Lappeenranta University of Technology
School of Engineering Science
Degree Program in Intelligent Computing

Tikhon Belousko

SOFTWARE FOR TRANSPORT ACCESSIBILITY ANALYSIS: THE CASE OF MOSCOW

Examiners: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

Supervisors: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Intelligent Computing

Tikhon Belousko

Software for Transport Accessibility Analysis: The Case of Moscow

2016

29 pages, 2 thingy, 3 stuff.

Examiners: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

Keywords: web-based maps, GIS, transport accessibility, vector tiles

Abstract here...

PREFACE

I wish to thank my supervisors ...

Lappeenranta, April 19th, 2016

Tikhon Belousko

CONTENTS

1 INTRODUCTION

1.1 Background

In recent years, there have been many papers describing ...

1.2 Objectives and Delimitations

The aim of this paper is to generalize methods ...

1.3 Structure of the Report

The paper is divided into N main sections...

2 OVERVIEW

2.1 GIS Tools for Accessibility Analysis

2.2 Modern Web-based Maps

2.3 Data Formats

2.3.1 GeoJSON

2.3.2 MBTiles

2.3.3 Protocol Buffers

3 SOFTWARE REQUIREMENTS SPECIFICATION

The software development process starts with writing a software requirements specification (SRS) so that it serve as an agreement between client and contractor about the set of the features which need to be implemented. Another important purpose of this document is to create understanding of the features with higher and lower priority, thus developer would know what should be implemented in the first version of the product and what could be added later.

In this section the simplified version of the SRS will be presented which includes description of the user interface, operating environment, and set of system features with detailed explanation.

3.1 Graphical User Interface

The interface design was provided by Mathrioshka LLC, which includes set of different states of the application with small description provided. The interface should have been implemented to work in browser with use of HTML5 standard. The whole program should be implemented as single page application, which means that all changes in the state of the application happen without page reload. The interface have to be responsive and support screen sizes starting from 800×600 pixels.

The manipulation of the data within the application is performed utilizing main control elements. Most important elements are listed below:

1. Mode selector is needed for switching from general overview to location-wise analysis and located in the left bottom corner of the screen.

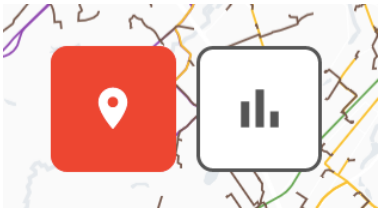

Figure 1. Mode selector.

2. Overview mode selector switches different types of overview analysis tools. For some views it has embedded slider used for filtering. The component is positioned on the right side of the screen.

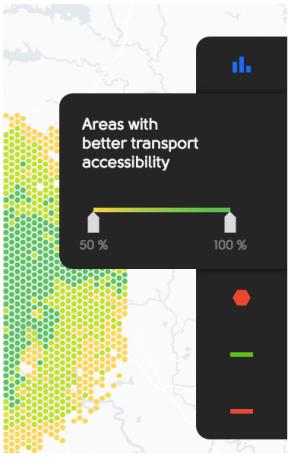

Figure 2. Overview mode selector.

3. Transport filter is a set of checkboxes where every checkbox represent certain type of transport. This component is placed on top of the screen.

Figure 3. Transport filter.

4. Current location indicator shows what point is selected on the map at this moment and also allows to clear current selection. The component is located on the right side of the window.

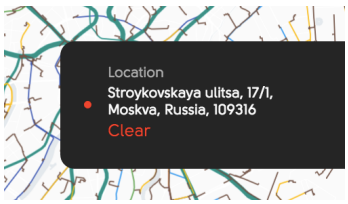

Figure 4. Current location indicator.

5. Map is the most important element of the interface which is presented on every screen of the application the only thing that changes is the content of the map. This component can show points, areas, routes and also provides functionality for selecting particular locations for analysis.

