МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
02.03.02 — Фундаментальная информатика
и информационные технологии

РАЗРАБОТКА СИСТЕМЫ ПРОВЕДЕНИЯ ОПРОСОВ АУДИТОРИИ ВО ВРЕМЯ ПУБЛИЧНЫХ ВЫСТУПЛЕНИЙ

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
Е. А. Тактарова

Научный руководитель:
к.ф.-м.н., доцент Е. М. Андреева

Допущено к защите:
руководитель направления ФИИТ _____ В. С. Пилиди

Ростов-на-Дону
2019

Содержание

Введение

Технологии проведения публичных выступлений и презентаций затрагивают навыки ораторства и внешний вид, дизайн медиасопровождения. Методы взаимодействия с аудиторией традиционно включают в первую составляющую. Выступающий, желающий взаимодействовать со слушающими, должен уже обладать определенным опытом в работе с ними и ограничен устными средствами. Крайне редко возможно почти полностью вовлечь аудиторию в выступление, ведь лишь немногие слушатели готовы, например, задать вопрос или ответить выступающему.

Распространение телефонов и мобильного доступа в интернет, позволяет использовать эти устройства как средства взаимодействия с аудиторией. Проекты, использующие эту идею, реализовывались неоднократно, но ни один из них не закрепился как широко используемый в публичных выступлениях. В первую очередь, идея взаимодействия с публикой через телефоны реализовывалась под конкретные единичные выступления. Последующие реализации, хотя и обладают обширным функционалом, в виде опросов, голосований и чатов, представлют собой отдельные веб-сервисы, направленные на монетизацию с пользователей. Все проекты закрыты проприетарными лицензиями и требуют от пользователей загрузки презентации на сторонний сервер.

Данная работа посвящена разработке проекта портативного веб-сервиса под свободной лицензией, который позволит проводить опросы аудитории во время публичных выступлений без привлечения сторонних сервисов. Свободная лицензия позволит любому человеку изменять и расширять возможности сервиса под свои нужды.

Задача по созданию такого проекта включает разработку как веб-интерфейса пользователя (фронтенд), так и внутренней логики сервиса (бэкенд), которые в совокупности обеспечат динамичное отображение результатов опросов.

1. Исследование предметной области

1.1. Обзор существующих решений

Как и упоминалось ранее, для опросов аудитории уже существует немалое число инструментов, однако в основой массе это закрытые решения в виде веб-сервисов:

- polleverywhere.com
- directpoll.com
- sli.do
- ficus.io

На этих сайтах и других подобных можно бесплатно один раз провести опрос или даже презентацию, но повторные показы и дополнительные функции ограничены для пользователей, не оплативших услуги сайтов. Более того, даже оплативший пользователь ограничен средствами и функциями сайта и не может модифицировать или изменить инструмент под свои нужды и цели.

Также стоит упомянуть об инструментах опросов, не использующих только Интернет [1]. Такие решения применяются в университетах США [2] и отличаются низкой способностью к масштабированию и высокой ценой как системы, так и индивидуальных приборов голосования.

1.2. Обзор инструментов разработки

При создании веб-сервиса самую важную роль занимает разработка серверной части. Так как веб-сайт должен динамически взаимодействовать с сервером, то архаичная связка из веб-сервера и CGI приложения очевидно не подойдет. Для решения данной задачи необходимо выбрать один из множества современных веб-

фреймворков [3], как основу для проекта. Отметим основные необходимые для задачи черты фреймворков:

1. легковесность
2. инкапсуляция веб-сервера
3. наличие актуального функционала(**JSON**, **AJAX**, **WebSocket**)

Рассмотрим несколько популярных фреймворков:

