файле.

Модули дают возможность избежать подобных проблем. *Модулем* называется часть программы, которая определяет, на какие другие компоненты она опирается и какие функциональные возможности предоставляет другим частям программы через свой интерфейс.

Паттерн модуля предназначен для группировки данных и поведения в логических единицах. С первых дней существования JavaScript-модули были важным и распространенным паттерном, который задействовался в множестве сценариев даже без специального синтаксиса поддержки модуля.

Основные признаки классического модуля – внешняя функция, которая выполняется минимум один раз и возвращает экземпляр модуля с одной или несколькими функциями, способными работать с внутренними данными экземпляра модуля. Так как модуль в этой форме – всего лишь функция,

вызов которой создает экземпляр модуля, такие функции также описываются как фабрики модулей.

В JavaScript существует принцип инкапсуляции и сокрытия данных – данные должны быть скрыты внутри объекта, прямой доступ к ним внешним объектам и пользователям должен быть запрещен. По умолчанию свойство не должно быть видно извне, необходимо иметь специальный метод, который будет его возвращать. В JavaScript есть возможность сымитировать скрытые свойства.

Для примера рассмотрим функцию `createEmployee()`, которая создает объекты с разными значениями параметров, принимая в качестве аргументов имя сотрудника и его зарплату (рис. 10).

```
C: > Users > Home > Desktop > JS modules.js > ...
1  function createEmployee(name, salary) {
2    return {
3      getName: () => name,
4      raiseSalary: (percent) => {
5        salary *= 1 + percent / 100;
6      },
7      getSalary: () => salary,
8    };
9  }
10
11 const bond = createEmployee("James Bond", 100000);
12 console.log(bond.getSalary());  100000
13
14 console.log({ bond });  { bond: { getName: [λ: getName], raiseSalary: [λ: raiseSalary], getSalary: [λ: getSalary] } }
15
16 //bond.raiseSalary(20);
17 //console.log(bond.getSalary());
```

```
ВЫХОДНЫЕ ДАННЫЕ   КОНСОЛЬ ОТЛАДКИ   ТЕРМИНАЛ
Quokka 'modules.js' (node: v16.14.2)

Reveal in value explorer
100000 at bond.getSalary() modules.js:12:3

Reveal in value explorer
{ bond:
  { getName: [λ: getName],
    raiseSalary: [λ: raiseSalary],
    getSalary: [λ: getSalary] } }
 at { bond } modules.js:14:3
```

Рисунок 10 – Реализации паттерна «Модуль» (пример 1)

Функция `createEmployee()` возвращает объект, который сохраняется в переменную `bond`.

У объекта `bond` нет свойств с данными, есть только методы `getName`, `raiseSalary`, `getSalary`. Но данные на самом деле есть, так как они были изначально присвоены аргументам функции.

Впоследствии значения, присвоенные этим данным объекта, можно поменять (рис. 11). В нашем пример это происходит при вызове метода `raiseSalary`, увеличивающего значение переменной `salary`. Изначально метод `getSalary` возвращал значение 100000, а теперь возвращает значение 120000, но самого свойства `salary` мы так и не видим.

```
C: > Users > Home > Desktop > JS modules.js > ...
1  function createEmployee(name, salary) {
2    return {
3      getName: () => name,
4      raiseSalary: (percent) => {
5        salary *= 1 + percent / 100;
6      },
7      getSalary: () => salary,
8    };
9  }
10
11 const bond = createEmployee("James Bond", 100000);
12 console.log(bond.getSalary());  100000
13
14 console.log({ bond });  { bond: { getName: [λ: getName], raiseSalary: [λ: raiseSalary], getSalary: [λ: getSalary] } }
15
16 bond.raiseSalary(20);
17 console.log(bond.getSalary());  120000
```

```
ВЫХОДНЫЕ ДАННЫЕ   КОНСОЛЬ ОТЛАДКИ   ТЕРМИНАЛ
Quokka 'modules.js' (node: v16.14.2)

Reveal in value explorer
100000 at bond.getSalary() modules.js:12:3

Reveal in value explorer
{ bond:
  { getName: [λ: getName],
    raiseSalary: [λ: raiseSalary],
    getSalary: [λ: getSalary] } }
at { bond } modules.js:14:3

Reveal in value explorer
120000 at bond.getSalary() modules.js:17:3
```

Рисунок 11 – Реализация паттерна «Модуль» (пример 2)

Функция `CreateEmployee()` возвращает объект с тремя свойствами функционального типа: `getName` возращает `name`, `raiseSalary` изменяет значение `salary`, `getSalary` возвращает `salary`. При ее вызове происходит передача двух значений (James Bond, 100000), которые попадают в функцию под именами `name` и `salary`.

При выполнении функции возвращается объект. Но далее идет обращение к названиям параметров функции, то есть параметры функции по-прежнему доступны, хотя функция `createEmployee()` и завершила свою работу. Это называется *свойством замыкания*.

Замыкание – это функция, у которой есть доступ к области видимости, сформированной внешней по отношению к ней функции даже после того, как эта внешняя функция завершила работу. Это значит, что в замыкании могут храниться переменные, объявленные во внешней функции и переданные ей аргументы.

Так при вызове `getSalary` `(console.log(bond.getSalary())` выполняется функция, которая продолжает работать с локальными переменными, полученными при первоначальном вызове. Таким образом реализуется паттерн «Модуль».

Интерфейс делает часть модуля доступной для внешнего мира и скрывает остальные детали реализации. Отношения между модулями называются *зависимостями*. Если модулю требуется фрагмент из другого модуля, говорят, что он зависит от этого модуля. Когда данный факт четко описан в самом модуле, его можно использовать для выяснения, какие другие модули необходимы для того, чтобы можно было применять данный модуль, и автоматически загружать зависимости.

Одно из преимуществ построения программы из отдельных частей с возможностью запускать их по отдельности – возможность использования одного и того же фрагмент кода в разных программах.

Предположим, что у нас есть функция, которая не имеет зависимостей, тогда мы можем скопировать её и использовать в другой программе. Но потом, если обнаружится ошибка в коде, то мы исправим ее той программе, с которой работаем сейчас, но можем забыть исправить в другой.

Решить эту проблему помогают пакеты. *Пакет* – это фрагмент кода, который можно распространять (копировать и устанавливать). Пакет может содержать один или несколько модулей, а также информацию о том, от каких

других пакетов он зависит. Пакет обычно поставляется в комплекте с документацией, объясняющей, что он делает.

Если в пакете обнаружена проблема или добавлена новая функция, то пакет обновляется. После этого программы, которые от него зависят (и которые также могут быть пакетами), могут обновиться до новой версии. Поддержка такого стиля работы требует соответствующей инфраструктуры. Необходимо место для хранения и поиска пакетов, а также удобный способ их установки и обновления.

В мире JavaScript данная инфраструктура предоставляется онлайн-сервисом NPM (https://www.npmjs.com/), с которого можно загружать (и где можно размещать) различные пакеты. В составе платформы Node.js поставляется утилита `npm`, помогающая устанавливать пакеты и управлять ими.

В JavaScript имеется несколько альтернативных стандартов определения модулей: AMD, CommonJS и ES6.

2.2 Модули стандарта AMD

Стандарт AMD возник на основе Dojo – набора инструментальных средств для написания клиентских веб-приложений на JavaScript. Стандарт AMD создан специально для браузеров и позволяет определять модули и их зависимости. В настоящее время наиболее распространенной реализацией стандарта AMD является загрузчик модулей RequireJS [4,5,6].

В качестве примера рассмотрим определение небольшого модуля, у которого имеется зависимость от библиотечного модуля jQuery:

```
// с помощью функции define (), определяется модуль, его
зависимости и фабричная функция, создающая модуль
define ('MouseCounterModule ',  [' jQuery '], $=>
let numClicks = 0;
const handeClick = () => {
    alert (++numClicks);
  };
//открытый интерфейс модуля
return {
  countClicks: () => {$(document).on("click",  handeClick);}
```

```
};
});
```

В стандарте AMD определена функция `define()`, которой передаются следующие аргументы:

- Идентификатор вновь созданного модуля, которым можно будет воспользоваться в дальнейшем, чтобы затребовать модуль из других частей системы.
- Список идентификаторов модулей, от которых зависит текущий модуль, т.е. обязательных модулей.
- Фабричная функция, инициализирующая модуль, которой передаются в качестве аргументов используемые модули.

В данном примере кода используется функция `define()` по стандарту AMD, чтобы создать модуль с идентификатором `MouseCounterModule` и зависимостью от библиотечного модуля jQuery. В силу этой зависимости по стандарту AMD функция сначала запрашивает библиотечный модуль jQuery, на что может уйти время, если файл должен быть загружен с удаленного сервера. Такое действие выполняется асинхронно во избежание блокировки. Как только все зависимости будут загружены и интерпретированы, вызывается фабричная функция модуля с одним аргументом для каждого запрашиваемого модуля. В нашем примере указан один аргумент, поскольку новому модулю требуется только библиотечный модуль jQuery. Новый модуль создается в теле фабричной функции таким же образом, как и при использовании паттерна Модуль, то есть путем возврата объекта, делающего видимым открытый интерфейс модуля.

Стандарт AMD предоставляет ряд интересных возможностей:

- Автоматическое разрешение зависимостей, избавляющее от необходимости думать о порядке, в котором следует включать модули.
- Асинхронная загрузка модулей, исключающая блокировки.
- Определение нескольких модулей в одном файле.

2.3 Модули стандарта CommonJS

Наиболее широко применяемый подход для построения системы модулей JavaScript называется CommonJS. Этот стандарт используется в Node.js, а также в большинстве пакетов NPM [4,5,6].

В стандарте CommonJS применяется файловая структура модулей, что позволяет сохранять модули в виде отдельных файлов. Для каждого модуля доступна переменная `module` со свойством `exports`, которое можно расширять дополнительными свойствами. В конечном счёте содержимое свойства `module.exports` используется в качестве открытого интерфейса модуля.

Если требуется воспользоваться модулем в других частях приложения, его сначала нужно запросить. Файл модуля будет загружен синхронно, после чего станет доступен его открытый интерфейс.

Именно поэтому стандарт CommonJS чаще всего используется на стороне сервера, где модули загружаются относительно быстро, поскольку для этого достаточно лишь выполнить операцию чтения в файловой системе, а не на стороне клиента, где модули приходиться загружать из удаленного сервера и их синхронная загрузка зачастую приводит к блокировке.

В качестве примера вновь рассмотрим определение модуля `MouseCounterModule`, но на этот раз по стандарту CommonJS:

```
// MouseCounterModule.js
const $require("jQuery"); // запрос на синхронную загрузку
библиотечного модуля jQuery
let numClicks = 0;
const handeClick = () => {
    alert (++numClicks);
  };
// изменить свойство module.exports, чтобы указать открытый
интерфейс модуля
module.exports = {
  countClicks: () => {$(document).on("click",  handeClick);}
};
```

Чтобы включить модуль в другой файл, нужно воспользоваться оператором `require`:

```
const MouseCounterModule = require("MouseCounterModule.js");
```

```
MouseCounterModule.countClicks()
```

Принцип действия стандарта CommonJS предполагает распределение модулей по отдельным файлам, поэтому любой код, размещаемый в файловом модуле, становится частью этого модуля.

Следовательно, отпадает необходимость заключать переменные в оболочку немедленно вызываемых функций. Все переменные, определяемые в модуле, безопасно содержатся в области видимости текущего модуля и никак не влияют на глобальную область видимости. Например, переменные (`$`, `numClicks` и `handleClick`) оказываются в области видимости рассматриваемого здесь модуля, несмотря на то, что они определены в коде верхнего уровня (т.е. за пределами всех функций и блоков), что фактически делает их глобальными переменными в стандартных файлах JavaScript.

В этом случае важно заметить, что из внешнего модуля доступны только переменные и функции, видимые через свойство объекта `module.exports`. Сама процедура – такая же, как и при использовании проектного шаблона Модуль, только вместо возврата нового объекта в среде заранее создается объект, который может быть расширен свойствами интерфейса модуля.

CommonJS-модули имеют следующие преимущества:

- Простой синтаксис. Достаточно указать только свойства `module.exports`, оставив остальную часть модуля практически такой же, как и при написании стандартного кода JavaScript. Подключение модулей также осуществляется просто, достаточно вызвать функцию `require()`.
- Формат CommonJS является стандартным для платформы Node.js. Благодаря этому тысячи пакетов становятся доступными через утилиту npm – диспетчер пакетов среды Node.js.

Самый крупный недостаток стандарта CommonJS заключается в том, что он разработан без учета среды браузеров. В интерпретаторе JavaScript браузера отсутствует поддержка переменной `module` и свойства `exports`,

поэтому CommonJS-модули приходится упаковывать в понятном для браузера формате с помощью специальных помощью утилит-сборщиков, например Browserify.

2.4 Модули по стандарту ES6

Модули EcmaScript, включенные в стандарт язык JavaScript в ES6, служат тем же целям, что и модули AMD и CommonJS. Но подход к реализации сильно отличается [4,5,6].

Во-первых, не существует функции-обертки для определения модуля. Контекстом обертки является файл. Модули ESM всегда базируются на файлах: один файл – один модуль.

Во-вторых, не обязательно явно взаимодействовать с API модуля; достаточно воспользоваться ключевым словом `export`, чтобы добавить переменную или метод в его определение открытого API. Если нечто определяется в модуле, но не экспортируется, то оно остается скрытым.

По сути, модули ES6 являются одиночками: в программе они существуют только в единственном экземпляре, который создается при первом импортировании, а все последующие команды импортирования просто получают ссылку на тот же экземпляр.

В стандарте ES6 модули определяется таким образом, чтобы можно было использовать преимущества обоих предыдущих стандартов:

- подобно стандарту CommonJS, модули обладают относительно простым синтаксисом и распределяются по отдельным файлам;
- подобно стандарту AMD, модули поддерживают асинхронную загрузку в браузере.

2.5 Системы сборки

При разработке кода необходимо, чтобы он был понятен человеку, текст имел структурные отступы и был распределен по разным папкам и файлам, имена переменных и функций соответствовали их смыслу.

Однако при подключении скрипта на веб-странице нужно, чтобы он занимал минимальный допустимый размер, поэтому возникает задача преобразования исходного кода в один результирующий файл. Это можно сделать с помощью специальных систем сборки (Browserify, Parcel, Webpack и пр.).

Суть работы бандлеров (сборщиков проектов) заключается в том, что они берут JavaScript-код, содержащийся во множестве файлов, и упаковывают его в один или несколько файлов, определённым образом упорядочивая и подготавливая к работе в браузерах. Кроме этой основной задачи бандлеры могут поддерживать дополнительные функции: запуск препроцессоров HTML и CSS, минификация JavaScript-кода и CSS-стилей, сжатие изображений, разделение кода на фрагменты, загружающиеся по необходимости. В целом можно сказать, что бандлеры помогают организовывать процесс веб-разработки.

2.5.1 Parcel

Упаковщик для веб-приложений Parcel обеспечивает быструю работу с использованием многоядерной обработки и не требует настройки. Как и другие бандлеры, Parcel преобразует отдельные JavaScript-модули их в один минифицированный файл (бандл), который подключается к HTML-странице. Это позволяет повысить производительность сайта, так как браузеру не нужно загружать много скриптов по отдельности [7,8].

Parcel имеет встроенный сервер разработки, который автоматически пересобирает приложение при изменении исходных файлов.

Parcel преобразует дерево ресурсов в дерево бандлов. Другие упаковщиков, как правило, основываются на сценариях JavaScript с добавлением других форматов, например, встраиванием в виде строк в JS-файлы. Parcel не зависит от типа файла — он будет работать с ресурсами любых типов, не требуя при этом дополнительной настройки.

Бандл в Parcel создается за три шага:

1. **Построение дерева ресурсов.** Parcel принимает в качестве входных данных один элемент ресурса, который может быть любого типа: JavaScript, HTML, CSS, изображение и т.д. Ресурсы анализируются, их зависимости извлекаются, и они преобразуются в их окончательную скомпилированную форму – дерево ресурсов.