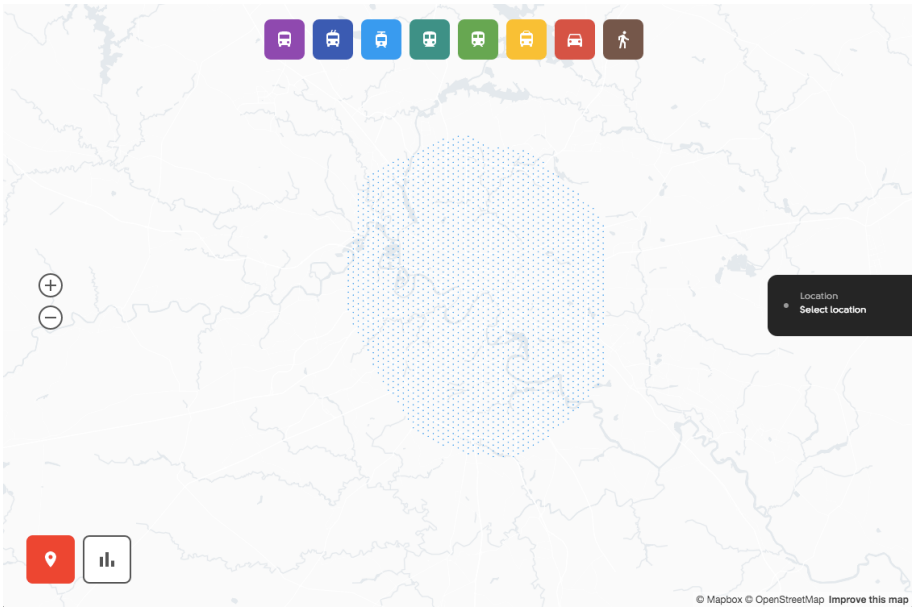

Figure 5. Map.

3.2 Operating Environment

The software is developed on x86 based computer using Mac OS X 10.11 operating system. The computer hardware is featured 8 GB 1600 MHz DDR3 RAM and 1.6 GHz Intel Core i5 processor. The client should operate on any system which is able to install browsers like Google Chrome 49.0, Safari, Firefox or Microsoft Edge. Hence it works well on Windows, Mac OS X and Linux. As for the server, it was tested to work on Ubuntu 14.04 and 512 Mb RAM. Running the server side on Windows was not tested.

3.3 System Features

Use case 1: Hello there this is the first use case... bla bla bla...

4 ARCHITECTURE

4.1 Client

The first significant part of the application is the client which will be the interface for the user to manipulate map and data. The core functionality of the client is follows:

1. Panning and zooming map.
2. Selecting a point on the map to build a whole graph of the routes needed to move from selected point to all other points of the city. All of the routes are parametrized by color and width. These parameters are calculated utilizing information about transport type specified in this point and the coefficient which is defined as number of times this particular line is used in all of the directions calculated from this point.
3. Showing prevailing transport in all points of the grid. Prevailing transport is defined by calculating total time spent in each transport type. The maximum of all of the values will be taken as prevailing.
4. Showing fastest and slowest roads. The map should also have interactive slider which will be used to filter roads within particular speed range.
5. Showing most accessible points on the grid. The transport accessibility coefficient of the point can be defined as total time spent in a transport while moving from selected point to all other points. All of the values after that scaled to be between 0 and 1. This view should also have a filter by transport accessibility coefficient.

For the reason of the easy distribution it was selected to use browser based client. Thus client is implemented as single-page web application, which should support all modern browsers such as Safari, Google Chrome, Firefox, Microsoft Edge.

4.2 RESTful Server

For serving data to the client it was selected to utilize RESTful [1] architectural style, which represents HTTP approach to read, update and delete data. Due to the fact, that our application will only read data and not modify it, then we will only need to describe all

of the end-points for data access where data will be encoded in GeoJSON [2]. For our application following end-points were selected:

- **`GET /points`**
  The returned points are described as FeatureCollection of points with following properties (see also Listing 1):

  name is a string;

  id is a unique identifier of the point encoded as string;

  prevalingTransport could be “BUS”, “TROLLEYBUS”, “TRAM”, “SUBWAY”, “COMMUTER_TRAIN”, “SHARE_TAXI”, “DRIVING”, “WALKING”.

  accessibility is a floating number in range from 0 to 1.

```
1   {
2     "type": "FeatureCollection",
3     "features": [
4       "type": "Feature",
5       "geometry": [0,0],
6       "properties": {
7         "id": 0,
8         "name": "Location Name",
9         "prevailingTransport": "SUBWAY",
10        "accessibility": 0.6
11      }
12    ]
13  }
```

Listing 1. Points response example

- **`GET /lines?point_id`**

  This endpoint is parametrized by point_id which means that if point id is presented then the client will receive only those lines which are associated with specified point. On the other hand, if no point id is not specified all lines are returned.

  The properties of the lines can be describe as follows (see also Listing 2):

  id is a unique identifier of the line encoded as string;

travelMode is on of the “BUS”, “TROLLEYBUS”, “TRAM”, “SUBWAY”, “COMMUTER_TRAIN”, “SHARE_TAXI”, “DRIVING”, or “WALKING”.

weight number of times this line was used in all of the routes across all of the directions’ set.

duration time needed to travel this line in seconds.

distance total length of this line in meters.

```
{
  "type": "FeatureCollection",
  "features": [
    "type": "Feature",
    "geometry": [[0,0], [0,0]],
    "properties": {
      "id": 0,
      "width": 1,
      "weight": 200
      "duration": 100,
      "distance": 400,
      "travelMode": "BUS"
    ]
  }
}
```

Listing 2. Lines response example

It is important to note that we could also add such filtering parameters as `speed` and `travelMode`, but it was revealed during set of experiments that filtering on server side can take significant amount of time which is up to few hundreds seconds. Hence, this parameters were dropped and filtering is performed on the client.