- **Django** — фреймворк на языке Python. Хотя на нем можно реализовать необходимый нам функционал, но его врядли можно назвать легковесным. Django в первую очередь предназначен для создания больших многостраничных сайтов и сервисов, которые будет длительное время поддерживать команда разработчиков и администраторов. Наличие бесполезного для задачи функционалла негативно сказывается на времени освоения и разработки [4].
- **Ruby on Rails** — фреймворк на языке Ruby. Основными минусами Ruby on Rails являются проксирование через отдельный веб-сервер и общая сложность освоения как фреймворка, так и самого языка. Стоит также отметить, что этот фреймворк сильно опирается на архитектуру модель-представление-контроллер, реализация которой усложняет задачу для небольшого приложения [5].
- **Express** — фреймворк на языке JavaScript, запускаемый на платформе Node.js. Express инкапсулирует веб-сервер,представляя только абстракцию в виде объектов HTTP запроса и ответа, а необходимый для приложения функционал, например WebSocket и шаблонизация, добавляются через совместимые модули Node.js. Так же фреймворк не затрагивает клиентскую часть веб-приложения [6]. Express своевременно обновляется, имеет обширную документацию и является популярным выбо-

ром среди разработчиков из-за своей простоты и понятности. Основным опасением является производительность однопоточной архитектуры Node.js, однако для небольших и средних приложений Node.js и Express показывают удовлетворительные результаты [7].

Таким образом, для решения рассматриваемой задачи фреймворк Express подходит идеально. Так как он не затрагивает клиентскую часть сервиса, то для нее можно использовать любой легковесный JavaScript-фреймворк для создания пользовательского интерфейса.

Также важно выбрать технологию обмена данных между сервером и клиентом. Так как в нашем случае сервер должен отправлять данные, даже когда клиент не запрашивает их явно, то данное веб-приложение соотвествует модели **COMET** [8]. Рассмотрим некоторые технологии реализации **COMET** в современных веб-приложениях:

Спрятанный iframe HTML-элемент **iframe**, который подгружает «бесконечную» подстраницу с сервера, состоящую из элементов **script** с данными. Такой метод требует особой модификации веб-сервера, но поддерживается во всех браузерах. Достаточно сложен в реализации как на сервере, так и на клиенте.

подгружаемый XMLHttpRequest аналогично с предыдущим методом, но вместо страницы загружается AJAX-запрос.

XMLHttpRequest long polling в этой технологии браузер отправляет AJAX-запрос, но ответ получает, только когда серверу нужно отправить данные клиенту.

WebSocket отдельный протокол двустороннего соединения поверх TCP. Его реализация есть во всех современных браузерах и серверах, в том числе и для Node.js. По сравнению с другими технологиями WebSocket отличается производительностью, скоростью передачи данных и удобством в работе.

Исходя из этого, для данной задачи лучше всего использовать технологию WebSocket.

Реализация современных веб-сайтов без использования фреймворков для создания интерфейса и дизайна не является актуальной задачей. Вручную описывать манипуляции с иерархической структурой HTML-страницы на чистом JavaScript крайне сложно даже для данного небольшого проекта с динамическим содержанием. Стоит отметить, что разнообразие размеров экранов мобильных устройств и персональных компьютеров также осложняет ручную верстку страницы.

Существует огромное количество фронтенд-фреймворков, вряд ли среди них можно выбрать абсолютно лучший, поэтому выбор фреймворка для маленького или среднего проектов это личное предпочтение. Я буду использовать фреймворк Vue.js. Основными его преимуществами являются:

Реактивность Данные связываются с интерфейсом, который изменяется автоматически, когда данные обновляются.

Легковестность и скорость по заявлению разработчиков, Vue.js быстрее и меньше других аналогичных фреймворков [9].

Независмость от серверной части Хотя Vue.js может интегрироваться в сервер для улучшения производительности и работы с поисковыми системами [10], однако подключение фреймворка через HTML-тег **script** успешно работает для ненагруженных страниц.

2. Постановка задачи

Решение задачи будет представлено в виде веб-сервиса на фреймворке Express на Node.js, имеющего следующие отличительные функции и особенности:

- Создание и проведение опросов.

- Динамическое отображение результатов опроса на странице.
- Каждый опрос имеет короткие ссылки для голосования и просмотра результатов.
- Параллельное проведение нескольких опросов на одном вебсервисе.
- Защита от вредоносного искажения результатов.
- Взаимодействие клиента и сервера через технологию WebSocket.
- Открытый исходный код под свободной лицензией.

