**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ИНДИВИДУАЛЬНОЕ ДОМАШНЕЕ ЗАДАНИЕ**

**по дисциплине «Разработка ПО информационных систем»**

**Тема: Разработка приложения для составления оптимального маршрута парковки вокруг точки назначения**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 4303 |  | Осмоловский П.Д.  Полушина Ю.И.  Таланов С.С. |
| Преподаватель |  | Заславский М.М. |

Санкт-Петербург

2017**ЗАДАНИЕ**

**НА ИНДИВИДУАЛЬНОЕ ДОМАШНЕЕ ЗАДАНИЕ**

|  |  |
| --- | --- |
| Студенты  Осмоловский П.Д.  Полушина Ю.И.  Таланов С.С. | |
| Группа 4303 | |
| Тема проекта: Разработка приложения для составления оптимального маршрута парковки вокруг точки назначения. | |
| Исходные данные:  Необходимо реализовать приложение, позволяющее составить оптимальный маршрут парковки вокруг точки назначения. | |
| Содержание пояснительной записки:  «Содержание»  «Введение»  «Качественные требования к решению»  «Сценарии использования»  «Модель данных»  «Разработанное приложение»  «Заключение»  «Приложения»  «Список использованных источников» | |
| Предполагаемый объем пояснительной записки:  Не менее 10 страниц. | |
| Дата выдачи задания: | |
| Дата сдачи реферата: | |
| Дата защиты реферата: | |

|  |  |  |
| --- | --- | --- |
| Студенты гр. 4303 |  | Осмоловский П.Д.  Полушина Ю.И.  Таланов С.С. |
| Преподаватель |  | Заславский М.М. |

**АННОТАЦИЯ**

В работе описано приложение, решающее проблему поиска места для парковки автомобиля. Разработанное приложение является веб-сервисом, использует графовую СУБД в качестве основного хранилища данных и картографический проект OpenStreetMap как источник информации.

**SUMMARY**

This explanatory note contains application description. The application solves the problem of finding a parking spot for a car. The developed application is a web-service, uses a graphical DBMS as the main data store and the OpenStreetMap mapping project as a source of information.

**СОДЕРЖАНИЕ**

[**1. КАЧЕСТВЕННЫЕ ТРЕБОВАНИЯ К РЕШЕНИЮ**](#_ewco9p4klfdt) **7**

[**2. СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ**](#_dryxpzt1176x) **8**

[2.1. Макет UI](#_3w6aasisjlcp) 9

[2.2. Описание сценариев использования](#_15zci4w4ytbv) 9

[**3. МОДЕЛЬ ДАННЫХ**](#_dmkdcie7frdb) **12**

[3.1. Нереляционная модель](#_wchzhbn469m) 12

[3.2. Реляционная модель](#_5dezy9da5crb) 13

[3.3. Сравнение моделей](#_djxdzqr5vmsb) 14

[**4. РАЗРАБОТАННОЕ ПРИЛОЖЕНИЕ**](#_r0e700mg1b3k) **15**

[4.1. Краткое описание](#_7ey1tvfb5nxd) 15

[4.2. Технологии](#_33afata45lci) 15

[4.3. Алгоритм](#_stpfe0n75dxe) 16

[Реализованный алгоритм.](#_duc3hc8it600) 16

[Оптимальный алгоритм.](#_qvv0odt7bz25) 16

[**ЗАКЛЮЧЕНИЕ**](#_nekwuft9unq) **20**

[**ПРИЛОЖЕНИЯ**](#_qa0uh0hb3yes) **22**

[Сборка приложения](#_1wjdak3u9uw8) 22

[Снимки экрана приложения](#_1wjdak3u9uw8) 22

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**](#_6z2pqz1em7tv) **23**

**ВВЕДЕНИЕ**

В данной работе было спроектировано и реализовано приложение, которое поможет решить извечную проблему автомобилистов всего мира и позволит более не ломать голову над тем, какой маршрут стоит выбрать водителю, чтобы иметь наибольший шанс припарковаться возле точки назначения, не покинув при этом ее зону пешей доступности.

# 1. КАЧЕСТВЕННЫЕ ТРЕБОВАНИЯ К РЕШЕНИЮ

Решение должно представлять собой программный продукт, предоставляющий пользователю минималистичный и привлекательный интерфейс.