Suggested architecture implies that client will perform rendering using GeoJSON data, although it was investigated that for our purposes rendering becomes rather slow and after series of optimizations the best rendering time was close to 3 seconds. The next improvement to the current scheme will be adding tile server which can significantly speed up rendering time.

4.3 Tile Server

Graphical map tiles are usually rectangular images in raster or vector format. Most of the popular map library providers utilize tiles [3, 4] for rendering their maps. Raster tiles are basically images which do not allow to change color of roads and landscapes. In contrast, vector tiles are not just images but structures which contain geometries and metadata such as roads, rivers, places in special compact format. Vector tiles only rendered when requested by the client.

There are several benefits of using vector tiles. The first, advantage is small size of the tiles, which allows rendering of high resolution and caching. The second advantage is an ability to change style of the layers: adjust colors of the geometry, line width, set borders to polygons, set background patterns. Finally, maps based on vector tiles allow smooth transitions between zoom levels. Although, the cost of usage of the vector tiles is the lack of compatibility with older browsers. At the moment most of the libraries demand WebGL support and latest browser versions which are Chrome 49.0, Safari 9, Internet Explorer 10 and higher, Microsoft Edge 13 [5, 6]. Another drawback of usage tiles is that geometry can not be transformed in real-time and rendering small amounts of data is actually slower than GeoJSON approach.

Switching from GeoJSON server to vector-tile server resulted in considerable speed improvement. If previously rendering time took around 3 seconds and amount of transfered data was around 1.8 Mb, then with tile server rendering takes less than a second and size of the data transfered to the client is around 100 kb.

4.4 Summary

Although, tiles suit well for rendering considerable amounts data, for points rendering it was decided to utilize GeoJSON data. The final architecture is illustrated on Figure 6 and can be described as follows:

1. Web-based client supporting latest browsers.
2. GeoJSON RESTfull server providing points data.
3. Tile server providing data about lines.

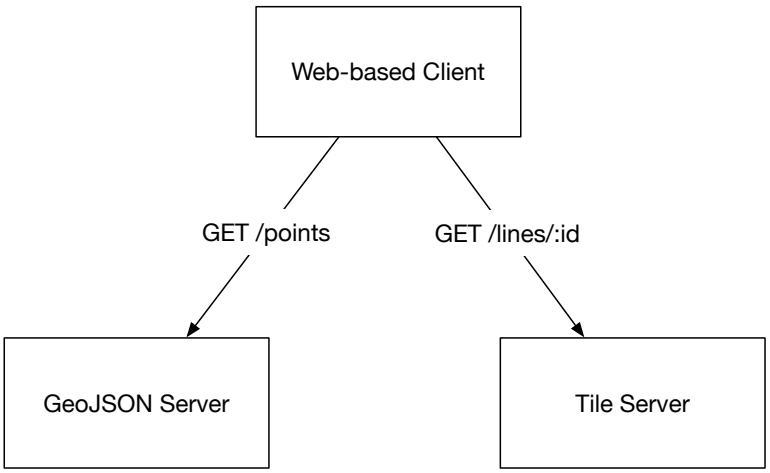

Figure 6. Final system architecture.

5 IMPLEMENTATION

5.1 Data Pre-processing

The work started by preparing the data so servers could work with it the most efficient way. This section is devoted to the description of the initial data and how it was processed. The section also involves comparison of deferent approaches of data structuring, their comparison and summarizing of the results.

5.1.1 Description of the Raw Data

The provided original data is collected using Google Directions API [7] applying following algorithm. First, on top of the city the grid of points was constructed using WGS 84 [8] coordinate system. Second, for every pair of points the directions were calculated using HTTP request to the API. Returned data is encoded in JSON format written to a text file. As result raw data is a text file where every line is a JSON object received from Google Directions API. The process of data collection is demonstrated on Figure 7.