3. Аспекты реализации

3.1. Структура проекта на Node.js

Для создания основы проекта с использованием Express я использовал генератор проектов **express-generator** [11]. Эта утилита позволяет избавится от необходимости конфигурировать встроенный сервер и параметры фреймворка. После ее выполнения в выбранном каталоге генерируется простое, готовое для запуска вебприложение структуры, представленной на листинге 3.1.

Разберем назначение некоторых файлов и каталогов:

package.json файл с указаниями для пакетного менеджера npm для Node.js. Содержит список всех пакетов, от которых зависит приложение, и точку входа для Node.js.

bin/www точка входа исполняемого кода JavaScript. В нем создаются и соединяются объекты HTTP-сервера и Express-приложения. Все остальные файлы кода для Node.js являются JavaScipts-модулями, которые подключаются в **www**, либо в других модулях, уже подключенных в **www**.

Листинг 3.1. Структура шаблонного проекта Express

```
sample-app
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

app.js файл с настройками, касающихся конкретно работы Express. Также в нем подключаются файлы с обработчиками путей запросов.

public/ в настройках проекта этот каталог сконфигурирован как общедоступный для HTTP-запросов вида GET. В нем хранятся статичные файлы для клиентской части веб-сервиса. Во время его работы эти файлы доступны по запросам вида **http://project.com/stylesheets/style.css**.

routes/ содержит обработчики путей запросов, поступающих в Express.

views/ содержит файлы шаблоны веб-страниц в формате PUG.

Такой шаблон проекта значительно упростил разработку сервиса, потому что любой написанный поверх него функционал сразу можно запустить и протестировать, а готовое файловое устройство

принуждает к структурированному стилю написания кода. В конечном проекте структура каталогов приложения остается такой же, и только изменяются и добавляются в них файлы кода.

3.2. Использование фреймворка Express

Разработка веб-сервиса на Express сводится к написанию обработчиков путей HTTP-запросов поступающих на сервер.

Листинг 3.2. Пример функции обработчика GET-запроса

```
router.get("/get_polled",function(req, res, next){
 let id = db.create_polling_session({
        title: "poll",
        options: [
          { title: "option1" },
          { title: "option2" },
          { title: "option3" }
 ]});
 res.status(200).json({link:db.polling_sessions[id].view_link});
});
```

В примере (листинг 3.2) для объекта **router**, который затем можно экспортировать для использовании в приложении Express, определяется поведение при поступлении GET-запроса по адресу **sample.com/get_polled**. Для этого используется анонимная функция обратного вызова, которая исполняется каждый раз при поступлении запроса. Аргументы **req**, **res**, **next** представляют, соотвественно, объекты с данными о запросе(**REQuest**), с функциями ответа (**RESponse**) и функцию для вызова следующего подходящего обработчика пути. В примере происходит взаимодействие с моделью данных и отправка клиенту ответа с результатами взаимодействия в формате JSON.

Стоит отметить, что возможность использовать регулярные выражения и сопоставление с образцом для путей запросов, а также

выстраивать цепочки из обработчиков позволило выстроить в веб-сервисе относительно сложную логику ответов на запросы. Хотя описание этой логики в исходном коде не занимает много места.

3.3. Использование WebSocket совместно с фреймворком Express

Для того чтобы добавить функциональность WebSocket в приложение, я воспользовался пакетом **express-ws** [12]. Этот пакет добавляет в Express функцию для создания обработчиков путей запроса соединения по протоколу WebSocket. Чтобы добавить желаемый функционал код из пакета модифицирует прототипы класса Router внутри Express и добавляет в указанный веб-сервер обработчик запросов типа:

```
GET /api/websocket HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
...
```

После этого в контексте фреймворка можно писать обработчики путей для WebSocket:

Листинг 3.3. Пример функции обработчика WS-запроса