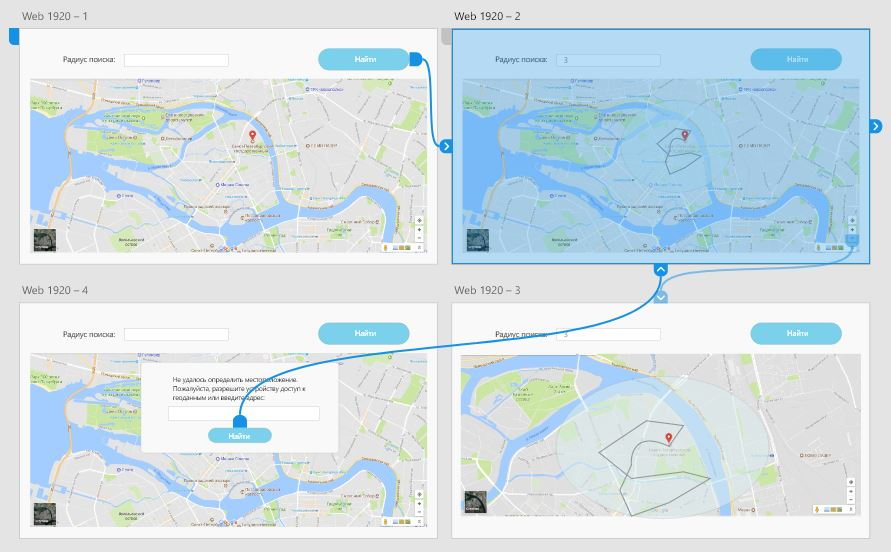
Решение должно обладать следующими качествами:

* Графический вывод, наглядность. Так как в процессе работы программы строится маршрут, необходимо в наиболее удобной форме вывести результаты пользователю.
* Обработка ошибок - приложение должно сообщать пользователю о неверном вводе и корректно обрабатывать все исключительные ситуации.
* Доступность - приложение должно быть легко доступно пользователю, то есть распространяться в виде установочного файла или же быть доступным из браузера.
* Импорт данных. Необходимо предоставить возможность частичного импорта данных, т.к. загрузка графов дорог для всех городов является ресурсозатратной задачей.

# 2. СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

## 2.1. Макет UI

Макет интерфейса представлен на рисунке №1.



*Рис.1 Макет интерфейса*

## 2.2. Описание сценариев использования

*Позиция* - место, обозначенное пользователем, как его текущее положение на карте.

*Пользователь -* потребитель продукта.

*Радиус -* расстояние по прямой до позиции.

*Маршрут* -путь вокруг заданной позиции, включающий в себя наибольшее количество парковочных мест, построенный таким образом, чтобы пользователь не оказался дальше от позиции, чем значение параметра радиус.

1. Авто режим - устройство с поддержкой автоопределения геопозиции. Пользователь открывает приложение, начинается автоматический поиск маршрута, результат выводится на экран. Пользователь может изменить Радиус, тогда маршрут будет перестроен. Пользователь может перейти в режим ручной установки Позиции.

2. Ручной режим - Устройство без поддержки определения геопозиции, либо условия плохого gps сигнала. Пользователь открывает приложение, на экран выводится уведомление о невозможности определения Позиции, пользователя просят ввести адрес Позиции. Пользователь вводит адрес, нажимает кнопку ОК, происходит поиск Позиции по данному адресу. Далее происходит построение маршрута вокруг Позиции. Пользователь может изменить Радиус, тогда маршрут будет перестроен.

3. Полуавтоматический режим. Пользователь действует по 1 или 2 сценарию, после окончания поиска он может с помощью мыши выбрать точку на карте, которая будет выбрана в качестве Позиции. Далее происходит построение маршрута вокруг Позиции. Пользователь может изменить Радиус, тогда маршрут будет перестроен.

Детализация относительно экранных форм:

1. Пользователь открывает приложение, в центре экрана появляется форма процесса поиска. После успешного подключения к серверу и поиска она пропадает и появляется основная экранная форма приложения, представляющая из себя строку для ввода текстового значения адреса для поиска маршрута, светло-синюю иконку определения геопозиции, овальную синюю кнопку “Поиск” и карту, на которой может быть отображен маршрут. В случае, если приложение не может получить маршрут от сервера в течение 30 секунд, на экране снизу появляется серая надпись на желтом фоне с надписью “Соединение не установлено, повторная попытка”, если соединение не было установлено в течение 3 минут, то вместо всего текущего содержимого экрана отображается уведомление “Не удалось соединиться с сервером. Повторите попытку позже.”.