2. **Построение дерева бандлов.** После создания дерева ресурсов они помещаются в дерево бандлов. Бандл создаётся для входного ресурса, а для динамических вызовов `import()` создаются дочерние бандлы, что в итоге приводит к разделению кода.

При импорте ресурсов другого типа создаются родственные бандлы. Например, если импортировать CSS-файл из JavaScript, он будет помещен в соседний бандл с соответствующим бандлом для JavaScript.

Если ресурс требуется более чем в одном бандле, он поднимается до ближайшего общего предка в дереве бандлов, поэтому он не включается более одного раза.

3. **Упаковка.** После построения дерева бандлов, каждый бандл записывается в файл упаковщиком, специфичным для данного типа файла. Упаковщики знают, как объединить код из каждого ресурса в конечный файл, который будет загружен браузером.

Отметим следующие достоинства данного сборщика:

- **Быстрая сборка.** Parcel использует worker process для многопоточной сборки, а так же имеет свой файловый кэш для быстрой пересборки при последующих изменениях.
- **Сборка вложений разных типов**. Из коробки поддерживается JavaScript стандарта ES6, TypeScript, CoffeeScript, HTML, SCSS, Stylus, raw-файлы. Дополнительных плагинов при этом подключать не требуется.
- **Автоматические преобразования**. Весь код автоматически проходит через Babel, PostCSS, PostHTML.

- **Разделение кода без лишней конфигурации.** Используя динамический `import()`, Parcel разделяет бандл для возможности быстрой начальной загрузки точки входа в приложение
- **Горячая перезагрузка**. При сохранении изменений происходит автоматическое применение их в браузере.
- **Дружелюбный вывод ошибок.** При возникновении ошибки подсвечивается фрагмент кода, в котором она произошла.

2.5.2 Webpack.

Сборщик с открытым исходным кодом Webpack создан, в первую очередь, для обработки модулей JavaScript, но также может преобразовывать внешние ресурсы, такие как HTML, CSS и изображения, если включены соответствующие загрузчики [9].

Webpack имеет следующие возможности:

- **Модульность.** С помощью Webpack можно разбить код своего проекта (во время разработки) на множество модулей, а потом воедино собрать их в один файл. Модульность позволяет разделить продукт на две версии: release (окончательная версия) и production (в производстве).
- **Транспиляция.** Webpack использует Babel для автоматического преобразования кода с новых стандартов JavaScript на более ранний для обеспечения работоспособности приложения в старых браузерах.
- **Встроенный сервер.** Webpack имеет свой собственный сервер с режимом контроля изменений. При изменении одного файла Webpack отображает изменения в проекте.
- **Разделение и оптимизация.** Webpack способен разделить релизный выходной файл (общий со всеми модулями) на несколько файлов меньшего размера. Это позволяет избежать чрезмерной нагрузки на браузер пользователя при первой загрузке приложения и помогает ускорить загрузку больших проектов (например, CRM-систем и крупных веб-приложений).

Можно выделить два основных принципа философии Webpack:

- **Что угодно может быть модулем.** Модулями могут быть как скрипты JavaScript, так и CSS-файлы, HTML-файлы или изображения. Таким образом, можно разбивать любой артефакт на меньшие части и использовать их повторно.
- **Загружается только то, что нужно и когда нужно.** Обычно сборщики модулей берут все модули и генерируют из них один большой файл bundle.js. Но во многих приложениях размер такого файла может достигать размера 10-15 мегабайт, а это уже слишком много. Потому Webpack оснащен рядом функций, позволяющих делить код и генерировать множество bundle-файлов, а также асинхронно загружать необходимые части приложения тогда, когда это нужно.

2.5.3 Browserify

Browserify может упаковывать модули Node.js для подключения их в веб-браузере. Таким образом при работе в браузере становится возможным использовать стиль модулей Node.js с использованием выражения `require("./наш_файл.js")` для подключения файлов JavaScript [10].

Аналогичным образом можно использовать публичные модули из NPM. С помощью импорта модулей не нужно тратить время на скачивания библиотек, которые можно подключить, используя `require()` (предварительно нужно проверить, что они установлены через npm).

Достоинства сборщика Browserify:

- Возможность повторного использования одного и того же кода и библиотек на сервере (в приложениях Node.js) и на клиенте.
- Поддержка модулей CommonJS.
- Browserify поддерживает гибкие параметры преобразования для подключения с помощью `require()` файлов, не являющихся JavaScript-сценариями.

Недостатки:

- Не все узлы стандартной библиотеки поддерживаются.
- Собственные модули расширения не поддерживаются.
- Библиотека Browserify для поддержки Node API увеличивает размер получаемого кода.

3 Пример разработки и сборки модульного веб-приложения

Рассмотрим процесс разработки веб-приложения, в котором при нажатии на кнопки будет выводиться необходимая информация, а именно: список всех друзей пользователя; список друзей, которые в данный момент находятся в ВКонтакте; новостная лента заданного пользователя и расположение друзей пользователя на карте. Для визуализации информации из соцсети ВКонтакте используется API. С помощью этого интерфейса будем получать информацию из базы данных с использованием HTTP-запросов к специальному серверу.

3.1 Простая реализация

В данном разделе опишем последовательность действий при разработке приложения с помощью одного файла. Программа будет предоставлять пользователю следующие функцию: вывод всех друзей пользователя в социальной сети ВКонтакте.

Многие веб-сервисы предоставляют общедоступный API (Application Program Interface) – набор методов, к которым можно обращаться из других приложений или веб-страниц. Используя данные методы, можно общаться с сервисом, обмениваться данными с сервером, где он расположен.

Используем для разработки приложения API социальной сети ВКонтакте. Методы, которые доступны в этом API, описаны в документации для разработчиков на сайте https://vk.com (рис. 12).

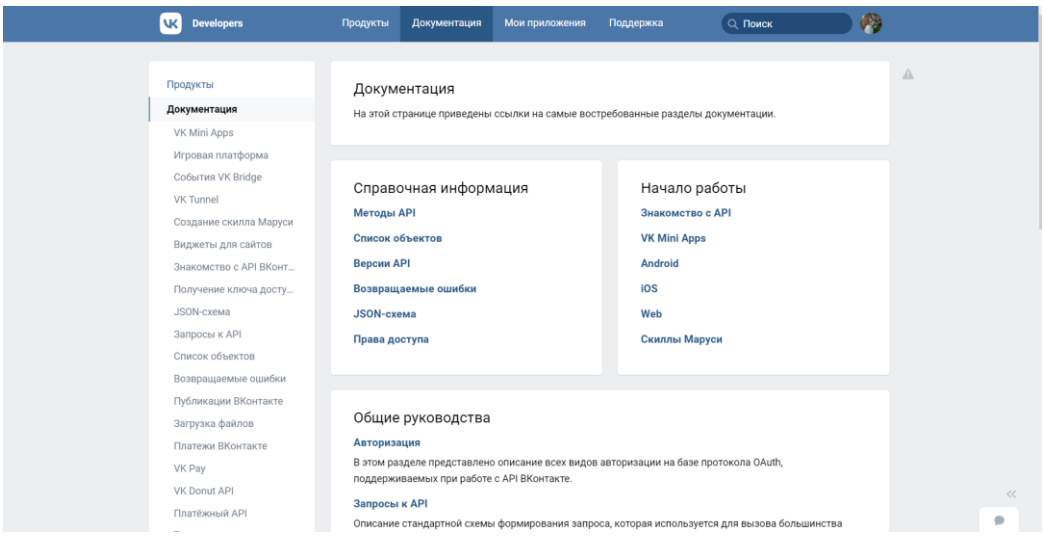

Рисунок 12 – Пример методов из документации разработчика

На данном сайте предоставлено много методов для управления различными компонентами социальной сети: учетными записями, виджетами, фотографиями, обсуждениями в группе, сообществами, сообщениями, новостями и т. д. Для написания приложения необходимы методы, которые предоставляют доступ к списку друзей (рис. 13), возвращают расширенную информацию о пользователе (рис. 14).

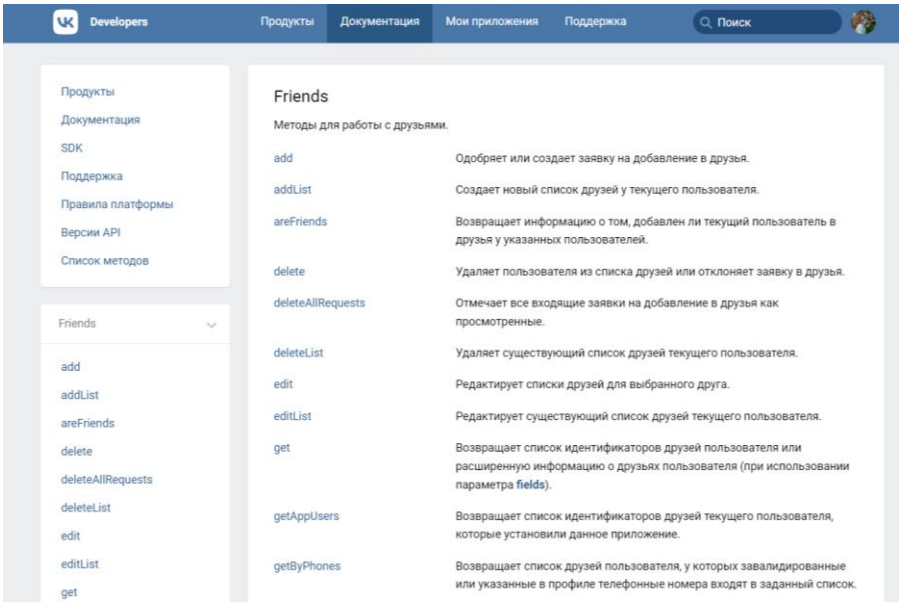

Рисунок 13 – Примеры методов для работы с друзьями

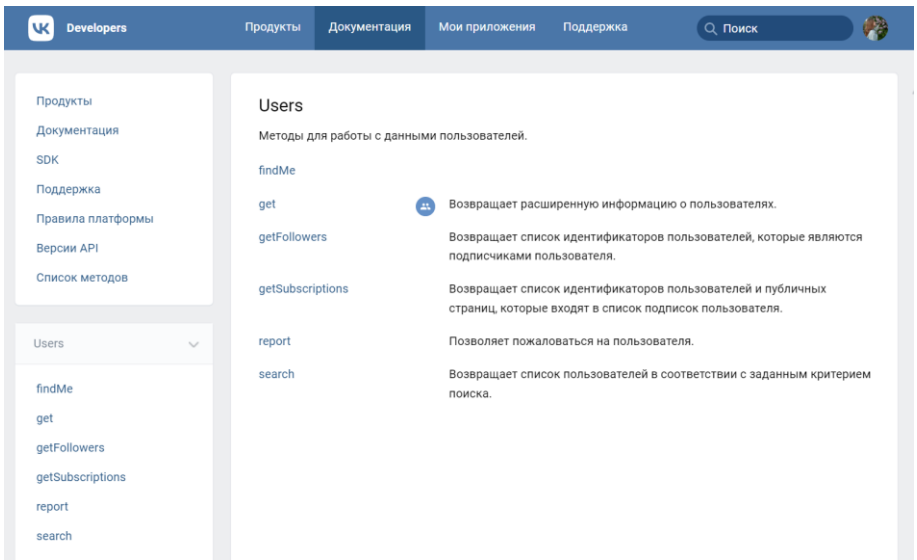

Рисунок 14 – Примеры методов для работы с данными пользователя

Получить с сервера список друзей можно с помощью метода `friends.get()` (рис. 15). На формирование этого списка влияют дополнительные параметры, которые передаются в данный метод.

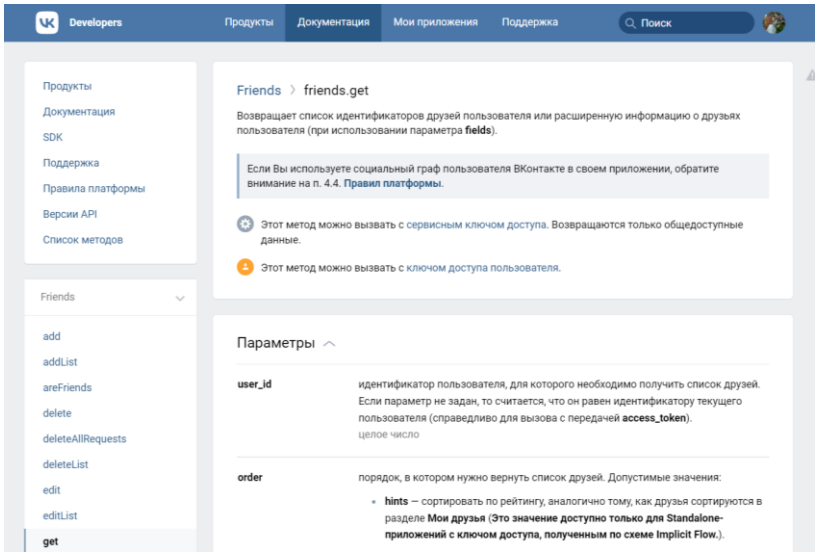

Рисунок 15 – Метод friends.get()

В частности, параметр `fields` позволяет указать, какая именно информация о друзьях нас интересует (например, статус в социальной сети, день рождения, фотография, дата последнего посещения страницы и т. п.) (рис. 16).

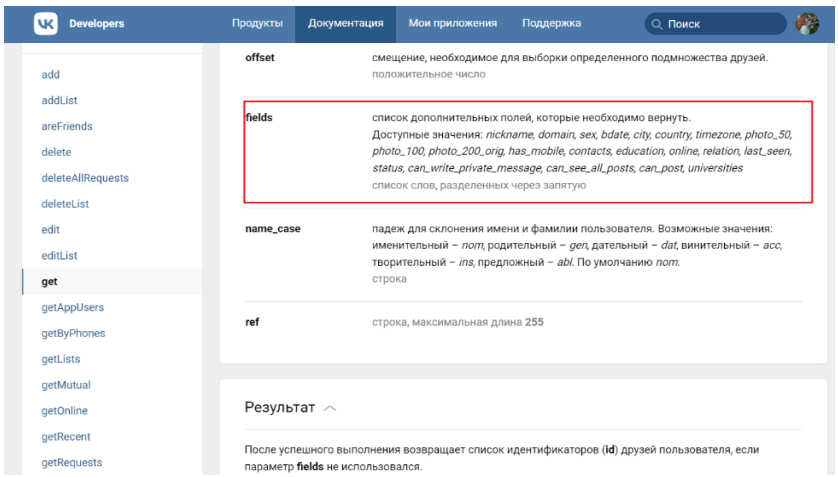

Рисунок 16 – Список дополнительных полей для параметра fields

Внизу на этой же странице можно в онлайн-режиме посмотреть, какие JSON-объекты возвращает метод `friends.get()` при различных значениях входных параметров (рис. 17).