```
app.ws('/echo', function(ws, req) {
        ws.on('message', function(msg) {
                ws.send(msg);
        });
});
```

В примере (листинг 3.3), в отличие от примера в листинге 3.2, нет объекта ответа **res**, но есть объект **ws** — асинхронно открывающийся веб-сокет, представляющий соединение между конкретным клиентом и сервером. К этому объекту уже можно привязать функции обра-

ботчики для событий изменения состояния вебсокета или получения данных. Когда состояние вебсокета изменится на **OPEN**, для объекта **ws** можно вызвать метод **send** и отправить клиенту любые данные.

Как было сказано в подразделе 3.1 в шаблоне **express-generator** создание объекта HTTP-сервера и связывание его с экземпляром объекта Express происходит в файле **bin/www**, который является входной точкой исполнения кода для Node.js. Такое устройство приложения по-умолчанию не подходит для **express-ws**, которому нужно вмешаться в устройство сервера и экземпляра Express. Чтобы это исправить, необходимо внести определенные изменения в файлы **app.js** и **bin/www**.

Пользуясь тем, что **app.js** импортируется в начале **www/bin**, а значит и интерпретируется раньше чем его основной код, я перенес создание экземпляров HTTP-сервера и Express в **app.js**. Там же происходит внедрение **express-ws**.

Листинг 3.4. Изменения в app.js

```
        ...
var express = require('express');
var http = require('http');
        ...
var app = express();
app.server = http.createServer(app);
var expressWS = require('express-ws')(app, app.server);
        ...
module.exports = app;
```

Внутри **bin/www** создание сервера я заменил на обращение к ссылке на веб-сервер из экземпляра Express.

Листинг 3.5. Изменения в bin/www

```
var app = require('../app');
        ...
var server = app.server;
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
        ...
```

Таким образом, можно воспользоваться технологией WebSocket, не нарушая структуру шаблонного приложения на фреймворке Express.

3.4. Проектирование и разработка модели данных

Модель данных веб-сервиса представляет информацию о его текущем состоянии и о пользовательских данных. Обычно часть информации хранится в базе данных, отделенной от основного сервера. Такой подход удобен для масштабных проектов, где важна сохранность данных в случае неполадок или ошибок в сервисе. Однако для данного проекта интегрирование работы с базой данных будет являться только обременением, потому что в нем нет важной информации, которую нужно хранить между перезапусками.

В этом проекте я использовал JavaScript-объект, который содержит все другие объекты-представления данных и методы взаимодействия с ними. Этот объект и весь сопутствующий код описываются в отдельном модуле **database.js**, который экспортирует экземпляр объекта. Таким образом, код в модуле интерпретируется только один раз, и, в какой бы точке проекта не был бы импортирован объект с моделью данных, это всегда будет один и тот же экземпляр.

Прежде всего нужно было определится с тем, какие данные необходимо хранить модели и как с ними будет взаимодействовать

остальная логика сервиса. Для этого был составлен список требований, которым должна отвечать модель:

- В модели может существовать неограниченное количество параллельных сессий, которые могут перемещаться между своими опросами.
- Каждая сессия имеет две короткие ссылки для просмотра и участия в опросе.
- Имея короткую ссылку, код должен уметь быстро переходить к данным о сессии, которой она принадлежит.
- Код должен быстро получать список пользователей, показывающих опрос или в нем участвующих.
- Пользователь может голосовать и перезагружать страницу неограниченное число раз, не вызывая подтасовку результатов.
- Модель должна быть устойчивой к добавлению новых видов взаимодействия пользователей с сервисом.

На основе этих требований я разработал следующую модель представления данных (см. листинг 3.6). В основе реализации модели лежит использование некоторых JavaScript-объектов как словарей, где ключи это свойства объекта, например словарь сессий опроса **polling_sessions** или словарь всех пользователей подключенных к сессии для просмотра опроса **views**. Ключи генерируются, как уникальные случайные строки. Стоить отметить, что для таких объектов нельзя определять методы, потому что в JavaScript метод также является свойством, и тогда возникнут трудности, если понадобится перебрать все элементы словаря.

Каждый пользователей в сессии представлен лишь уникальным идентификатором, по которому доступен экземпляр WebSocket. Существует два вида пользователей: просматривающие результаты опросов и в нем участвующие. Для краткости такие пользователи бу-

Листинг 3.6. Пример объекта модели во время работы приложения