2. При загрузке приложения пользователь нажал кнопку “Отмена”. Открывается главная экранная форма приложения, пользователь может:

* Нажать кнопку в виде стрелки для определения текущей геопозиции. В адресной строке отобразится адрес определенной геопозиции;
* Ввести адрес в адресную строку;
* Установить текущее местоположение на карте с помощью нажатия на нужное место левой кнопкой мыши. В адресной строке отобразится адрес установленной точки;

3. Далее пользователь может нажать на шестеренку с подсказкой “Настройка”. Поверх текущих окон откроется небольшая белая форма, в которой можно задать радиус поиска маршрута, применить настройки клавишей “Ок”, либо оставить предыдущее значение, нажав на крестик в правом верхнем углу.

4. Далее пользователь нажимает кнопку “Поиск”, поверх текущего окна отображается форма процесса поиска.

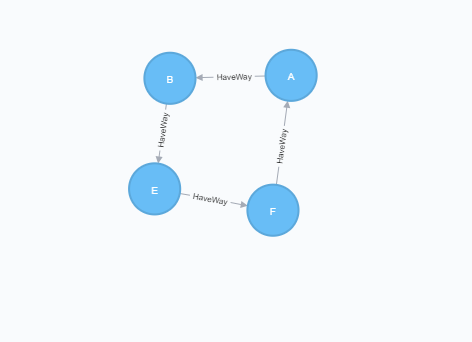
5. Если маршрут не может быть найден, то выводится форма уведомления - светлая форма в красной рамке. Внутри надпись “Маршрут не может быть найден, повторите попытку, либо измените параметры поиска”. Снизу кнопка “Повторить” и “Отмена”. При нажатии на кнопку “Повторить” происходит повторный поиск, “Отмена” открывает основную форму.

Форма процесса поиска - прямоугольная белая форма, внутри которой находится иконка загрузки, которая выглядит как круглый статусбар, внутри текст “Ищу маршрут”, а под ним кнопка “Отмена”

# 

# 3. МОДЕЛЬ ДАННЫХ

## 3.1. Нереляционная модель



*Рис.2 Представление графовой модели данных*

Вершины это перекрестки, их структура {position; name}, где position - координаты перекрестка, а name - его условное обозначение (может отсутствовать).

Ребра это отрезки улиц, соединяющих перекрестки. Их структура {vertA; vertB; name}, где содержатся данные о двух вершинах, которые соединяет отрезок, и название - условное обозначение, может присваиваться по названию улицы, к которой относится данный отрезок.

Оценка удельного объема информации, хранимой в модели (сколько потребуется памяти, чтобы сохранить):

* \_id - счетчик, 8 байт
* name - String , 20 байт
* vertA, vertB - счетчик, по 8 байт, в сумме 16

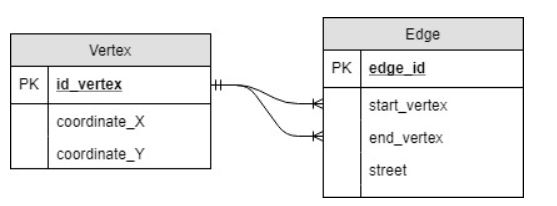
## 3.2. Реляционная модель

Вышеописанную модель данных для хранения дорог также можно представить в виде реляционной модели с помощью таблиц:

1. Множество всех вершин с идентификатором, координатами.

2. Все ребра, содержащие в себе идентификатор, название, стартовую и конечные вершины.

ER-модель представлена на рисунке №&.



*Рис.3 Представление реляционной модели*

Оценка удельного объема информации, хранимой в модели (сколько потребуется памяти, чтобы сохранить)

1. Vertex (26 байта)

* id - счетчик, 8 байт
* coordinate\_x - decimal, 9 байт(при точности 10-19 символов)
* coordinate\_y- decimal, 9 байт(при точности 10-19 символов)

1. Edge (44 байта)

* id - счетчик, 8 байт
* start\_vertex -счетчик, 8 байт
* end\_vertex -счетчик, 8 байт
* name - varchar(20) , 20 байт

Примеры запросов:

1. Найти координаты стартовой вершины

*SELECT Vertex.coordinate\_x, Vertex.coordinate\_y FROM Edge INNER JOIN Vertex ON Edge.start\_vertex = Vertex.id;*

2. Найти координаты конечной вершины

*SELECT Vertex.coordinate\_x, Vertex.coordinate\_y FROM Edge INNER JOIN Vertex ON Edge.end\_vertex = Vertex.id;*

## 3.3. Сравнение моделей

Объем необходимой памяти для реляционной и нереляционной базы данных:

* - количество вершин
* - количество отрезков
* Память\_реляционной\_БД =
* Память\_нереляционной\_БД =

Для большом количестве вершин реляционная БД сильно разрастется и будет занимать больше месте, чем нереляционная БД.

Запросы по юзкейсам: Запросы в реляционной БД на порядок сложнее: используется большое количество join-ов и в получении результата задействованы 2 таблицы.

Любое действие в нереляционной базе данных требуют одного запроса из единственной коллекции.

Можно сделать вывод, что в большинстве случаев преимущество SQL заключается в рациональном использовании памяти, а в NoSQL - в скорости доступа. Данное приложение сильно усложнит использование SQL, т.к. модель данных хоть и небольшая, и занимает небольшой объем памяти, но имеет сложную структуру. Для простейшей операции надо будет использовать не один запрос. В данном приложении приоритет отдается скорости получения информации и простоте реализации.

# 

# 4. РАЗРАБОТАННОЕ ПРИЛОЖЕНИЕ

## 4.1. Краткое описание

## Разработанное приложение представляет собой веб-сервис, состоящий из клиентской и серверной части. Пользователю предоставляется простой интерфейс для ввода радиуса поиска для построения маршрута. В процессе работы отображается текущее состояние приложение. Результирующие данные представлены в виде маршрута на карте.

## 4.2. Технологии

## Приложение имеет монолитную архитектуру.

## Интерфейс приложения реализован с помощью фреймворка Angular4[[3](#_aqllckg8slxi)]. Для работы с картой OpenStreetMap выбрана библиотека ngx-openlayers[[5](#_l0mkyrrzjfhf)].

## Серверная часть приложения разработана с использованием фреймворка Spring Boot[[4](#_6z2pqz1em7tv)]. Клиент-серверное взаимодействие реализовано с помощью REST API.

## В качестве инструмента для хранения данных используется графовая база данных Neo4j[[2](#_ngsxecmtluvc)], так как она имеет встроенное API для поиска кратчайших путей и эффективного импорта большого объема данных.

## 4.3. Алгоритм

## На данный момент в приложение встроен гибкий исследовательский алгоритм, позволивший команде разобраться в основных операциях с графовой базой данных, а также провести анализ зависимостей параметров алгоритма от времени поиска оптимального маршрута. Приведем реализованный алгоритм и тот, к которому необходимо перейти для запуска прототипа продукта.

### Реализованный алгоритм.

Алгоритм работает пошагово с последовательным увеличением радиуса поиска подходящих вершин на заранее заданную величину. Отмеченная пользователем точка принимается за исходную, происходит поиск ближайшего перекрестка, который принимается за стартовую вершину работы алгоритма. Значение шага сейчас - 100 метров. Так, на первом шаге происходит поиск точки в радиусе 100 метров (меньшее значение, если пользователь ввел значение радиуса поиска маршрута не превышающее 100 м), далее алгоритм принимает найденную вершину за стартовую и повторяет поиск до тех пор, пока не выйдет за пределы очерченной введенным пользователем радиусом площади, или у стартовой вершины не окажется ни одного ребенка.

### Оптимальный алгоритм.

*Дано:*

* Граф дорог, в котором вершины - перекрестки, а ребра - дороги.
* Nodes - вершины имеют название улицы, к которой они относятся, координату в формате [WGS-84] [[6]](#_g6og2git8j76)
* Relation — ребра имеют вес (длина в метрах) и направление (дороги бывают односторонние)

*Пользователь задает при использовании приложения:*

* DestinationCoordinate — координату точки назначения (не обязательно должна быть вершиной)
* R\_max — максимальный радиус отдаления от точки назначения, образует область, в которой пользователь готов припарковать машину