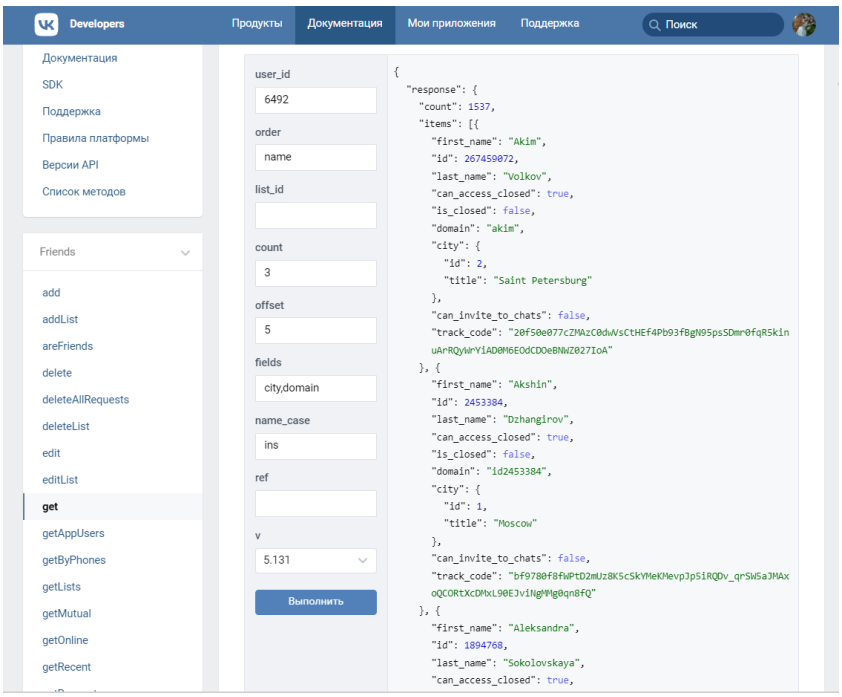

Рисунок 17 – JSON-объекты, которые возвращает метод friends.get()

Формат возвращаемых данных (состав и названия полей в объекте) зависит от версии API, которую можно выбрать из раскрывающегося списка в поле v.

Для работы с данными пользователя следует использовать метод `users.get ()` (рис. 18).

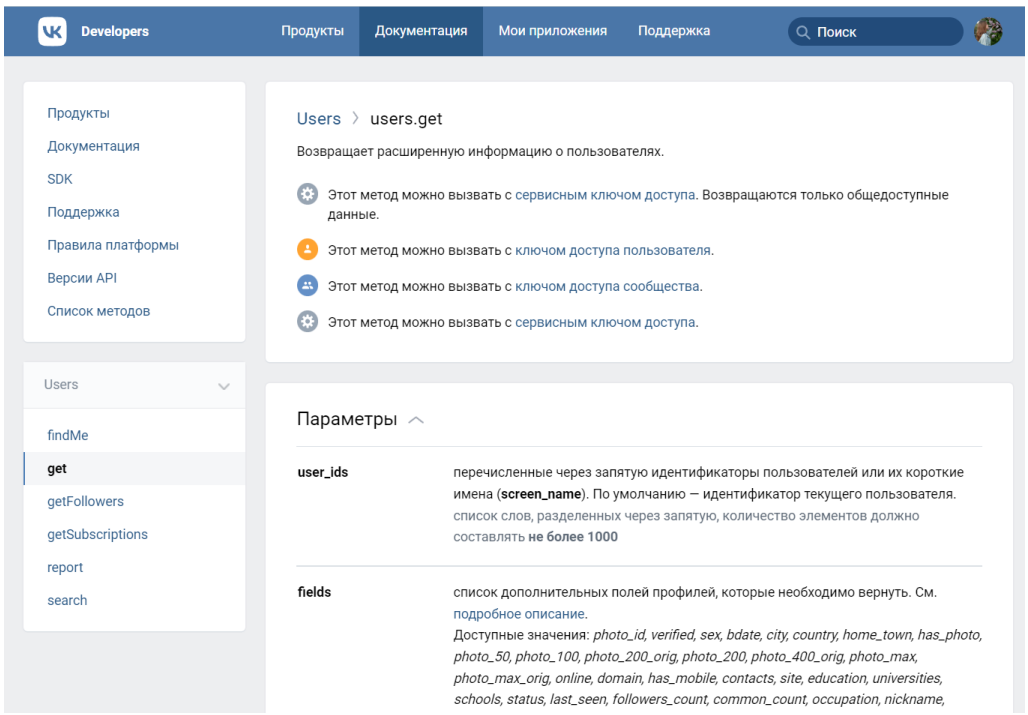

Рисунок 18 – Метод users.get()

Для того, чтобы сервер ВКонтакте обрабатывал поступающие запросы, нужно предварительно зарегистрировать на вкладке **Мои приложения** сайта https://vk.com/dev специальное приложение (рис. 19).

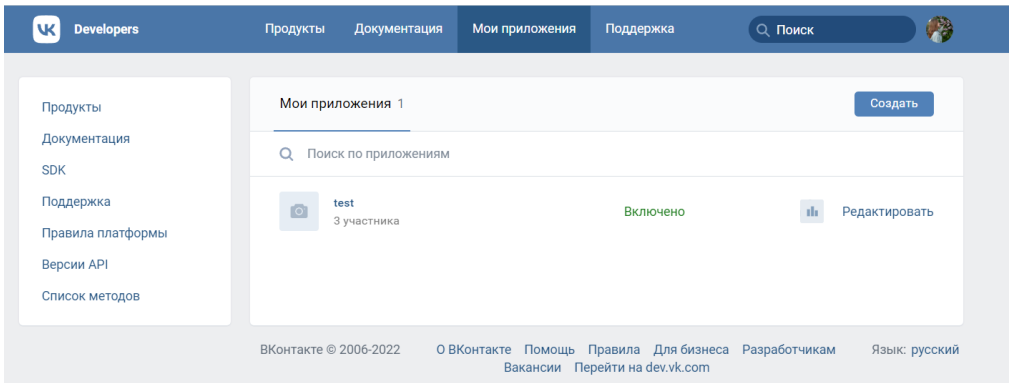

Рисунок 19 – Вкладка Мои приложения на сайте

Каждое зарегистрированное приложение содержит уникальный идентификатор (ID приложения), по которому сервер ВКонтакте определяет, кто именно обращается к нему с запросом (запросы без такого идентификатора игнорируются).

Создадим новое приложение **test** с типом **Сайт** – сервис ВКонтакте будет принимать запросы от такого приложения, только если они идут с сервера, указанного при создании этого приложения (рис. 20).

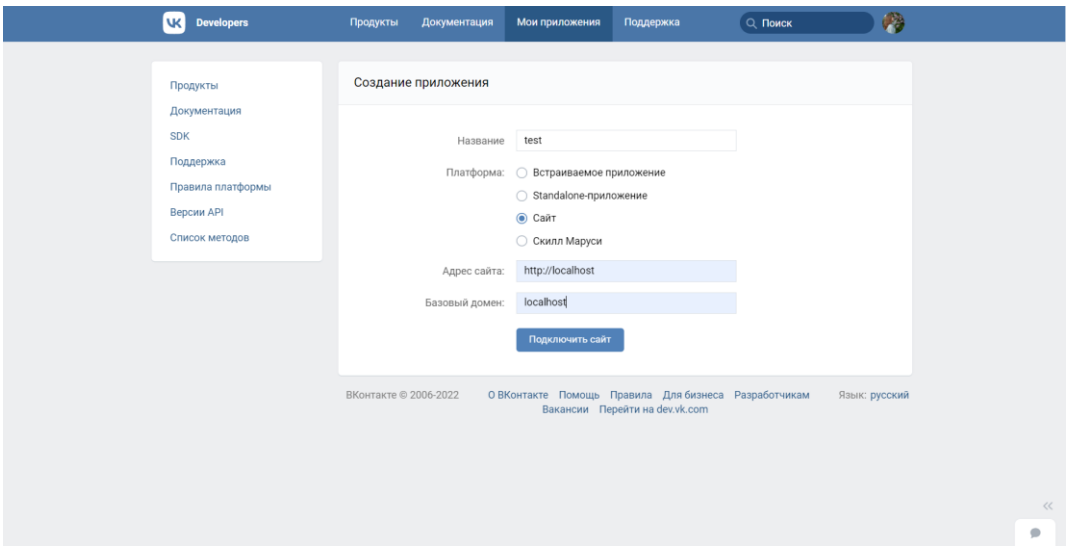

Рисунок 20 – Пример создания приложения

Идентификатор созданного приложения, который в дальнейшем необходимо указывать в скриптах, можно посмотреть в разделе **Настройки** (рис. 21).

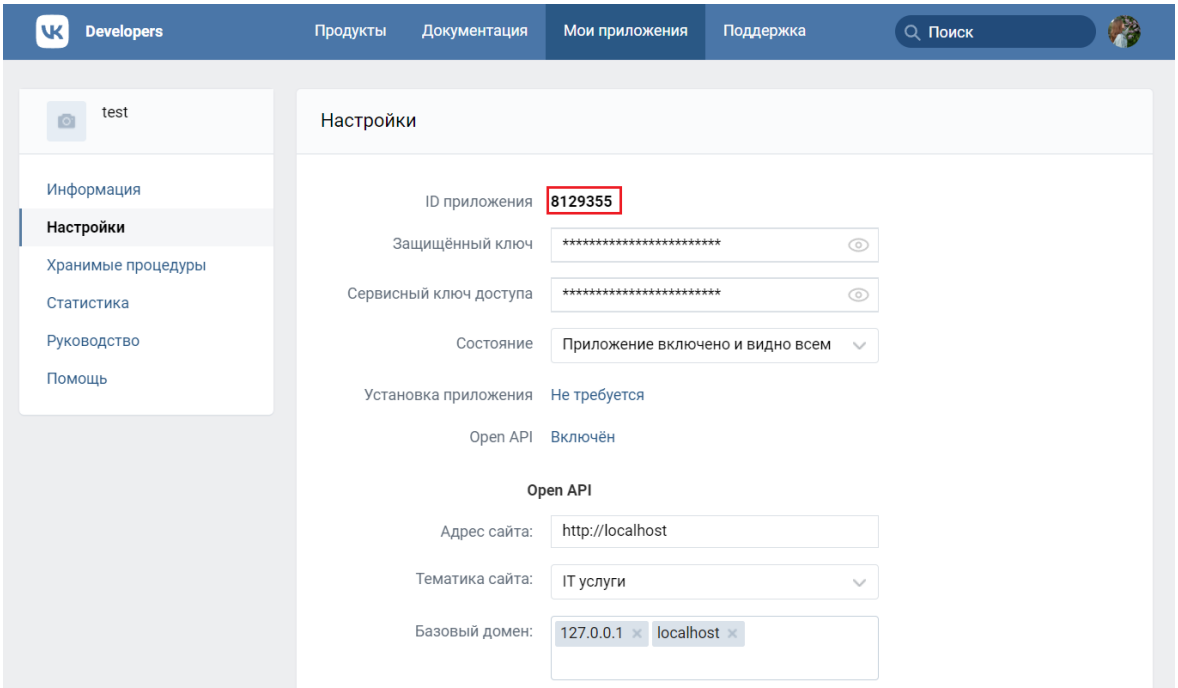

Рисунок 21 – ID приложения

Полный листинг данного веб-приложения необходимо посмотреть в приложении А Простая реализация.

Проанализируем код HTML-страницы index.html, в которой с помощью сценария JavaScript будет отображаться список наших друзей из социальной сети ВКонтакте.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
   <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.7.6/
handlebars.js">
   </script>
   <script src="https://vk.com/js/api/openapi.js?168"
type="text/javascript"></script>
<style>
body {
        font: 16px Helvetica;
       }

       .friends {
           display: flex;
           flex-wrap: wrap;
       }

       .friend {
           width: 100px;
           margin: 0 40px 40px 0;
           text-align: center;
       }

       .friend img {
           border-radius: 50%;
       }
   </style>
</head>
<body>
   <div class="container">
       <h1 id="headerInfo"></h1>
       <div id="results"></div>
   </div>

   <script id="user-template" type="text/x-handlebars-template">
       <div class="friends">
           {{#each items}}
           <div class="friend">
               <img src="{{photo_100}}"/>
               <div>{{first_name}} {{last_name}}</div>
           </div>
           {{/each}}
       </div>
   </script>
   <script src="./script.js"></script>
</body>
</html>
```

Для формирования HTML-кода списка друзей на основании данных, запрашиваемых через API ВКонтакте, используется шаблонизатор Handlebars. Библиотека Handlebars подключается на странице в секции `<head>` следующим образом:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.7.6/
handlebars.js">
</script>
```

Шаблонизатор Handlebars позволяет подставлять разные данные (но которые имеют одну и ту же структуру) в один заранее подготовленный шаблон, генерируя в результате строку, соответствующую этим данным.

Для того, чтобы со страницы можно было отправлять запросы на сервер ВКонтакте, нужно подключить специальный скрипт:

```
<script src=https://vk.com/js/api/openapi.js?168
type="text/javascript"></script>
```

Для разметки страницы используем общий контейнер `<div class="container">`, содержащий заголовок `<h1 id="headerInfo">` и контейнер для самого списка `<div id="result">`:

```
<div class="container">
    <h1 id="headerInfo">></h1>
    <div id="result">></div>
</div>
```

Также в HTML-файле есть тег <script> с нестандартным типом:

```
<script id="user-template" type="text/x-handlebars-template">
   . . .
</script>
```

Если в теге `<script>` указан неизвестный браузеру тип, то содержимое этого тега рассматривается как текст, а не как исполняемый скрипт. Поэтому в данном теге мы сохранили шаблон Handlebars, который понадобится при отображении загруженных с сервера данных.

В самом конце страницы из файла script.js подключается основной сценарий, из которого используются нужные методы API и формируются DOM-элементы для отображения списка друзей.

Файла script.js будет содержать следующий код:

```
VK.init({
```

```
   apiId: 8129355
});

function auth() {
   return new Promise((resolve, reject) => {
       VK.Auth.login(data => {
           if (data.session) {
               resolve();
           } else {
               reject(new Error('Не удалось авторизоваться'));
           }
       }, 2);
   });
}

function callAPI(method, params) {
    params.v = '5.81';

    return new Promise((resolve, reject) => {
        VK.api(method, params, (data) => {
            if (data.error) {
                reject(data.error);
            } else {
                resolve(data.response);
            }
        });
    })
 }

 auth()
 .then(() => {
     return callAPI('users.get', {name_case: 'gen'});
 })
 .then(([me]) => {
     const headerInfo = document.querySelector('#headerInfo');
     headerInfo.textContent = `Друзья на странице
${me.first_name} ${me.last_name}`;

     return callAPI('friends.get', {fields: 'city, country,
photo_100'});
 })
 .then(friends => {
     const template = document.querySelector('#user-
template').textContent;
     const render = Handlebars.compile(template);
     const html = render(friends);
     const results = document.querySelector('#results');

     results.innerHTML = html;
 });
```

В самом начале script.js происходит инициализация приложения ВКонтакте – без этого не будет возможности посылать запросы на сервер и

получать от него данные. Для этого нужно вызвать метод `init()` объекта VK, который становится доступным в глобальной области видимости после подключения на странице скрипта https://vk.com/js/api/openapi.js. В качестве параметра в метод `init()` передаётся объект с настройками. Для данного приложения достаточно указать свойство `appId` со значением, равным идентификатору приложения ВКонтакте:

```
VK.init({
   apiId: 8129355
});
```

Далее необходима авторизация на сервере ВКонтакте, после этого все запросы, которые страница будет посылать на сервер, будут идти от нашего имени (запросы от неавторизованных пользователей сервер отвергает).

При попытке авторизации в браузере появится всплывающее окно, в котором нужно подтвердить доступ зарегистрированного приложения ВКонтакте к учетной записи в этой социальной сети (рис. 22).

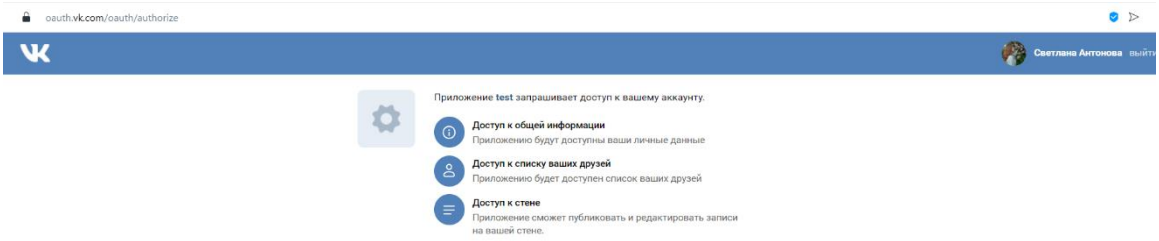

Рисунок 22 – Запрос доступа к аккаунту

Авторизация вызывается при помощи метода `VK.Auth.login()`, в который в качестве первого параметра передаётся коллбек-функция. Эта функция будет запущена после того, как закроется окно авторизации с запретом или разрешением доступа приложению ВКонтакте к аккаунту. Если доступ был разрешён, то в коллбек-функцию поступит объект, у которого есть поле `session`. Вторым параметром в метод `VK.Auth.login()` передается число, определяющее запрашиваемые права доступа для приложения (2 – это доступ к списку друзей).