```
{
 polling_sessions: {
  aebd99ccd3: {
        views: { "6feb3f116c": {WebSocket Object} },
        slaves: { "641b151c81": {WebSocket Object} },
        password: "admin",
        view_link:"ecdd",
        slave_link:"79dd4b",
        state: {
                current_poll: 0,
                title: "poll1",
                options: [
                 { title: "option1", count: 1 },
                 { title: "option2", count: 0 },
                 { title: "option3", count: 0 }
                ],
                voters: {"641b151c81": 0 }
         },
        polls: [
                {title: "poll1",
                 options: [
                        { title: "option1" },
                        { title: "option2" },
                        { title: "option3" }
                 ]
                },
                {title: "poll2",
                 options: [
                        { title: "option3" },
                        { title: "option4" },
                        { title: "option5" }
                 ]
                }
        ]
  }
 },
 short_links: {
        "79dd4b":{ type:"slave", session: {Session Object}},
        "ecdd":  { type: "view", session: {Session Object}}
        }
};
```

дут называться **view** и **slave** соотвественно. Если пользователь перезагрузит страницу, то его экземпляр WebSocket изменится, но по идентификатору модель его распознает и обновит экземпляр. Также по идентификаторам модель отслеживает, кто и за что проголосовал в опросе. Отдать два голоса невозможно в принципе.

Для объектов, которые не являются словарями, например сессии опросов **«aebd99ccd3»**, можно определять любые методы, инкапсулирующие взаимодействие данных и пользователей. Так, например, в файле **database.js** функция **locac_db.patch_polling_session** добавляет к передаваемому ей экземпляру новой сессии все методы, которые меняют ее состояние.

3.5. Взаимодействие клиента и сервера через протокол WebSocket

В клиентской части веб-приложения за работу с WebSocket отвечает статичный JavaScript-файл **ws.js**, единый как для **slave**, так и для **view** пользователей. Этот скрипт извлекает идентификатор пользователя из страницы или cookie, если он там есть, а затем открывает WebSocket-соединение по адресу:

```
ws:[URL веб-сервиса]/ws/[короткая ссылка]
```

Короткая ссылка в адресе соединения позволяет серверу понять, к какой сессии принадлежит пользователь. Сразу после открытия подключения (событие **open** у объекта WebSocket), клиент отправляет свой идентификатор серверу и переходит в режим ожидания ответных сообщений, содержащих, например, состояние страницы.

Листинг 3.7. Обработчик нового WebSocket соединения, поступившего на сервер

```
var db = require('../database.js');
        ...
router.ws('/ws/:shortlink',function(ws,req){
 let link = db.short_links[req.params.shortlink];
 if(!link)
  next();
 let id_type = link.type;
 let poll_session = link.session;
 let got_id = false;
 ws.on('message', function(data){
  got_id = true;
  data = JSON.parse(data);
  if(!poll_session.verify_update(data.session_id,id_type,ws))
   ws.close();
 });
 setTimeout(function(){
  if(!got_id)
   ws.close();
 },4000);
});
```

На стороне сервера новое WebSocket соединение не сразу передается под управление модели. Как видно в листинге 3.7, новое соединение должно пройти ряд проверок перед отправкой в модель:

- Короткая ссылка из пути запроса должна существовать в модели.
- В течение четырех секунд клиент должен прислать приемлемый JSON со своим идентификатором, иначе соединение будет закрытою.

и в самой модели внутри функции **verify_update**:

- Идентификатор пользователя должен существовать в сессии.

- Заявленный через короткую ссылку тип пользователя должен совпадать с типом полученного идентификатора.

Провал хотя бы одной проверки приводит к закрытию соединения и освобождению идентификатора. Если же проверки успешны, то объект сохраняется в сессию, и функция **polling_session_object.patch_websocket** расширяет обработчик события сообщения обработчиками **polling_session_object.message_from_slave** или **polling_session_object.message_from_view**, которые получают через аргументы не только новые данные, но идентификатор и ссылку на объект WebSocket.

Как показало тестирование и отладка приложения, оказалось очень важно удалять ненужные обработчики событий, так, например, неубранный обработчик на событие **message**, приводил к множественным вызовам **polling_session_object.patch_websocket**, которые добавляли еще больше обработчиков на один WebSocket.

На основе этой архитектуры я выстроил простой протокол сообщений в формате JSON (см. листинг 3.8), через которые клиенты могут получать новые состояния страниц, голосовать на опросах, повыситься до владельца опроса, листать опросы и так далее.

Листинг 3.8. Пример сообщения клиенту от сервера с текущим состоянием приложения