*Данные, использующиеся в работе алгоритма:*

* R\_start - начальный радиус поиска, по умолчанию всегда равен 100м;
* R\_increment — шаг увеличения радиуса, по умолчанию равен 100м;
* R\_i - текущий радиус поиска, равный (R\_start + R\_increment \* i);
* i - счетчик тактов работы алгоритма, с каждым тактом радиус поиска R\_i увеличивается на величину R\_increment;
* NodesArray - массив вершин, удаленных от DestinationCoordinate на расстояние, не превышающее R\_i;
* WaysArrays - массив массивов путей, образующих цикл[[7]](#_g6og2git8j76), состоит из последовательности вершин, по которым нужно двигаться, чтобы "кружить", также сохраняется длина пути L и максимальное расстояние MaxDistance между DestinationCoordinate и вершиной из Way\_i (худший случай парковки для данного цикла). WaysArrays = { Way\_1 = { 100м, A, B, C, A}, Way\_2{ 300м, A, C, D, B, A} }

*Программа выдаёт на выходе:*

* SotutionArray — массив вершин, последовательно перемещаясь по которым пользователь будет двигаться по оптимальному маршруту "кружения". (путь от StartNode до области кружения можно для упрощения не делать)\*
* SolutionLength — длина пути (сумма всех ребер между вершинами SolutionArray)

*Описание алгоритма:*

1. Пользователь задаёт DestinationCoordinate, R\_max, i = 0.

2. Найти все вершины, удаленные от DestinationCoordinate на расстояние не превышающее R\_i, записать их в массив NodeArray.

3. Проверяем, можно ли из вершин в NodeArray составить цикл[[7]](#_6z2pqz1em7tv), которого нет в WaysArrays.

3.1. Если да, то считаем длину пути этого цикла и вносим его в WaysArrays.

3.1.1. Повторить пункт 3.

3.2. Если таких циклов нет, то увеличиваем R\_i на R\_increment.

4. R\_i > R\_max?

4.1 Если нет, то возвращаемся к пункту 3

4.2 Если да, то все допустимые вершины уже просмотрены, переходим к составлению маршрута.

Исходим из того, что пользователь прежде всего хочет припарковать свою машину как можно ближе к DestinationCoordinate и циклы будут проходиться по мере возрастания значения MaxDistance (рассматриваем худший случай для этого цикла, т.к. лучшее значение часто бы повторялось и не давало бы нужной информации) для каждого Way\_i. Также предположим, что по каждому циклу мы хотим двигаться не более одного раза (повторное прохождение по уже пройденным ребрам разрешается, но кружение по одному циклу - нет).

Если массив WaysArrays пуст, то вывести сообщение с грустным смайлом, иначе:

5. Сортируем массив WaysArrays по параметру MaxDistance для каждого Way\_i.

6. Ищем кратчайший путь от StartNode до любой из вершины Way\_1, результат (порядок вершин, по которым нужно перемещаться) сохраняем в SolutionArray, длину пути соответственно в SolutionLength.

7. Добавляем в SolutionArray вершины цикла Way\_1, сумму ребер в SolutionLength.

8. Удаляем пройденный цикл из WaysArrays.

9. Проверяем, есть ли ещё циклы в WaysArrays?

9.1. Если нет, то замыкаем наш маршрут (ищем путь от последней вершины до первой заносим данные в SolutionArray и SolutionLength, чтобы в случае неудачи, пользователь пошёл на второй круг).

9.1.1. Выводим пользователю SolutionArray и SolutionLength.

9.2 Если циклы в WaysArrays ещё есть, то берем следующий цикл Way\_i и проверяем, является ли вершина, в которой мы сейчас находимся (последняя вершина в массиве SolutionArray), вершиной цикла Way\_i?

9.2.1 Если да, проверяем, является ли вершина, в которой мы находимся первой в цикле Way\_i?

9.2.1.1. Если да, то по очереди добавляем вершины из Way\_i в SolutionArray, а суммы ребер прибавляем к SolutionLength

9.2.1.2. Если вершина не является первой (К примеру, мы находимся в вершине B, считав цикл {A, B, C, A}), то необходимо перестроить цикл так, чтобы он начинался с той вершины, в которой мы находимся (в {B, C, A, C} ).

9.2.2. Если вершина, в которой мы находимся не является вершиной цикла Way\_i, то ищем кратчайший путь до любой из вершин цикла Way\_i (Issue #11), записываем его в SolutionArray, а сумму ребер соответственно в SolutionLength.