Метод `VK.Auth.login()` является асинхронным, поэтому необходимо обернуть его в промис:

```
function auth() {
   return new Promise((resolve, reject) => {
       VK.Auth.login(data => {
           if (data.session) {
               resolve();
           } else {
               reject(new Error('Не удалось авторизоваться'));
           }
       }, 2);
   });
}
```

Для отправки запросов на сервер ВКонтакте необходима функция `callAPI()`, которая будет принимать два параметра: вызываемый метод и параметры, передаваемые в этот метод. Данная функция возвращает промис, в котором вызывается метод `VK.api()`, с помощью которого запрос уходит на сервер. Метод `VK.api()` – это асинхронная функция с тремя параметрами: имя нужного метода API; объект с параметрами для выбранного метода; коллбек-функция, которая вызывается при получении ответа на запрос от сервера ВКонтакте:

```
function callAPI(method, params) {
    params.v = '5.81';

    return new Promise((resolve, reject) => {
        VK.api(method, params, (data) => {
            if (data.error) {
                reject(data.error);
            } else {
                resolve(data.response);
            }
        });
    })
 }
```

Содержимое ответа от сервера получим из параметра `data`. Если запрос выполнен успешно, то поле `data.error` будет пустым и будет разрешен промис с помощью функции `resolve()`, в которую необходим передать полученные от сервера данные (поле `data.response`). Если же при выполнении запроса произошла ошибка (поле `data.error` содержит данные), то информация о ней будет передана в функцию `reject()`.

Перед вызовом `VK.api()` фиксируется версия 5.81 ответа, который ожидается ожидаем от сервера (`params.v = '5.81'`). Это позволяет избежать проблем при возможном изменении разработчиками API формата ответов от сервера.

После успешной авторизации приложения в обработчике `then()` вызывается метод `users.get`, чтобы получить имя авторизованного пользователя. Полученное имя будет в дальнейшем вставлено в DOM-дерево страницы в качестве текстового значения элемента `<h1 id="headerInfo">`, причем в заголовке имя должно стоять в родительном падеже. Сервер ВКонтакте может сразу вернуть имя пользователя в нужном родительном падеже, для этого при вызове метода `users.get` нужно указать параметр `{name_case: 'gen'}` (если этот параметр не указан, то ответ поступит в именительном падеже).

Далее добавлен второй обработчик `then()`, который сработает при успешном получении от сервера имени пользователя. В нём необходим первый элемент из массива полученных от сервера данных – получить его можно с помощью операции деструктуризации. Данные из полученного таким образом объекта `me` добавим в текстовое содержимое элемента `<h1 id="headerInfo">`.

Для формирования списка друзей необходимо запросить у сервера ВКонтакте данные о всех друзьях авторизованного пользователя. Для этого нужно вызвать метод `friends.get()`, указав в качестве параметра объект с полем `fields`, в значении которого будут перечислены все интересующие атрибуты друзей (город, страна, ссылка на фотографию).

Сервер вернёт данные о друзьях, которые будут доступны в третьем обработчике `then()`.

После получения от сервера данных о друзьях сформируется HTML-код для отображения списка друзей с помощью шаблонизатора Handlebars. Шаблон был заранее помещен на HTML-страницу:

```
<script id="user-template" type="text/x-handlebars-template">
```

```
    <div class="friends">
        {{#each items}}
        <div class="friend">
            <img src="{{photo_100}}"/>
            <div>{{first_name}} {{last_name}}</div>
        </div>
        {{/each}}
    </div>
</script>
```

В данном шаблоне для каждого объекта-элемента массива `items` будет сгенерирована нужная нам разметка, в которую будут подставлены значения полей `first_name, last_name` и `photo_100` текущего элемента.

Для того, чтобы воспользоваться данным шаблоном, сохраним его содержимое в переменную `template` и компилируем шаблон в переменную `render`:

```
auth()
   .then(() => {
       return callAPI('users.get', {name_case: 'gen'});
   })
   .then(([me]) => {
       const headerInfo = document.querySelector('#headerInfo');
       headerInfo.textContent = `Друзья на странице ${me.first_name} ${me.last_name}`;

       return callAPI('friends.get', {fields: 'city, country, photo_100'});
   })
   .then(friends => {
       const template = document.querySelector('#user-template').textContent;
       const render = Handlebars.compile(template);
       const html = render(friends);
       const results = document.querySelector('#results');

       results.innerHTML = html;
   });
```

Результат компиляции шаблона Handlebars – это функция, в которую при вызове передаются нужные данные. Передадим в функцию `render()` объект `friends` и запишем полученный результат в переменную `html`. Теперь осталось добавить полученный HTML-код в элемент `<div id="results">`.

В результате на странице отобразятся имена и фамилии друзей, а также их фотографии (рис. 23).

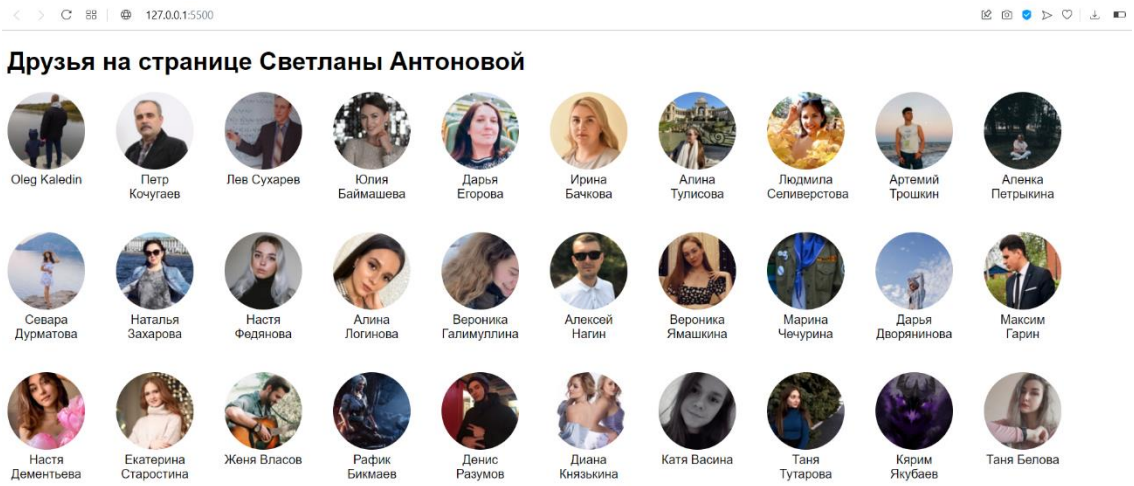

Рисунок 23 – Полученный результат

3.2 Переход к модели MVC

Вариант реализации, описанный ранее, имеет недостаток – при большом объеме кода поддерживать данное приложение станет неудобно. Поэтому код нужно разделить на независимые модули, которые будет проще тестировать, поддерживать и повторно применять.

Реализуем это с помощью паттерна MVC, помогающего решить задачу разделения логики приложения, получения и представления данных.

Model-View-Controller – схема разделения данных приложения и управляющей логики на три отдельных компонента: модель, контроллер и представление. В результате этого, модификация каждого компонента может осуществляться независимо.

В этом паттерне `Model` означает данные, `View` – визуальное представление этих данных, а `Controller` – посредник между `View` и `Model`, обеспечивающий их взаимодействие.

Добавим в разработанное ранее приложение дополнительную функциональность:

1. Вывод друзей пользователя, которые в данный момент находятся в ВКонтакте.
2. Показ на карте местоположения друзей пользователя.
3. Вывод списка новостей для текущего пользователя.

Для реализации 1-2 пунктов необходимо использовать ранее упомянутый метод `friends.get()`.

Выполнить 3 пункт можно с помощью методов для работы с новостной лентой пользователя (рис. 24).

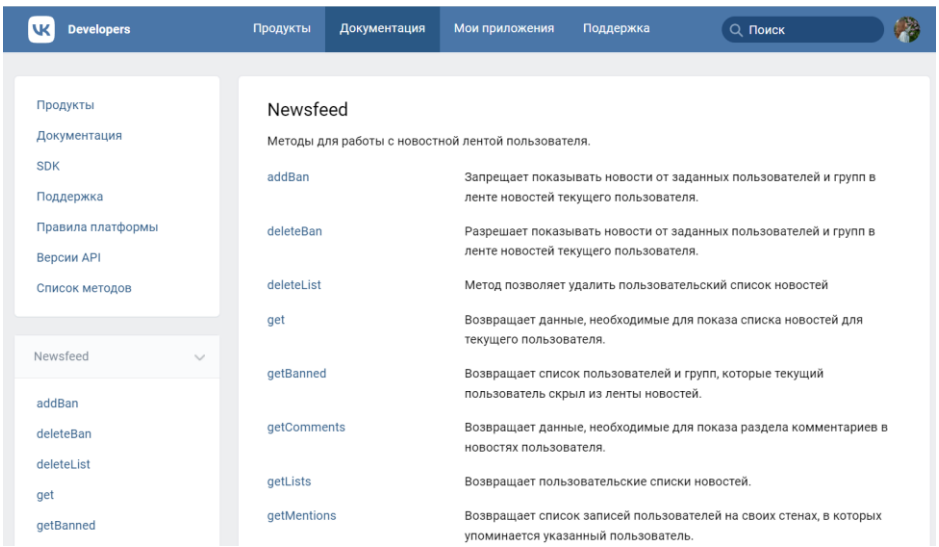

Рисунок 24 – Примеры методов для работы с новостной лентой пользователя

Получить список новостей для текущего пользователя можно с помощью метода `newsfeed.get()` (рис. 25).

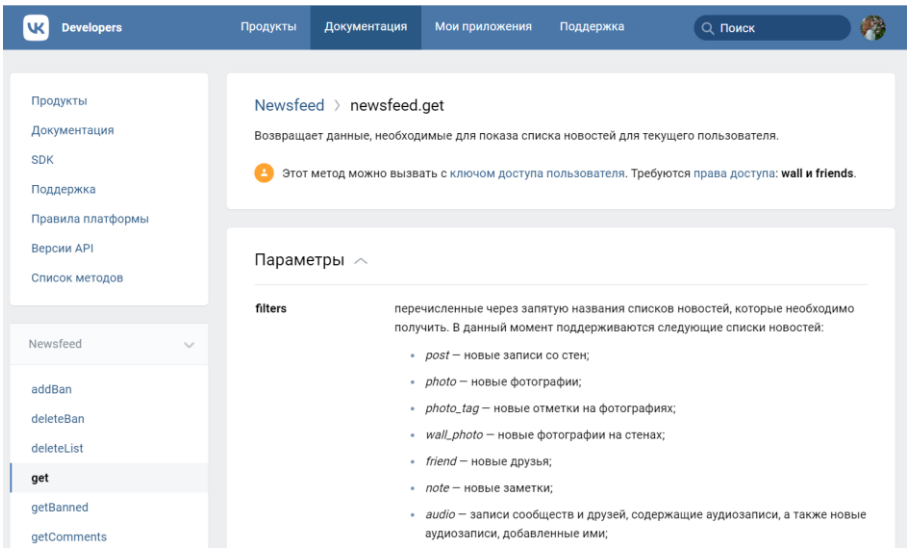

Рисунок 25 – Метод newsfeed.get()

Параметр `filters` позволяет выделить нужный список новостей (например, новые записи со стен, новые фотографии, новые заметки и т. п.) (рис. 26).

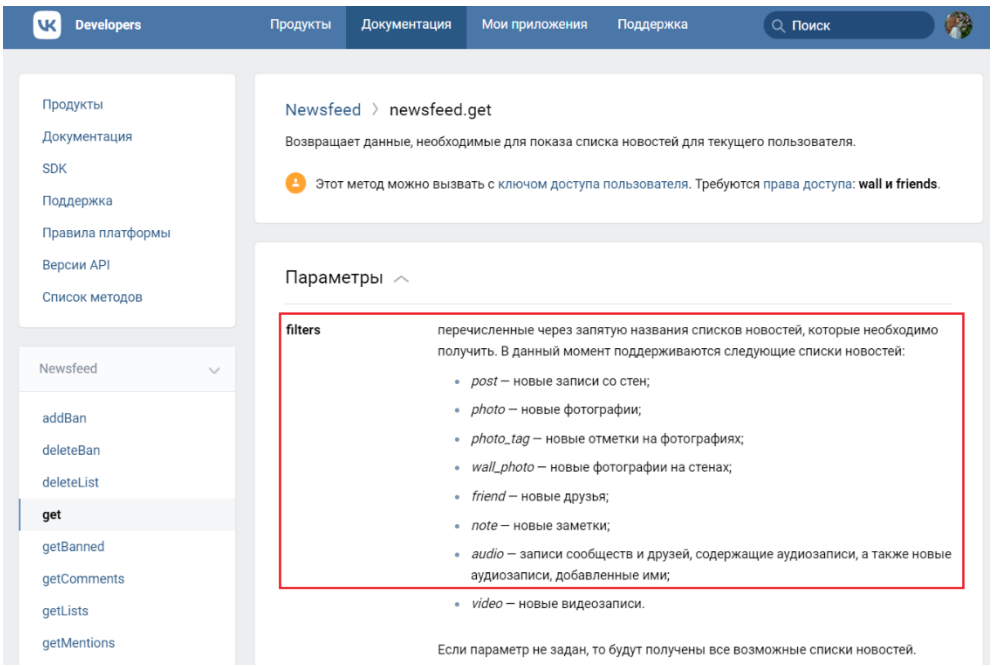

Рисунок 26 – Список дополнительных полей для параметра filters

При открытии страницы приложения в браузере отображается заголовок с именем пользователя и четыре кнопки (рис. 27).

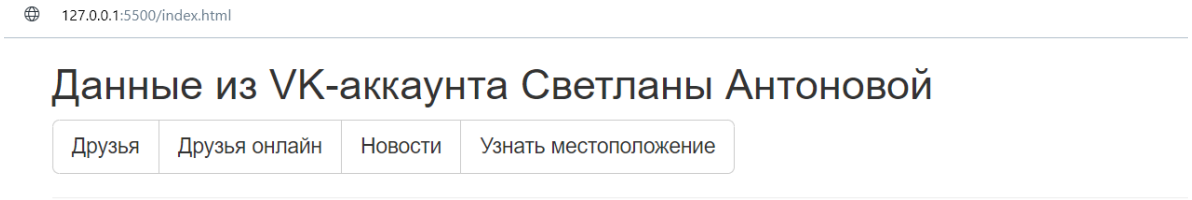

Рисунок 27 – Начальная страница приложения в браузере

Нажав на кнопку **Друзья**, получим список своих друзей в социальной сети ВКонтакте (рис. 28).