```
{
 "session_id": "4e643652f903975fbdd304e0115bfcb6f2a53f35",
 "type": "new_state",
 "state": {
   "title": "poll",
   "options": [
        { "title": "option1", "count": 3 },
        { "title": "option2", "count": 0 },
        { "title": "option3", "count": 10 },
   ]
  }
}
```

В случае изменения данных в модели, например **slave**-клиент проголосовал в опросе, нужно послать всем пользователям обновленные данные. Если просто вызывать функцию отправки данных всем **view**-клиентам в конце callback-функции обработки сообщения от одного **slave**-клиента, то даже при паре пользователей, непрерывно меняющих голос в опросе, сервер перестанет справляться с объемами отправляемой информации, и пользователям будет приходить запоздалая и неактуальная информация.

Чтобы решить эту проблему, я воспользовался пакетом debounce [13] для Node.js. Debounce предоставляет обертку для функций, не позволяющую вызывать функцию чаще, чем в определенный промежуток времени. Вызовы, произошедшие в промежуток, откладываются в очередь и выполняются через этот промежуток.

Листинг 3.9. Использование debounce для контроля отправки нового состояния всем пользователям

```
polling_session_object.send_state_all = debounce(function() {
 var state = { title: polling_session_object.state.title, options
    : polling_session_object.state.options };
 for (var id in polling_session_object.views) {
                 ...
 view.send(JSON.stringify({ session_id: id, type: "new_state",
    state: state }));
  }
 for (var id in polling_session_object.slaves) {
                 ...
 view.send(JSON.stringify({ session_id: id, type: "new_state",
    state: state }));
 }
 this.send_state_all.clear();
},200);
```

Также важно очищать очередь вызовов после отработки кода через **debounce.clear** как в примере на листинге 3.9. Тогда нескольким

вызовам обернутой функции, произошедшим в один промежуток, соответствует только одно исполнение кода.

3.6. Создание пользовательского интерфейса

В данном веб-сервисе существует два основных вида страниц, которые я реализовал в первую очередь. Страница просмотра результатов опроса и страница голосования, которые в модели данных представляются **view** и **slave** пользователями соответственно.

Страница просмотра результатов не только динамично обновляет результаты опроса, но также показывает ссылку по которой можно присоединится к нему и, при вводе секретной фразы, указанной при создании опроса, дает пользователю контроль над проведением опроса.

Для того что бы отправлять страницы пользователям, я использую шаблонный движок Pug. Его синтаксис состоит из смеси JavaScript и видоизмененного HTML (см. листинг 3.10). С помощью табуляции описывается древовидная структура страницы, точка обозначает блок сплошного текста, а «#...» окружает JavaScript-код, который интерпретируется на стороне сервера, а результат записывает на его место [14]. Pug — мощный инструмент шаблонизации, но в данном проекте я использую его, чтобы передать пользователю только основу страницы, в которой указаны ссылки на загрузку частей фреймворка и начальные данные о состоянии страницы.

Основную работу по отображению элементов выполняет фреймворк Vue.js. Ядро фреймворка загружается на страницу из CDN (пер. Сеть доставки содержимого) через тег **script**. После этого в последующих скриптах можно описывать компоненты страницы как объекты Vue.

В файлах **javascripts/slave_ui.js** и **javascripts/view_ui.js** я определил Vue-компоненты для соответствующих страниц. Компоненты напрямую связаны с данными, например, о состоянии страницы, и

Листинг 3.10. Шаблон разметки страницы на языке Pug