На этом шаге мы гарантировано находимся в вершине цикла Way\_i, поэтому возвращаемся к пункту 9.2.1.

# ЗАКЛЮЧЕНИЕ

В результате работы было разработано приложение, решающее проблему поиска оптимального маршрута парковки вокруг точки назначения.

Слабая связность компонентов приложения предполагает возможность кастомизации приложения (другие источники и механизмы хранения данных).

Эффективность по расходуемой памяти достигается за счет использования графовой базы данных и отказа от импортирования большого объема данных.

Недостатком работы является отсутствие возможности отменить процесс поиска. Построение маршрута выполняется в среднем за две минуты. Кроме того, приложение не покрыто тестами. Для улучшения решения необходимо протестировать приложение и добавить недостающие возможности. Исследование производительности и поиск способов её улучшения позволит улучшить взаимодействие пользователей с продуктом.

Дальнейшая работа будет направлена на оптимизацию поиска маршрута, на оценку скорости работы в зависимости от заданного радиуса поиска и размера базы. Для запуска приложения в опытную эксплуатацию необходимо разработать мобильное приложение.

# ПРИЛОЖЕНИЯ

## Сборка приложения

## Скачать проект из репозитория [[1](#_fxci40satinp)]

## Neo4j требует начальной инициализации, поэтому перед началом работы необходимо в браузере перейти по адресу localhost:7474 и ввести имя и пароль.

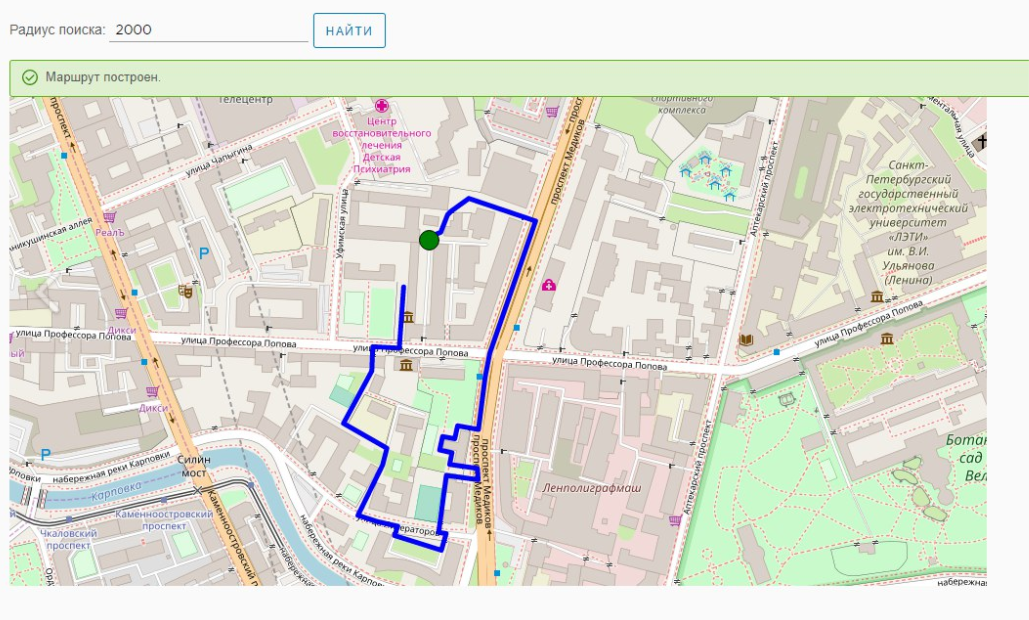
## Запустить серверную часть приложения на порту 8080.

## Запустить Angular c помощью команды ng serve.

После всех шагов приложение доступно по адресу localhost:4200

## 

## Снимки экрана приложения

****

*Рис.4 Демонстрация работы приложения*

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

#### Исходный код приложения на github.com: <https://github.com/moevm/nosql-2017-parking_around>

#### Документация Neo4j (<https://neo4j.com/docs/>)

#### Документация Angular (<https://angular.io/docs>)

#### Документация Spring Boot (<https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-documentation>)

#### Документация по ngx-openlayers (<https://github.com/quentin-ol/ngx-openlayers/tree/master/documentation>)

#### Описание формата координат WGS-84 (<http://spatialreference.org/ref/epsg/4326/>)

#### Цикл при поиске маршрута (<https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84-%D1%86%D0%B8%D0%BA%D0%BB>)