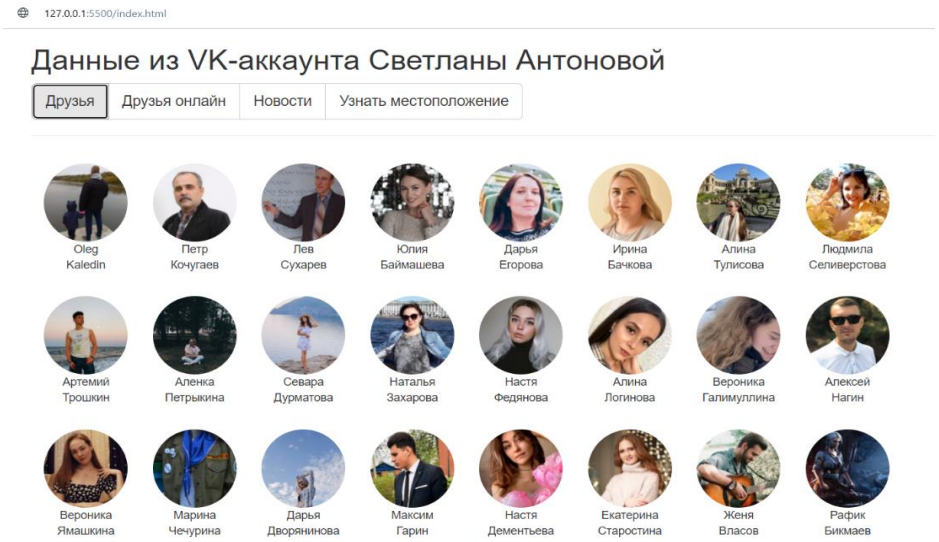

Рисунок 28 – Результат нажатия кнопки Друзья

Кнопка **Друзья онлайн** позволяет увидеть друзей, находящихся в данный момент в соцсети (рис. 29):

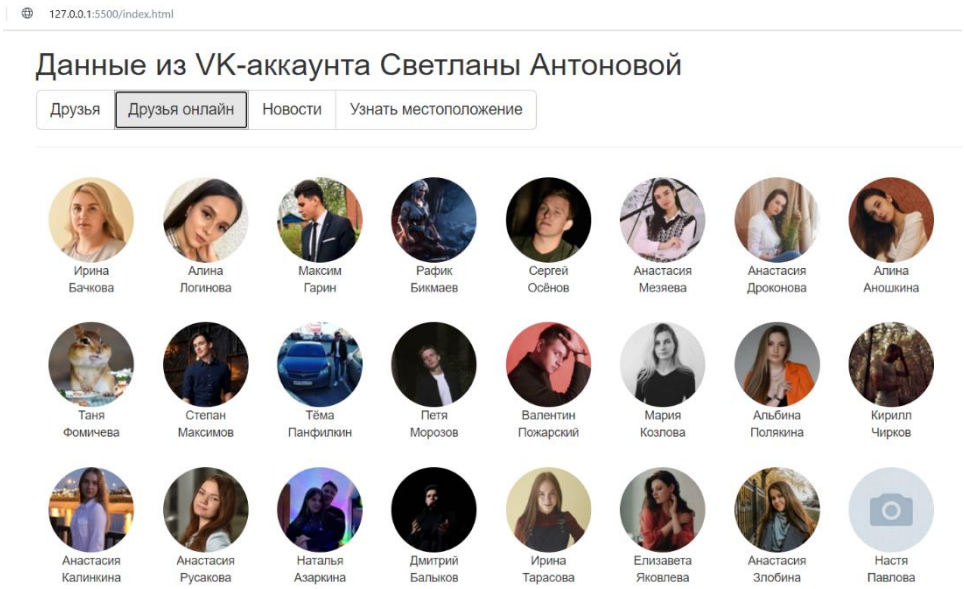

Рисунок 29 – Результат нажатия кнопки Друзья онлайн

При нажатии на кнопку **Новости** получим список новостей с профиля текущего пользователя (рис. 30):

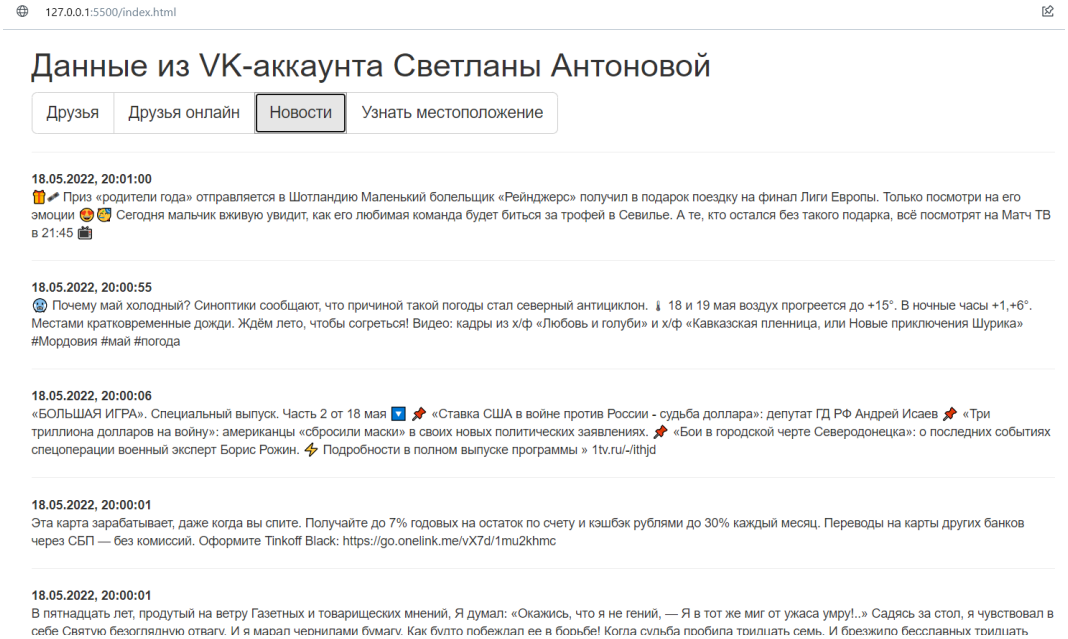

Рисунок 30 – Результат нажатия кнопки Новости

После нажатия кнопки **Узнать местоположение** появляется карта, на которой показано местоположение друзей (рис. 31):

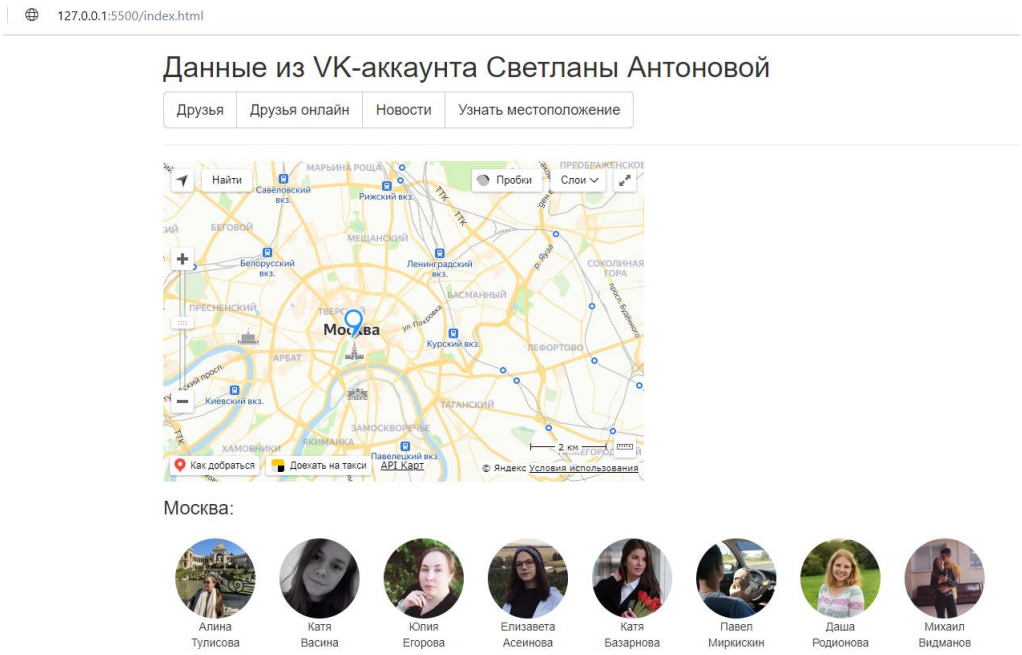

Рисунок 31 – Результат нажатия кнопки Узнать местоположение

Полный листинг данного веб-приложения необходимо посмотреть в приложении Б Применение модели MVC в приложении.

Рассмотрим HTML-разметку этого приложения (файл index.html). Для использования API Яндекс.Карт необходимо, чтобы компоненты API были загружены вместе с кодом страницы, как обычный внешний JavaScript-файл. Для этого подключим в прежний файл в самом начале страницы в секции `<head>` скрипт:

```
<script src="https://api-maps.yandex.ru/2.1/?apikey=72990671-fb8f-4843-a5a0-2f8820e2bdaf&lang=ru_RU" type="text/javascript"></script>
```

Всю разметку страницы поместим в контейнер `<div class="container">`. В нём будет находиться контейнер для заголовка `<div id="header">` и четыре кнопки `<button>` для вывода в контейнер `<div id="results">` данных о всех друзьях пользователя, друзьях онлайн, расположении друзей на карте и новостях пользователя:

```
<div class="container">
    <div id="header"></div>

    <div class="btn-group btn-group-lg">
        <button type="button" class="btn btn-default" onclick="Router.handle('friends')">Друзья</button>
```

```
        <button     type="button"     class="btn     btn-default"
onclick="Router.handle('friendsOnline')">Друзья онлайн</button>
        <button     type="button"     class="btn     btn-default"
onclick="Router.handle('news')">Новости</button>
        <button     type="button"     class="btn     btn-default"
onclick="Router.handle('map')">Узнать местоположение</button>
    </div>
    <hr>
    <div id="map" style="width: 600px; height: 400px;"></div>
    <div id="results"></div>
</div>
```

Далее на странице подключаются несколько скриптов, в которых реализована вся логика MVC-приложения:

```
<script src="model.js"></script>
<script src="view.js"></script>
<script src="controller.js"></script>
<script src="router.js"></script>
<script src="entry.js"></script>
```

В конце страницы разместим три шаблона Handlebars, оформленных в виде тегов `<script>` с нестандартным значением атрибут `type`. Эти шаблоны с идентификаторами `"headerTemplate"`, `"friendsTemplate"` и `"newsTemplate"` будут использоваться для визуализации данных, получаемых от сервера (заголовок с именем пользователя, список всех друзей и список новостей соответственно):

```
<script type="text/x-handlebars-template" id="headerTemplate">
    <h1>Данные из VK-аккаунта {{first_name}} {{last_name}}</h1>
</script>

<script type="text/x-handlebars-template" id="friendsTemplate">
    <div id="friendList">
        {{#each list}}
            <div class="friend text-center">
                <img src="{{photo_100}}" class="img-circle">
                <div>{{first_name}}<br>{{last_name}}</div>
            </div>
        {{/each}}
    </div>
</script>

<script type="text/x-handlebars-template" id="newsTemplate">
    <div class="news">
        {{#each list}}
            {{#if text}}
                <div class="post">
                    <b>{{formatDate date}}</b>
                    <div class="post-text">{{{text}}}</div>
```

```
                </div>
                <hr>
            {{/if}}
        {{/each}}
    </div>
</script>

<script                          type="text/x-handlebars-template"
id="friendsOnlineTemplate">
    <div id="friendsOnlineList">
        {{#each list}}
            {{#if online}}
                <div class="friends text-center" style="display:
inline-block; margin: 15px;">
                    <img src="{{photo_100}}" class="img-circle">
                    <div>{{first_name}}<br>{{last_name}}</div>
                </div>
            {{/if}}
        {{/each}}
    </div>
</script>
```

Перейдём теперь к рассмотрению JavaScript-кода.

В скрипте model.js определим получение данных от сервера. Здесь будет существовать глобальная переменная, объект `Model` с несколькими методами:

- `login()` – регистрация пользователя на сервере ВКонтакте;
- `callAPI()` – вызов API-функции;
- `getUser()` – получение данных о зарегистрированном пользователе;
- `getFriends()` – получение списка друзей пользователя;
- `getNews()` – получение списка новостей со страницы пользователя.

Три последних метода вызывают универсальный метод `callApi()`, передавая в него параметры для вызова нужных API-функций `'users.get'`, `'friends.get'` и `'newsfeed.get'`. Объект `Model` занимается только получением данных от сервера и ничего не знает о том, как в дальнейшем будут использоваться эти данные, как они будут визуализироваться.

```
window.Model = {
    login(appId, perms) {
        return new Promise((resolve, reject) => {
```

```
            VK.init({
                apiId: appId
            });

            VK.Auth.login(response => {
                if (response.session) {
                    resolve(response);
                } else {
                    reject(new Error('Не удалось
авторизоваться'));
                }
            }, perms);
        });
    },
    callApi(method, params) {
        params.v = params.v || '5.81';

        return new Promise((resolve, reject) => {
            VK.api(method, params, response => {
                if (response.error) {
                    reject(new Error(response.error.error_msg));
                } else {
                    resolve(response.response);
                }
            });
        });
    },
    getUser(params = {}) {
        return this.callApi('users.get', params);
    },
    getFriends(params = {}) {
        return this.callApi('friends.get', params);
    },
    getNews(params = {}) {
        return this.callApi('newsfeed.get', params);
    },
};
```

Скрипт view.js используется для подготовки HTML-кода для визуализации данных на странице. В этом скрипте создаётся объект `View` (глобальная переменная) с единственным методом `render()`. В качестве аргументов этот метод принимает имя шаблона для отображения данных и сами эти данные:

```
window.View = {
    render(templateName,model) {
        templateName = templateName + 'Template';

        const                    templateElement                    =
document.getElementById(templateName);
        const templateSource = templateElement.innerHTML;
```

```
        const renderFn = Handlebars.compile(templateSource);

        return renderFn(model);
    }
};
```

Нужный шаблон извлекается из DOM-дерева с помощью свойства `innerHTML` элемента с идентификатором, содержащим имя шаблона. Затем этот шаблон с помощью метода `Handlebars.compile()` компилируется, то есть преобразуется в функцию `renderFn`. Полученная таким образом функция `renderFn()` вызывается, а в качестве аргумента ей передаются данных, хранящиеся в `model`.

Объект `View` ничего не знает о том, как в дальнейшем будут использоваться HTML-разметка, возвращаемая его методом `render()`. Другими словами, метод `render()` является универсальным – ему безразлично, откуда поступили данные, и как в дальнейшем будут работать с кодом для визуализации, который `render()` возвращает.

Как уже было описано ранее в MVC-приложении модель (объект `Model`) и представление (объект `View`) не знают о существовании друг друга. Для организации взаимодействия между ними нужен посредник – контроллер. Контроллер представлен объектом `Controller` (глобальная переменная), который создаётся в скрипте controller.js и содержит четыре метода: `friendsRoute()`, `newsRoute()`, `friendsOnlineRoute()` и `mapRoute()`.

```
let myMap;
window.Controller = {
    async friendsRoute() {
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const friends = await Model.getFriends({ fields:
'photo_100'});
        results.innerHTML = View.render('friends', {list:
friends.items});
    },
    async newsRoute() {
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const news = await Model.getNews({ filters: 'post' ,
count: 20 });
```

```
        results.innerHTML    =    View.render('news',    {list:
news.items});
    },
    async friendsOnlineRoute(){
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const   friends   =   await   Model.getFriends({   fields:
'photo_100,online'});
        results.innerHTML  =  View.render('friendsOnline',  {list:
friends.items});
    },
    async mapRoute(){
        document.querySelector('#results').innerHTML="";
        var cityBlock = document.querySelector('#city');
        cityBlock.innerHTML = "";
        document.querySelector('#map').style.display = 'block';
        const   friends   =   await   Model.getFriends({   fields:
'photo_100,city'});
        await ymaps.ready(function (){
            if(myMap    ===    undefined)    myMap    =    new
ymaps.Map('map',{ center: [55.76, 37.64], zoom: 12 });
            myMap.geoObjects.removeAll();
            let mapa = new Map();
            for(let i = 0; i < friends.items.length; i++){
                if(friends.items[i].city != undefined){
                    mapa[friends.items[i].city.title]          =
mapa[friends.items[i].city.title] || [];

mapa[friends.items[i].city.title].push(friends.items[i]);
                }
            }
            for(let city in mapa){
                ymaps.geocode(city).then(function(res){
                    var                 coord                 =
res.geoObjects.get(0).geometry.getCoordinates();
                    var        myPlacemark        =        new
ymaps.Placemark(coord,{hintContent : 'Клик для просмотра друзей'
});
                    myPlacemark.events.add('click',function(){
                        const             results             =
document.querySelector('#results');
                        cityBlock                             =
document.querySelector('#city');
                        cityBlock.innerHTML = city+":";
                        results.innerHTML                     =
View.render('friendsMap',{list: mapa[city]});
                    });
                    myMap.geoObjects.add(myPlacemark);
                });
            }
        });
    },
};
```

Задачи этих методов:

- Получить данные путем вызова нужного метода объекта `Model` (`getFriends()` для вывода списка всех друзей, друзей онлайн и показа друзей на карте; `getNews()` для вывода списка новостей).
- Получить HTML-разметку для отображения этих данных. Для этого используется метод `render` объекта `View`, в который передаются полученные данные.
- Поместить эту HTML-разметку внутрь элемента `<div id="results">`, ссылка на который сохраняется в переменной `results`.