```
doctype html
html
 head
  meta(charset="utf-8")
  script(src="/javascripts/js.cookie.js")
  link(rel="stylesheet" href="https://unpkg.com/vue-material@beta
     /dist/vue-material.min.css")
 body
  script(type='text/javascript').

     var sessionInfo = {
                id : "#{session_id}",
                type : "#{session_type}",
                viewLink: "#{view_shortlink}",
                slaveLink:  "#{slave_shortlink}"
        }

  main#app
        block content
  script(src="/javascripts/ws.js")
  script(src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js")
  script(src="https://unpkg.com/vue-material@beta")
  script.
        Vue.use(VueMaterial.default);
  block footer_scripts
```

если изменяются данные, то компоненты перерисовываются. Синтаксис шаблонов Vue позволяет описывать элементы через условия и циклы, чем я и пользуюсь, чтобы отображать опросы с любым количеством вариантов.

В файле **javascripts/ws.js** описана логика соединения с сервером через WebSocket, но ее запуск обернут в метод **sessionInfo.init_WS**. Это связано с тем, что экземпляру Vue требуется время, чтобы полностью создаться и связать данные с компонентами, поэтому вызов **init_WS** происходит только после события **created** в экземпляре Vue. Тогда фреймворк уже отслеживает изменения данных.

4. Обзор проекта

4.1. Внутреннее устройство

За время разработки общая структура проекта не менялась кардинально, JavaScript-модули и другие файлы кода, реализующий новый функционал, добавлялись в соответствующие каталоги. В текущей итерации разработки проект имеет структуру, представленную на листинге 4.1. Некоторые из новых файлов уже упоминались выше, но все же разберем каждый из них:

javascripts/..._ui.js Это JavaScript-файлы, в которых содержится описание интерфейса каждой страницы в контексте фреймворка Vue.js.

javascripts/ws.js В этом файле описана логика соединения клиента и сервера по протоколу WebSocket.

javascripts/js.cookie.js Свободная легковесная библиотека для работы с куки на стороне клиента.

stylesheets/..._style.css Каскадные таблицы стилей для каждой страницы.

Листинг 4.1. Структура проекта Poll_app

```
poll_app
├── bin
│   └── www
├── public
│   ├── images
│   ├── javascripts
│   │   ├── builder_ui.js
│   │   ├── index_ui.js
│   │   ├── slave_ui.js
│   │   ├── view_ui.js
│   │   ├── js.cookie.js
│   │   └── ws.js
│   └── stylesheets
│       ├── builder_style.css
│       ├── index_style.css
│       ├── slave_style.css
│       └── view_style.css
├── routes
│   └── poll.js
├── views
│   ├── error.pug
│   ├── index.pug
│   ├── layout.pug
│   ├── builder.pug
│   ├── view.pug
│   └── slave.pug
├── app.js
├── database.js
├── package.json
├── LICENSE
└── README.md
```

views/ ... .pug Файлы HTML-шаблонов страниц в формате Pug. Каждый шаблон подключает файл **layout.pug**, в котором описаны общие для всех страниц HTML-элементы.

routes/poll.js Файл, в котором описаны все пути HTTP-запросов, поступающих в приложение, и их обработчики в контексте фреймворка Express.

database.js JavaScript-модуль, экспортирующий экземпляр объекта модели данных в приложении.

Приложение состоит из четырех страниц. По адресам «/» и «/**index**», доступна начальная страница. На ней отображаются все текущие и доступные сессии опросов. К текущим сессиям можно присоединиться как зритель или участник. А доступные опросы можно начать, если ввести кодовое слово, с которым опрос был создан.

На странице «/**build_poll**» можно создать опрос и отправить его на сервер. Указанное при создании кодовое слово затем позволит запустить опрос или управлять им во время показа.

Страницы просмотра и участия в опросе имеют уникальные для каждого опроса короткие ссылки. Страница участия оптимизирована в первую очередь для мобильных устройств. Со страницы просмотра можно управлять ходом опроса, если ввести кодовое слово. Содержание этих страниц обновляется динамически, когда изменяется состояние опроса.

4.2. Установка и использование проекта

Исходные коды проекта можно получить в Git репозитории по ссылке: github.com/Jeday/Sycidium.git

Для GNU/Linux систем:

1. Установите последние версии Node.js и npm.
2. Склонируйте файлы приложения в отдельную папку.

3. В корне приложения выполните **npm install**, чтобы установить зависимости.

4. Запустите приложение через **npm start**.

5. Приложение доступно по адресу **http://localhost:3000/**

Чтобы запусть приложение в рабочем режиме с произвольным портом нужно изменить порт в файле **package.json** и выполнить **npm run prod**.

Листинг 4.2. Файл package.json с установленным портом 80

```
{
 "name": "poll-app",
 "version": "0.7.0",
 "private": true,
 "scripts": {
        "start": "node ./bin/www",
        "prod": "NODE_ENV=production PORT=80 node ./bin/www"
    },
 "dependencies": {
        ...
 }
}
```

Заключение

В рамках данной работы был реализован полноценный веб-сервис для проведения опросов во время публичных выступлений. Веб-сервис позволяет создавать, запускать и управлять опросами.Пользователи могут сразу же поучаствовать в опросе со своего мобильного устройства, перейдя по короткой ссылке, указанной на экране опроса. А их выбор моментально отобразится на главном экране. Также реализована основная защита от вредоносного вмешательства в процесс опроса, не позволяющая испортить результаты или остановить ход опроса. Веб-сервис имеет открытые исходные коды и доступен для развертывания, использования и модификации по свободной лицензии MIT.

Наиболее очевидным способом улучшения сервиса является добавление больших возможностей для взаимодействия с аудиторией и интеграцией с потоком презентации. К таким функциям можно отнести: чат, отображающийся на экране; слияние опросов и слайдом презентации; разные формы и виды опросов.

Список литературы

1. OMBEA | Actionable insights in real-time. — URL: http://www.ombea.com/ (дата обр. 20.03.2019).
2. NEA - Clickers and Classroom Dynamics. — URL: http://www.nea.org/home/34690.htm (дата обр. 20.03.2019).
3. Comparison of web frameworks - Wikipedia. — URL: https://en.wikipedia.org/wiki/Comparison_of_web_frameworks (дата обр. 22.03.2019).
4. The Web framework for perfectionists with deadlines | Django. — URL: https://www.djangoproject.com/ (дата обр. 22.03.2019).
5. Ruby on Rails | A web-application framework. — URL: https://rubyonrails.org/ (дата обр. 22.03.2019).
6. Express - фреймворк веб-приложений Node.js. — URL: https://expressjs.com/ru/ (дата обр. 22.03.2019).
7. *Lei K.*, *Ma Y.*, *Tan Z.* Performance Comparison and Evaluation of Web Development Technologies in PHP, Python and Node,js // Computational Science and Engineering (CSE), IEEE. — 2014. — С. 661—668.
8. *Krill P.* AJAX alliance recognizes mashups // InfoWorld. — 2007.
9. Comparison with Other Frameworks — Vue.js. — URL: https://vuejs.org/v2/guide/comparison.html (дата обр. 24.03.2019).
10. Vue.js Server-Side Rendering Guide | Vue SSR Guide. — URL: https://ssr.vuejs.org/#what-is-server-side-rendering-ssr (дата обр. 24.03.2019).
11. express-generator — npm. — URL: https://www.npmjs.com/package/express-generator (дата обр. 24.03.2019).

12. express-ws — npm. — URL: https://www.npmjs.com/package/express-ws (дата обр. 25.03.2019).
13. debounce — npm. — URL: https://www.npmjs.com/package/debounce (дата обр. 30.03.2019).
14. Interpolation – Pug. — URL: https://pugjs.org/language/interpolation.html (дата обр. 02.04.2019).