Скрипт entry.js определим как входную точку приложения, он содержит код, который автоматически выполняется при загрузке страницы. В начале этого скрипта есть две вспомогательные настройки для корректного отображения даты и времени в шаблонах Handlebars.

Затем происходит регистрация приложения на сервере ВКонтакте. Для этого вызывается метод `login()` объекта `Model`, в качестве аргументов передаются идентификатор приложения и требуемые права доступа.

Если регистрация прошла успешно, то вызывается метод `Model.getUser()` для получения с сервера имени пользователя, запустившего приложение. HTML-код для отображения полученной информации о пользователе вставляется внутрь элемента `<div id="header">`.

Если при регистрации приложения или получении информации с сервера возникнет ошибка, то сообщение о ней будет выведено в диалоговое окно с помощью команды `alert()`.

```
Handlebars.registerHelper('formatTime', time => {
    let minutes = (time / 60).toFixed();
    let seconds = time - minutes * 60;

    minutes = minutes.toString().length === 1 ? '0' + minutes :
minutes;
```

```
    seconds = seconds.toString().length === 1 ? '0' + seconds :
seconds;

    return minutes + ':' + seconds;
});

Handlebars.registerHelper('formatDate', ts => {
    return new Date(ts * 1000).toLocaleString();
});
Model.login(8129355, 2 | 8192)
    .then(() => {
        return Model.getUser({ name_case: 'gen' }).then(([me])
=> {
            const header = document.querySelector('#header');
            header.innerHTML = View.render('header', me);
        });
    })
    .catch(e => {
        console.error(e);
        alert('Ошибка: ' + e.message);
    });
```

Файл router.js, как и entry.js, не относится непосредственно к паттерну MVC, это вспомогательные скрипты, необходимые для запуска приложения.

Скрипт router.js нужен для вызова методов контроллера. Метод `Router.handle()` указан в файле index.html в качестве обработчика события клика по кнопкам **Друзья**, **Друзья онлайн**, **Новости** и **Указать местоположение**. В качестве аргумента в этот метод передаётся строка `'friends'`, `'friendsOnline'`, `'news'` или `'map'` соответственно.

```
window.Router = {
    handle(route) {
        const routeName= route + 'Route';
        Controller[routeName]();
    }
};
```

В результате на странице отобразятся имена и фамилии друзей, а также их фотографии (рис. 32).

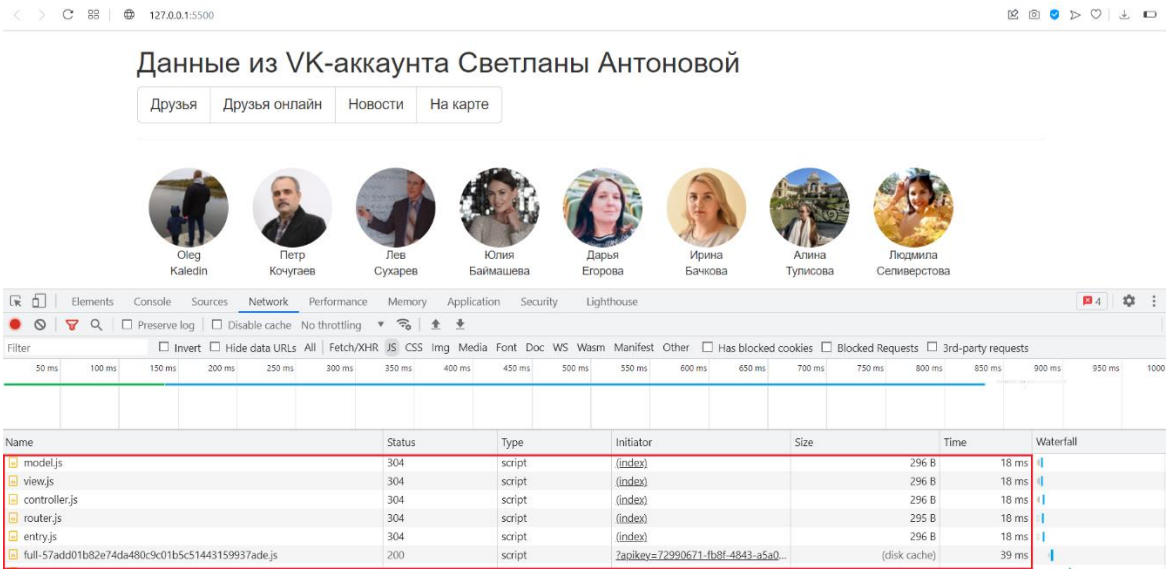

Рисунок 32 – Полученный результат

3.3 Переход к модулям

В приложении, реализованном в предыдущем разделе, использовались несколько скриптов, в которых создавались глобальные переменные (объекты):

```
// controller.js
window.Controller = {
    . . .
};

// model.js
window.Model = {
    . . .
};

// view.js
window.View = {
    . . .
};

// router.js
window.Router = {
    . . .
};
```

Все эти файлы по отдельности подключались в HTML-файле index.html. Хотя скрипты находятся в отдельных файлах, настоящей модульности здесь нет, так как после подключения весь код оказывается в глобальной области видимости.

Необходимо переписать проект с использованием модулей и функции `require()`,вынести в отдельные модули части кода, отвечающие за вывод всех друзей пользователя (showFriends.js), друзей онлайн (showFriendsOnline.js), показ друзей на карте (showMap.js) и вывод списка новостей (showNews.js),

Но перед подключением этих файлов на HTML-страницу необходимо объединить их в один скрипт. Когда сценариев немного, такую сборку можно произвести и вручную, однако реальные проекты могут состоять из десятков, сотен и даже тысяч файлов. В этом случае не обойтись без специальных сборщиков.

Для начала определим граф зависимостей модулей. Точкой входа у нас является скрипт entry.js, в котором используются переменные `Model` и `View`. Добавим в данный скрипт код для подключения с помощью `require()` модулей из model.js и view.js:

```
// entry.js
. . .
var Model = require('./model.js');
var View = require('./view.js');
```

Перейдем к файлу model.js. В нём не используются объекты, которые нужно было бы подключать в виде модулей. Поэтому достаточно будет превратить model.js в модуль, заменив переменную `window.Model` на свойство `module.exports`:

```
// model.js
module.exports = {
    login(appId, perms) {
        return new Promise((resolve, reject) => {
            VK.init({
                apiId: appId
            });
    . . .
};
```

Теперь при подключении модуля model.js в файле entry.js переменная `Model` будет указывать на объект, который мы сохранили в `module.exports` в файле model.js.

Перейдём к файлу view.js. Здесь также не нужны подключения других модулей, поэтому достаточно превратить view.js в модуль, изменив `window.View` на `module.exports`:

```
module.exports = {
    . . .
}
```

При подключении модуля model.js в файле entry.js переменная `View` будет указывать на объект, который мы сохранили в `module.exports` в файле view.js.

Перейдем теперь к HTML-файлу index.html и уберём в нем строки, в которых подключались наши скрипты. Внутри `<body>` останутся только те теги `<script>`, которые задают шаблоны Handlebars:

```
<body>
   <div class="container">
       <div id="header"></div>
       . . .
       <div id="results"></div>
   </div>
    . . .
</body>
```

Теперь необходимо изменить механизм вызова методов контроллера, которые до этого описывались непосредственно в HTML-разметке с помощью атрибута `onclick` в кнопках. Теперь вместо него укажем data-атрибуты, которые в дальнейшем будем обрабатывать в скриптах:

```
<button type="button" class="btn btn-default" data-
route="friends">Друзья</button>
        <button type="button" class="btn btn-default" data-
route="friendsOnline">Друзья онлайн</button>
        <button type="button" class="btn btn-default" data-
route="news">Новости</button>
        <button type="button" class="btn btn-default" data-
route="map">Указать местоположение</button>
```

Обработчик кликов по кнопкам заданы с помощью делегирования, выделяя нужные элементы по наличию data-атрибута `route`. Определение этого обработчика поместим в метод `then()`, который срабатывает после успешного получения данных о пользователе в скрипте entry.js:

```
// entry.js
. . .
```

```
Model.login(8129355, 2 | 8192)
.then(() => Model.getUser({ name_case: 'gen' }))
.then(([me]) => {
        const header = document.querySelector('#header');
        header.innerHTML = View.render('header', me);

        document.addEventListener('click', e => {
            const {route} = e.target.dataset;

            if (route) {
               Router.handle(route);
            }
        });
})
.catch(e => {
    console.error(e);
    alert('Ошибка: ' + e.message);
});
```

В entry.js происходит вызов метода `handle()` объекта `Router`, для этого необходимо подключить модуль из файла router.js:

```
// entry.js
. . .
var Model = require('./model.js');
var View = require('./view.js');
var Router = require('./router.js');
```

Далее превращаем router.js в модуль и подключаем в начале скрипта модуль controller.js, так как используется объект `Controller`:

```
// router.js
var Controller = require('./controller.js');

module.exports = {
   handle(route) {
       const routeName= route + 'Route';
       Controller[routeName]();
   }
};
```

Также превращаем файл controller.js в модуль, в начале скрипта подключаем модули showFriends.js, showNews.js, showFriendsOnline.js и showMap.js.

```
const showFriends = require('./showFriends.js');
const showNews = require('./showNews.js');
const showFriendsOnline = require('./showFriendsOnline.js');
const showMap = require('./showMap.js');

module.exports = {
```

```
    async friendsRoute() {
        return showFriends();
    },
    async newsRoute() {
        return showNews();
    },
    async friendsOnlineRoute(){
        return showFriendsOnline();
    },
    async mapRoute(){
        return showMap();
    },
};
```

Затем необходимо в модулях, отвечающих за вывод информации, подключить model.js и view.js, так как в них используются объекты `Model` и `View`.

```
// showOnlineFriends.js
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function(){
. . .
};

// showFriends.js
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function(){
. . .
};

// showMap.js
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function(){
. . .
};

// showNews.js
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function(){
. . .
};
```

Полный листинг данного веб-приложения необходимо посмотреть в приложении В Применение модулей в приложении.

Таким образом, мы получили следующее дерево зависимостей модулей:

```
// Entry
// - - - Model
// - - - View
// - - - Router
// - - - - Controller
// - - - - - - showFriends
// - - - - - - - - - Model
// - - - - - - - - - View
// - - - - - - showOnlineFriends
// - - - - - - - - - Model
// - - - - - - - - - View
// - - - - - - showMap
// - - - - - - - - - Model
// - - - - - - - - - View
// - - - - - - showNews
// - - - - - - - - - Model
// - - - - - - - - - View
```

3.4 Сборка проекта

Первым будем использовать сборщик Browserify. Напомним, что для работы с ним на компьютере заранее должны быть установлены Node.js и npm [10, 11, 12].

Установим Browserify на свой компьютер, выполнив в консоли команду: `npm install -g browserify`

Воспользуемся данным бандлером и соберём приложение, начиная с точки входа entry.js: `browserify ./entry.js -o build.js`

В результате будет создан единственный файл (бандл) build.js, который подключается на HTML-странице:

```
<body>
   <div class="container">
       . . .
   </div>

   <script src="build.js"></script>

   <script type="text/x-handlebars-template"
id="headerTemplate">
```

```
      <h1>Данные из VK-аккаунта {{first_name}}
{{last_name}}</h1>
   </script>
   . . .
</body>
```

После открытия приложения в браузере убедимся, что оно работает – при нажатии на кнопки выводится нужная информация. При этом на страницу загружается единственный файл build.js (рис. 33):

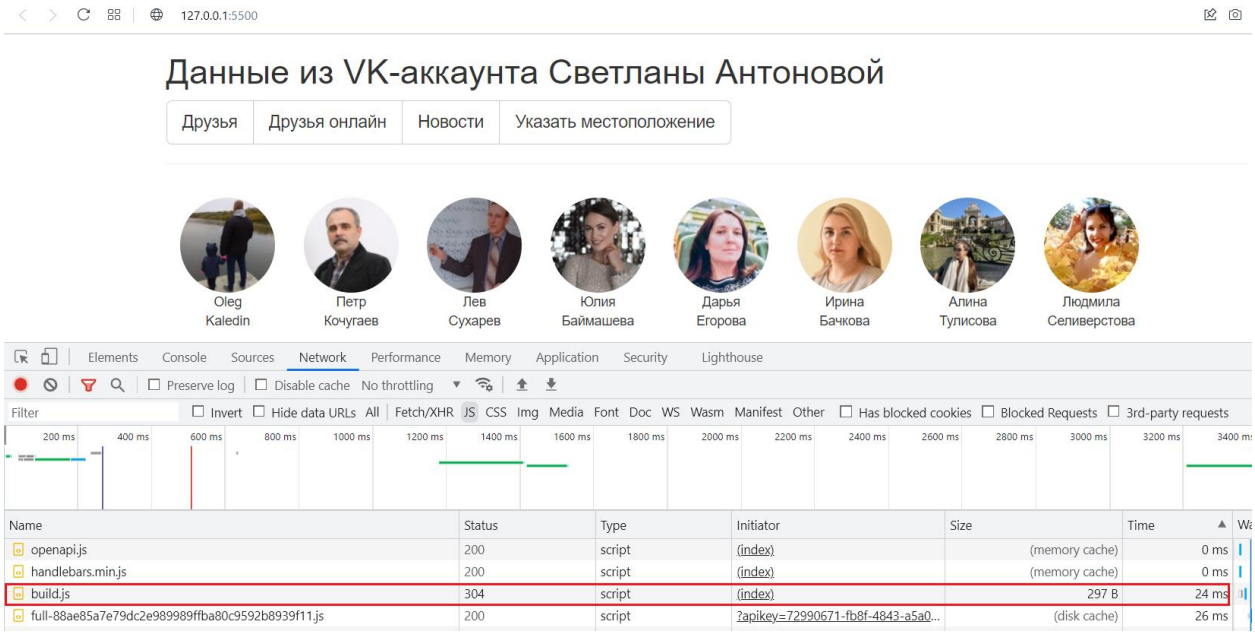

Рисунок 33 – Сборка приложения с помощью бандлера Browserify

Зайдя в папку, которая содержит файлы, необходимые для сборки можем наблюдать в ней скрипт build.js (рис. 34). Размер данного файла составляет 9 КБ. В то время, как суммарный объем файлов, которые бы были подключены без использования данного сборщика составляет 13 КБ. Исходя из вышесказанного, можно сделать вывод, что использование бандлера Browserify уменьшает размер подключаемого к index.html файла на 4 КБ.

Имя	Дата изменения	Тип	Размер
build.js	19.05.2022 12:54	Исходный файл JavaScript	9 КБ
controller.js	25.04.2022 7:04	Исходный файл JavaScript	1 КБ
entry.js	18.05.2022 23:23	Исходный файл JavaScript	2 КБ
index.html	19.05.2022 12:54	Opera Web Document	3 КБ
model.js	18.05.2022 23:21	Исходный файл JavaScript	2 КБ
router.js	18.05.2022 23:24	Исходный файл JavaScript	1 КБ
showFriends.js	25.04.2022 15:48	Исходный файл JavaScript	1 КБ
showFriendsOnline.js	25.04.2022 15:48	Исходный файл JavaScript	1 КБ
showMap.js	25.04.2022 15:49	Исходный файл JavaScript	3 КБ
showNews.js	25.04.2022 15:48	Исходный файл JavaScript	1 КБ
view.js	23.04.2022 20:11	Исходный файл JavaScript	1 КБ

Рисунок 34 – Папка, содержащая файлы для сборки и бандл build.js

Теперь выполним сборку приложения с помощью Webpack. Как и Browserify, данный сборщик построен на Node.js, для его работы требуется наличие npm [13, 14].

Для инициализации нашего приложения выполним команду: `npm init`

После создания файла package.json с настройками по умолчанию нужно в этом файле изменить точку входа с index.js на entry.js.

Для добавления Webpack к проекту выполняем следующую команду: `npm install webpack -D`

В результате:

- В файл package.json в качестве devDependency добавится иИмя «Webpack».
- К проекту добавится новый каталог node_modules.
- В каталоге node_modules появятся файлы модуля Webpack.

Добавим в package.json скрипт для запуска бандлера:

```
"scripts": {
  "build": "webpack",
  "start": "webpack --watch"
},
```

В итоге файл package.json будет иметь следующую структуру:

```
{
  "name": "swebpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1",
      "build": "webpack",
      "start": "webpack --watch"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^5.72.1"
  }
}
```

Создадим в корне проекта конфигурационный файл webpack.config.js и запишем в него следующий код:

```
const path = require('path');

module.exports = {
  entry: './src/entry.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  }
};
```

- «Entry» (входные данные). Путь к точке входа, в нашем случае это файл entry.js в каталоге src.
- «Output» (выходные данные). Путь к результирующему файлу и его имя, в нашем случае dist/bundle.js.

После запуска Webpack в терминале командой: `npm run build` в папке dist появится файл bundle.js, который необходимо подключить в index.html.

Откроем приложение в браузере и убедимся, что оно работает и загружается единственный файл bundle.js (рис. 35):

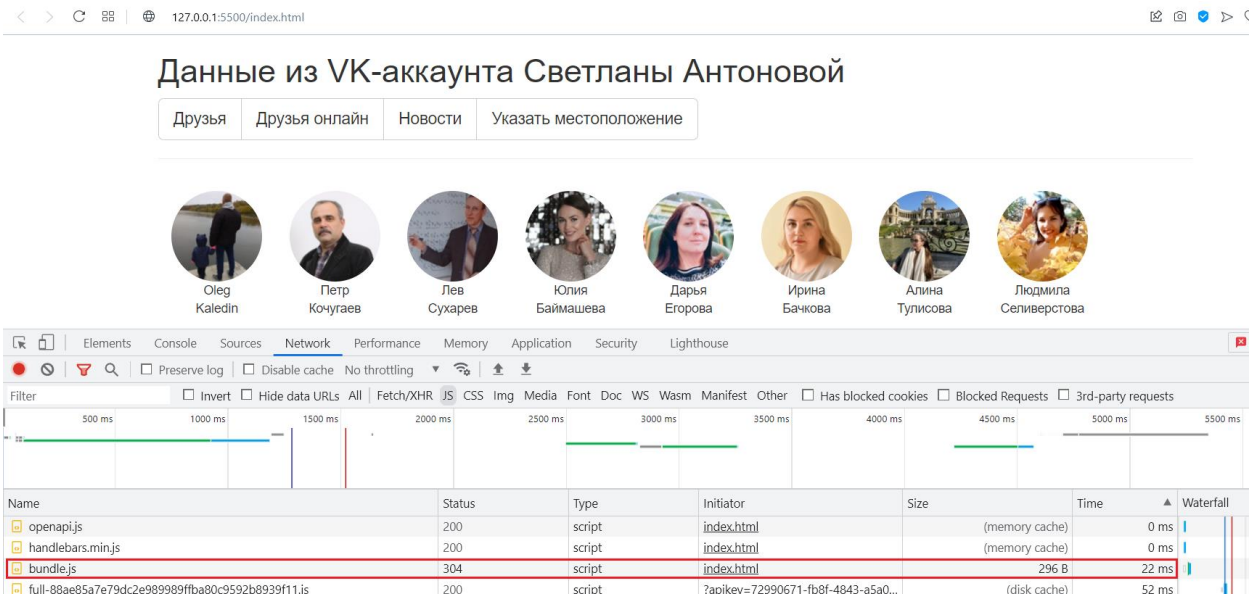

Рисунок 35 – Сборка приложения с помощью бандлера Webpack

В папке dist содержится скрипт bundle.js, который имеет размер равный 4 КБ (рис. 36). Напомним, что размер файлов, которые бы были подключены без применение сборщика составляет 13 КБ. Из этого следует, что использование бандлера Webpack значительно уменьшает размер подключаемого к index.html файла на 9 КБ.

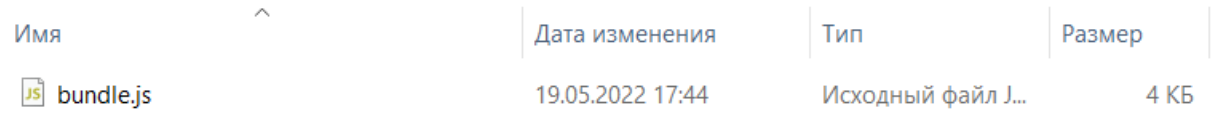

Рисунок 36 – Папка, содержащая бандл bundle.js

Теперь произведем сборку с помощью еще одного инструмента, который облегчает жизнь разработчикам. Parcel не требует конфигурации, необходимо только указать точку входа для приложения, и он самостоятельно объединит, преобразует и минимизирует все ресурсы. Эта утилита хорошо подходит для работы в условиях жёстких временных рамок и для быстрого создания прототипов приложений.

Для работы с Parcel необходимо установить пакет parcel-bundler. Установим его глобально с помощью npm: `npm i -g parcel-bundler`

Далее инициализируем проект с помощью npm: `npm init -y`

В результате будет создан файл package.json [7, 8]:

```
{
  "name": "parcel",
```

```
  "version": "1.0.0",
  "description": "",
  "main": "entry.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Осталось сообщить Parcel о том, какой файл является входной точкой проекта. Это позволит бандлеру узнать о том, что его серверу нужно отдавать клиентам. В приложение таким файлом будет entry.js. Для этого необходима команда: `parcel entry.js`.

В консоли, после успешного запуска сервера, будет выведено примерно следующее (рис. 37):

```
√  Built in 2.37s.
Server running at http://localhost:1234
```

Рисунок 37 – Вывод в консоли после успешного запуска

Parcel имеет встроенный сервер разработки, который будет автоматически выполнять пересборку приложения каждый раз, когда сохраняются изменения, внесённые в файлы проекта. Это возможно из-за того, что сервер поддерживает горячую замену модулей для увеличения скорости разработки.

Вернёмся в папку проекта. Здесь можно увидеть новые папки и файлы, созданные бандлером (рис. 38).

Рисунок 38 – Папки и файлы, созданные бандлером

В папке dist находим файл entry.js и подключаем его в index.html.

После открытия приложения удостоверимся, что все работает и на страницу подгружается только один файл (рис. 39):

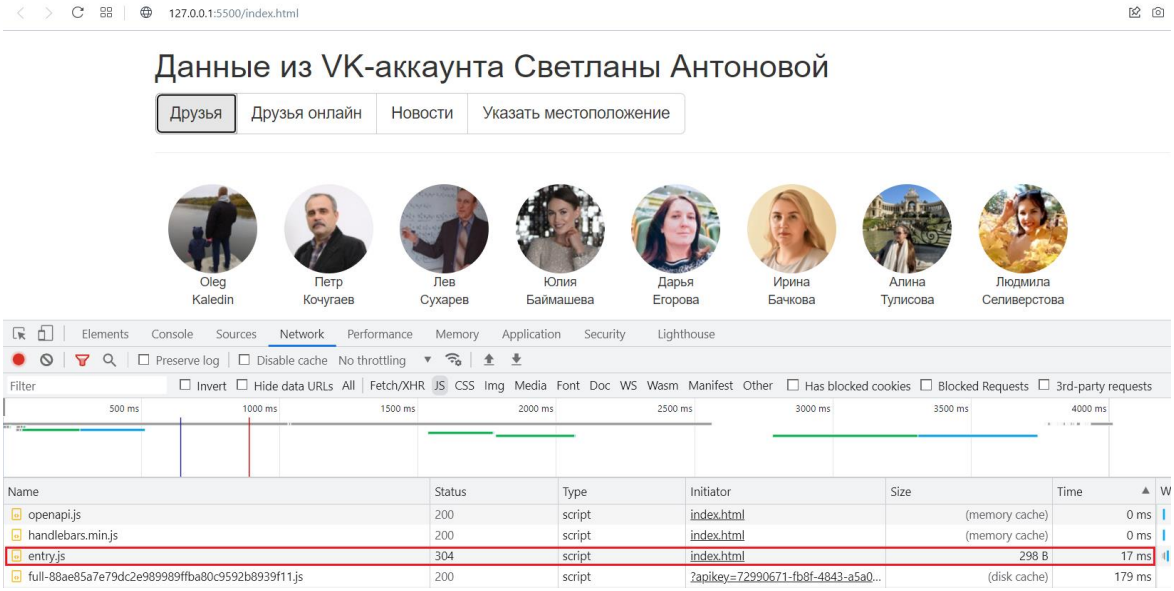

Рисунок 39 – Сборка приложения с помощью бандлера Parcel

Перейдя в папку с проектом, увидим в ней файл entry.js, который имеет размер 85 КБ (рис. 40). Именно он является файлом, который создал сборщик. А размер файлов, которые бы были подключены без применение сборщика составляет 13 КБ. Из этого следует, что использование бандлера Parcel увеличивает размер подключаемого к index.html файла на 72 КБ.

Имя	Дата изменения	Тип	Размер
entry.js	19.05.2022 22:00	Исходный файл Ja...	85 КБ

Рисунок 40 – Папка, содержащая бандл entry.js

Из все трех использованных выше сборщиков, самый минимальный размер получаемого файла получили у Webpack (4 КБ), далее Browserify (9 КБ), а затем Parcel (85 КБ).

ЗАКЛЮЧЕНИЕ

Несмотря на то, что новые языки программирования появляются каждый год, JavaScript остаётся одним из самых распространённых и стремительной развивающихся языков.

В потоке возникающих задач, решения, написанные на языке JavaScript, становятся труднее воспринимать, так как они становятся все сложнее и объемнее. Если решение содержится внутри одного HTML- или JavaScript-файла и имеет сотни и тысячи строк, то такой код физически невозможно поддерживать внутри него. Поэтому для повышения внутреннего качества кода и удобства сопровождения приложение делят на небольшие слабосвязанные фрагменты – модули. Модули упрощают работу с зависимостями, создание тестов и статический анализ кода. Принцип модульности относится не только к JavaScript-коду, но также и к CSS-стилям, HTML-шаблонам и к различным библиотекам.

Однако при использовании модулей даже небольшое приложение может содержать сотни файлов (внешних зависимостей). Вынуждать браузер пользователя делать сотни HTTP-запросов нецелесообразно, это сильно замедлит загрузку написанного проекта. Поэтому одна из самых важных задач различных сборщиков – это уменьшение количества файлов, склеивание их в бандл.

В данной работе было создано модульное приложение на языке JavaScript с использованием различных систем сборки и проанализированы результирующие бандлы.

Минимальный размер оказался у файла, собранного с помощью Webpack. Однако следует учитывать, что данный инструмент имеет самую сложную систему настройки.

Выбор инструментов для разработки – это серьезный вопрос. В любом случае выбор сборщика при разработке приложений остается за разработчиком. Выбор между Parcel, Browserify и Webpack зависит от того,

какие задачи ставятся и какие дополнительные инструменты используются. Если приложение и его зависимости тесно связаны с Node.js, то правильными вариантами будут Parcel и Browserify. Если необходимы возможности по управлению всеми статическими ресурсами, а не только скриптами, то тогда лучше подойдет Webpack.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Mozilla Developer Network: сайт / IT-ресурс / URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Methods (дата обращения: 14.03.2022) – Текст: электронный.

2. Симпсон Кайл Познакомьтесь, JavaScript. – Санкт-Петербург: Питер, 2022. – 192 с. – ISBN 978-5-4461-1875-5. Текст: непосредственный.

3. Хабр: сайт / IT-ресурс / – 2018 – URL: https://habr.com/ru/company/ruvds/blog/422893/ (дата обращения 18.05.2022) – Текст: электронный.

4. Кей Хорстман Современный JavaScript для нетерпеливых. – Москва: ДМК Пресс, 2021. – 288 с. – ISBN 978-5-97060-177-8. – Текст: непосредственный.

5. Региз Джон, Бибо Бэар, Марас Иосип Секреты JavaScript ниндзя. – Санкт-Петербург: ООО «Альфа книга», 2017. – 544 с. – ISBN 978-5-9908911-8-0. – Текст: непосредственный.

6. Хавербеке Марейн Выразительный JavaScript. Современное веб – программирование. – Санкт-Петербург: Питер, 2019. – 480 с. – ISBN 978-5-4461-1226-5. – Текст: непосредственный.

7. Хабр: сайт / IT-ресурс / – URL: https://habr.com/ru/company/ruvds/blog/473764/ (дата обращения 15.03.2022) – Текст: электронный.

8. Parcel: сайт / IT-ресурс / – URL: https://ru.parceljs.org/getting_started.html/ (дата обращения: 07.05.2022) – Текст: электронный.

9. Published in NOP::Nuances of Programming: сайт / IT-ресурс / URL: https://medium.com/nuances-of-programming/введение-в-webpack-для-новичков-6cafbf562386 (дата обращения 02.05.2022) – Текст: электронный.

10. Browserify: сайт / IT-ресурс / URL: https://browserify.org/ (дата обращения: 13.05.2022) – Текст: электронный.

11. Дэвид Хэррон Node.js. Разработка серверных приложений в JavaScript. – Москва: ДМК Пресс, 2012. – 143 с. – ISBN: 978-5-94074-809-0. – Текст: непосредственный.

12. Николай Прохоренко, Владимир Дронов JavaScript и Node.js для веб-разработчиков. – Санкт-Петербург: БХВ-Петербург, 2022. – 768 с. – ISBN 978-5-9775-6847-0. – Текст: непосредственный.

13. Webpack: сайт / IT-ресурс / URL: https://webpack.js.org/ (дата обращения: 20.05.2022) – Текст: электронный.

14. Хабр: сайт / IT-ресурс / – 2020 – URL: https://habr.com/ru/post/514838/ (дата обращения 20.04.2022) – Текст: электронный.

ПРИЛОЖЕНИЕ А Простая реализация

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
   <script

src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.7.6/
handlebars.js">
   </script>
   <script src="https://vk.com/js/api/openapi.js?168"
type="text/javascript"></script>
<style>
body {
           font: 16px Helvetica;
       }

       .friends {
           display: flex;
           flex-wrap: wrap;
       }

       .friend {
           width: 100px;
           margin: 0 40px 40px 0;
           text-align: center;
       }

       .friend img {
           border-radius: 50%;
       }
   </style>
</head>
<body>
   <div class="container">
       <h1 id="headerInfo"></h1>
       <div id="results"></div>
   </div>

   <script id="user-template" type="text/x-handlebars-template">
       <div class="friends">
           {{#each items}}
           <div class="friend">
               <img src="{{photo_100}}"/>
               <div>{{first_name}} {{last_name}}</div>
           </div>
           {{/each}}
       </div>
   </script>
```

```
   <script src="./script.js"></script>
</body>
</html>
```

script.js

```
VK.init({
   apiId: 8129355
});

function auth() {
   return new Promise((resolve, reject) => {
       VK.Auth.login(data => {
           if (data.session) {
               resolve();
           } else {
               reject(new Error('Не удалось авторизоваться'));
           }
       }, 2);
   });
}

function callAPI(method, params) {
    params.v = '5.81';

    return new Promise((resolve, reject) => {
        VK.api(method, params, (data) => {
            if (data.error) {
                reject(data.error);
            } else {
                resolve(data.response);
            }
        });
    })
 }

auth()
 .then(() => {
     return callAPI('users.get', {name_case: 'gen'});
 })
 .then(([me]) => {
     const headerInfo = document.querySelector('#headerInfo');
     headerInfo.textContent = `Друзья на странице
${me.first_name} ${me.last_name}`;

     return callAPI('friends.get', {fields: 'city, country,
photo_100'});
 })
 .then(friends => {
     const template = document.querySelector('#user-
template').textContent;
     const render = Handlebars.compile(template);
     const html = render(friends);
```

```
    const results = document.querySelector('#results');

    results.innerHTML = html;
});
```

ПРИЛОЖЕНИЕ Б Применение модели MVC в приложении

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootst
rap.min.css">
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.5/
handlebars.min.js"></script>
    <script src="https://vk.com/js/api/openapi.js"
type="text/javascript"></script>
    <script src="https://api-
maps.yandex.ru/2.1/?apikey=72990671-fb8f-4843-a5a0-
2f8820e2bdaf&lang=ru_RU" type="text/javascript"></script>
    <style>
        .friend {
            float: left;
            margin: 15px;
        }
    </style>
</head>
<body>
<div class="container">
    <div id="header"></div>

    <div class="btn-group btn-group-lg">
        <button type="button" class="btn btn-default"
onclick="Router.handle('friends')">Друзья</button>
        <button type="button" class="btn btn-default"
onclick="Router.handle('friendsOnline')">Друзья онлайн</button>
        <button type="button" class="btn btn-default"
onclick="Router.handle('news')">Новости</button>
        <button type="button" class="btn btn-default"
onclick="Router.handle('map')">На карте</button>
    </div>
    <hr>
    <div id="map" style="width: 600px; height: 400px;"></div>
    <div id="results"></div>
</div>

<script src="model.js"></script>
<script src="view.js"></script>
<script src="controller.js"></script>
<script src="router.js"></script>
<script src="entry.js"></script>

<script type="text/x-handlebars-template" id="headerTemplate">
```

```
    <h1>Данные из VK-аккаунта {{first_name}} {{last_name}}</h1>
</script>

<script type="text/x-handlebars-template" id="friendsTemplate">
    <div id="friendList">
        {{#each list}}
            <div class="friend text-center">
                <img src="{{photo_100}}" class="img-circle">
                <div>{{first_name}}<br>{{last_name}}</div>
            </div>
        {{/each}}
    </div>
</script>

<script type="text/x-handlebars-template" id="newsTemplate">
    <div class="news">
        {{#each list}}
            {{#if text}}
                <div class="post">
                    <b>{{formatDate date}}</b>
                    <div class="post-text">{{{text}}}</div>
                </div>
                <hr>
            {{/if}}
        {{/each}}
    </div>
</script>

<script type="text/x-handlebars-template"
id="friendsOnlineTemplate">
    <div id="friendsOnlineList">
        {{#each list}}
            {{#if online}}
                <div class="friends text-center" style="display:
inline-block; margin: 15px;">
                    <img src="{{photo_100}}" class="img-circle">
                    <div>{{first_name}}<br>{{last_name}}</div>
                </div>
            {{/if}}
        {{/each}}
    </div>
</script>

<script type="text/x-handlebars-template"
id="friendsMapTemplate">
    <div id="friendsMapList">
        {{#each list}}
            <div class="friendsMap text-center" style="display:
inline-block; margin: 15px;">
                <img src="{{photo_100}}" class="img-circle">
                <div>{{first_name}}<br>{{last_name}}</div>
            </div>
        {{/each}}
```

```
    </div>
</script>

</body>
</html>
```

controller.js

```
let myMap;
window.Controller = {
    async friendsRoute() {
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const friends = await Model.getFriends({ fields:
'photo_100'});
        results.innerHTML = View.render('friends', {list:
friends.items});
    },
    async newsRoute() {
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const news = await Model.getNews({ filters: 'post' ,
count: 20 });
        results.innerHTML = View.render('news', {list:
news.items});
    },
    async friendsOnlineRoute(){
        document.querySelector('#map').style.display = 'none';
        const results = document.querySelector('#results');
        const friends = await Model.getFriends({ fields:
'photo_100,online'});
        results.innerHTML = View.render('friendsOnline', {list:
friends.items});
    },
    async mapRoute(){
        document.querySelector('#results').innerHTML="";
        document.querySelector('#map').style.display = 'block';
        const friends = await Model.getFriends({ fields:
'photo_100,city'});
        await ymaps.ready(function (){
            if(myMap === undefined) myMap = new
ymaps.Map('map',{ center: [55.76, 37.64], zoom: 12 });
            myMap.geoObjects.removeAll();
            let mapa = new Map();
            for(let i = 0; i < friends.items.length; i++){
                if(friends.items[i].city != undefined){
                    mapa[friends.items[i].city.title] =
mapa[friends.items[i].city.title] || [];

mapa[friends.items[i].city.title].push(friends.items[i]);
                }
            }
            for(let city in mapa){
```

```
                ymaps.geocode(city).then(function(res){
                    var coord =
res.geoObjects.get(0).geometry.getCoordinates();
                    var myPlacemark = new
ymaps.Placemark(coord,{hintContent : 'Клик для просмотра друзей'
});
                    myPlacemark.events.add('click',function(){
                        const results =
document.querySelector('#results');
                        results.innerHTML =
View.render('friendsMap',{list: mapa[city]});
                    });
                    myMap.geoObjects.add(myPlacemark);
                });
            }
        });
    },
};
```

entry.js

```
Handlebars.registerHelper('formatTime', time => {
    let minutes = (time / 60).toFixed();
    let seconds = time - minutes * 60;

    minutes = minutes.toString().length === 1 ? '0' + minutes :
minutes;
    seconds = seconds.toString().length === 1 ? '0' + seconds :
seconds;

    return minutes + ':' + seconds;
});

Handlebars.registerHelper('formatDate', ts => {
    return new Date(ts * 1000).toLocaleString();
});
Model.login(8129355, 2 | 8192)
    .then(() => {
        return Model.getUser({ name_case: 'gen' }).then(([me])
=> {
            const header = document.querySelector('#header');
            header.innerHTML = View.render('header', me);
        });
    })
    .catch(e => {
        console.error(e);
        alert('Ошибка: ' + e.message);
    });
```

model.js

```
window.Model = {
    login(appId, perms) {
        return new Promise((resolve, reject) => {
```

```
            VK.init({
                apiId: appId
            });

            VK.Auth.login(response => {
                if (response.session) {
                    resolve(response);
                } else {
                    reject(new Error('Не удалось
авторизоваться'));
                }
            }, perms);
        });
    },
    callApi(method, params) {
        params.v = params.v || '5.81';

        return new Promise((resolve, reject) => {
            VK.api(method, params, response => {
                if (response.error) {
                    reject(new Error(response.error.error_msg));
                } else {
                    resolve(response.response);
                }
            });
        });
    },
    getUser(params = {}) {
        return this.callApi('users.get', params);
    },
    getFriends(params = {}) {
        return this.callApi('friends.get', params);
    },
    getNews(params = {}) {
        return this.callApi('newsfeed.get', params);
    },
};
```

router.js

```
window.Router = {
    handle(route) {
        const routeName= route + 'Route';
        Controller[routeName]();
    }
};
```

view.js

```
window.View = {
    render(templateName,model) { // имя шаблона,данные
        templateName = templateName + 'Template';
```

```
        const templateElement =
document.getElementById(templateName);
        const templateSource = templateElement.innerHTML;
        const renderFn = Handlebars.compile(templateSource);

        return renderFn(model);
    }
};
```

ПРИЛОЖЕНИЕ В Применение модулей в приложении

index.js

```
<body>
<div class="container">
    <div id="header"></div>
    <script src="файл.js"></script>
    <div class="btn-group btn-group-lg">
        <button type="button" class="btn btn-default" data-
route="friends">Друзья</button>
        <button type="button" class="btn btn-default" data-
route="friendsOnline">Друзья онлайн</button>
        <button type="button" class="btn btn-default" data-
route="news">Новости</button>
        <button type="button" class="btn btn-default" data-
route="map">Указать местоположение</button>
    </div>
    <hr>
    <div id="map" style="width: 600px; height: 400px;"></div>
    <h3 id="city" class="city-label text-left"></h3>
    <div id="results"></div>
</div>


<script type="text/x-handlebars-template" id="headerTemplate">
    <h1>Данные из VK-аккаунта {{first_name}} {{last_name}}</h1>
</script>

<script type="text/x-handlebars-template" id="friendsTemplate">
    <div id="friendList">
        {{#each list}}
            <div class="friend text-center">
                <img src="{{photo_100}}" class="img-circle">
                <div>{{first_name}}<br>{{last_name}}</div>
            </div>
        {{/each}}
    </div>
</script>

<script type="text/x-handlebars-template" id="newsTemplate">
    <div class="news">
        {{#each list}}
            {{#if text}}
                <div class="post">
                    <b>{{formatDate date}}</b>
                    <div class="post-text">{{{text}}}</div>
                </div>
                <hr>
            {{/if}}
        {{/each}}
    </div>
</script>
```

```
<script type="text/x-handlebars-template"
id="friendsOnlineTemplate">
    <div id="friendsOnlineList">
        {{#each list}}
            {{#if online}}
                <div class="friends text-center" style="display:
inline-block; margin: 15px;">
                    <img src="{{photo_100}}" class="img-circle">
                    <div>{{first_name}}<br>{{last_name}}</div>
                </div>
            {{/if}}
        {{/each}}
    </div>
</script>

</body>
</html>
```

controller.js

```
const showFriends = require('./showFriends.js');
const showNews = require('./showNews.js');
const showFriendsOnline = require('./showFriendsOnline.js');
const showMap = require('./showMap.js');

module.exports = {
    async friendsRoute() {
        return showFriends();
    },
    async newsRoute() {
        return showNews();
    },
    async friendsOnlineRoute(){
        return showFriendsOnline();
    },
    async mapRoute(){
        return showMap();
    },
};
```

entry.js

```
var Model = require('./model.js');
var View = require("./view.js");
var Router = require('./router.js');

Handlebars.registerHelper('formatTime', time => {
    let minutes = (time / 60).toFixed();
    let seconds = time - minutes * 60;

    minutes = minutes.toString().length === 1 ? '0' + minutes :
minutes;
```

```
    seconds = seconds.toString().length === 1 ? '0' + seconds :
seconds;

    return minutes + ':' + seconds;
});

Handlebars.registerHelper('formatDate', ts => {
    return new Date(ts * 1000).toLocaleString();
});
Model.login(8129355, 2 | 8192)
.then(() => Model.getUser({ name_case: 'gen' }))
.then(([me]) => {
        const header = document.querySelector('#header');
        header.innerHTML = View.render('header', me);

        document.addEventListener('click', e => {
            const {route} = e.target.dataset;

            if (route) {
               Router.handle(route);
            }
        });
})
.catch(e => {
    console.error(e);
    alert('Ошибка: ' + e.message);
});
```

model.js

```
module.exports = {
    login(appId, perms) {
        return new Promise((resolve, reject) => {
            VK.init({
                apiId: appId
            });

            VK.Auth.login(response => {
                if (response.session) {
                    resolve(response);
                } else {
                    reject(new Error('Не удалось
авторизоваться'));
                }
            }, perms);
        });
    },
    callApi(method, params) {
        params.v = params.v || '5.81';

        return new Promise((resolve, reject) => {
            VK.api(method, params, response => {
                if (response.error) {
```

```
                    reject(new Error(response.error.error_msg));
                } else {
                    resolve(response.response);
                }
            });
        });
    },
    getUser(params = {}) {
        return this.callApi('users.get', params);
    },
    getFriends(params = {}) {
        return this.callApi('friends.get', params);
    },
    getNews(params = {}) {
        return this.callApi('newsfeed.get', params);
    }
}
```

router.js

```
var Controller = require('./controller.js');

module.exports = {
    handle(route) {
        const routeName= route + 'Route';
        Controller[routeName]();
    }
}
```

view.js

```
module.exports = {
    render(templateName,model) { // имя шаблона,данные
        templateName = templateName + 'Template';

        const templateElement =
document.getElementById(templateName);
        const templateSource = templateElement.innerHTML;
        const renderFn = Handlebars.compile(templateSource);

        return renderFn(model);
    }
}
```

showFriends.js

```
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function() {
    document.querySelector('#city').style.display = 'none';
    document.querySelector('#map').style.display = 'none';
    const results = document.querySelector('#results');
    var friends = await Model.getFriends({ fields:
'photo_100'});
```

```
    results.innerHTML = View.render('friends', {list:
friends.items});
};
```

showOnlineFriends.js

```
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function(){
    document.querySelector('#city').style.display = 'none';
    document.querySelector('#map').style.display = 'none';
    const results = document.querySelector('#results');
    const friends = await Model.getFriends({ fields:
'photo_100,online'});
    results.innerHTML = View.render('friendsOnline', {list:
friends.items});
};
```

showMap.js

```
const Model = require('./model.js');
const View = require('./view.js');
let myMap;
//mapFriends
module.exports = async function() {
    document.querySelector('#results').innerHTML="";
    var cityBlock = document.querySelector('#city');
    cityBlock.innerHTML = "";
    cityBlock.style.display = 'block';
    document.querySelector('#map').style.display = 'block';
    const friends = await Model.getFriends({ fields:
'photo_100,city'});
    await ymaps.ready(function (){
        //Если карта ещё не создана, создаём
        if(myMap === undefined) myMap = new ymaps.Map('map',{
center: [55.76, 37.64], zoom: 12 });
        myMap.geoObjects.removeAll();//удаляем все прошлые метки
с карты
        let mapa = new Map();//Создаём список пар ключ - город,
значение - объект friend
        for(let i = 0; i < friends.items.length; i++){//проходим
по списку друзей
            if(friends.items[i].city != undefined){//Если город
задан в профиле
                //обращаемся по ключу - город, и приводим
значение к типу "массив"
                mapa[friends.items[i].city.title] =
mapa[friends.items[i].city.title] || [];
                //Добавлем в конец массива значение

mapa[friends.items[i].city.title].push(friends.items[i]);
            }
        }
```

```
        for(let city in mapa){//Проходим по мапе
            ymaps.geocode(city).then(function(res){//получаем
координаты города через геокодер
                var coord =
res.geoObjects.get(0).geometry.getCoordinates();
                //Создаём метку по координатам
                var myPlacemark = new
ymaps.Placemark(coord,{hintContent : 'Клик для просмотра друзей'
});
                //Добавляем событие клик по метке
                myPlacemark.events.add('click',function(){
                    const results =
document.querySelector('#results');
                    cityBlock = document.querySelector('#city');
                    cityBlock.innerHTML = city+":";
                    //Данной метке будет соответствовать свой
список пользователей
                    results.innerHTML =
View.render('friends',{list: mapa[city]});
                });
                //Добавляем метку на карту
                myMap.geoObjects.add(myPlacemark);
            });
        }
    });
};
```

showNews.js

```
const Model = require('./model.js');
const View = require('./view.js');

module.exports = async function() {
    document.querySelector('#city').style.display = 'none';
    document.querySelector('#map').style.display = 'none';
    const results = document.querySelector('#results');
    var news = await Model.getNews({ filters: 'post' , count: 20
});
    results.innerHTML = View.render('news', {list: news.items});
}
```