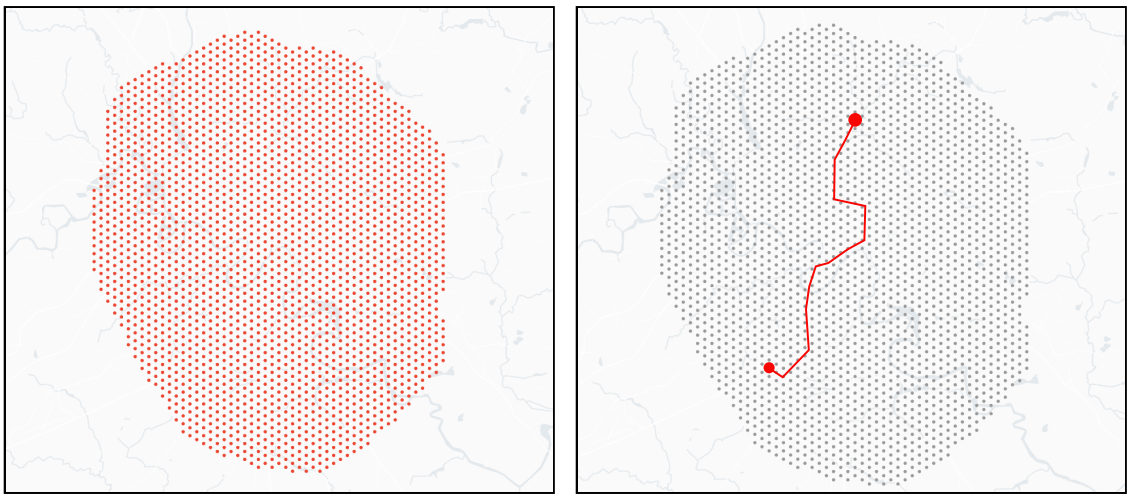

Figure 7. The process of the collecting data.

In fact there is also meta information such as object id, job status, waypoints order, warnings, but if we simplify and leave only important fields that we will need in pre-processing step, then JSON object could look like this:

```
{
  "start_lat": 55.726497,
  "start_long": 37.338183,
  "end_lat": 55.886619,
  "end_long": 37.579683,
  "data": {
    "routes": [{
      "legs": [{
        "distance": { "text": "45.1 km", "value": 45093 },
        "end_address": "Novgorodskaya ulitsa...",
        "start_address": "Razdorovskaya, Romashkovo...",
        "steps": [{
          "travel_mode": "DRIVING",
          "polyline": { "points": "wicsI{u{bFs@rDC" },
          "distance": { "text": "4.2 km", "value": 4166 },
          "duration": { "text": "7 mins", "value": 438 }
        }, ... ]
      }, ... ]
    }, ... ]
  }
}
```

Listing 3. Google Directions API simplified response example

As been be seen from simplified response example, the object contains coordinates of the start and end point, array routes which consists of `legs`. `Legs` are parts of the route between waypoints. Since in our queries we do not have middle waypoints, the routes array will have only one element in all cases. Each leg contains total distance information about start and end location. Also inside `legs` there is `steps`, each step is a part of route which can be described by single command and type of transport, for example “move forward by bus” is clearly a step. Step keeps information about its distance and duration which will be needed to complete this step. Other important things are `travel_mode` which indicates what type of transport is used in this step and `polyline` which is essentially a set points of points encoded using lossy Google’s algorithm which converts array of float numbers first to binary representation, then to decimal integers and finally to string using ASCII codes [9].

5.1.2 Extracting Points

The data collection was initiated before by Mathrioshka, thus the first task was extracting grid points from the raw data. This step was performed utilizing set script written in Python, which is reading file line by line and extracting starting location point from the data. The idea of the algorithm is presented in Listing 4. As can be seen from the listing, Geohash [10] standard was used to prevent repetitions of the points. Thus, coordinates could be mapped to strings which can be utilized as ids in points dictionary. In our implementation for encoding `python-geohash` library was used [11]. For storing, the result of the algorithm the pickle [12] format was selected, since it has quite simple interface and built in inside Python standard library.

```
1  points = []
2
3  for line in jsonfile:
4      json_obj = json.loads(line)
5      slat = json_obj['start_lat']
6      slng = json_obj['start_long']
7      point = {
8          'point_id': geohash.encode(slat, slng),
9          'lat': slat,
10         'lng': slng
11     }
12     points[point['point_id']] = point
```

Listing 4. Points extraction

5.1.3 Extracting Lines

Another important task was to extract lines from the file, but first to make experiments faster it was decided to convert data to more convenient format, hence it would be easier to experiment and process data more effectively. Once we have data extracted we will convert it to the set of GeoJSON files and after that generate vector tiles which will be served by our tile server.

For intermediate lines representation the CSV file format was selected, due to its simplicity and availability of the encoders and decoders inside standard Python library. The

algorithm is presented in Listing 5. First, we initialize table of lines which will contain all unique lines encoded in raw data. Second, we read file object by object. Each object is then decomposed into set of lines using `lines_from_json()` function. Once lines are extracted the assertion is performed to check whether the line was already met before. If the answer is no, then we initialize line, otherwise we just sum distance and duration and increase weight by one. The distance, duration and weight are necessary to compute line width, speed, prevailing transport and accessibility of the point.

```
1  # Table of lines
2  T = {}
3
4  with open(DATA_PATH) as jsonfile:
5
6      for json_str in jsonfile:
7          json_obj = json.loads(json_str)
8
9          for line in lines_from_json(json_obj):
10             line_hash = line['line_hash']
11
12             if line_hash not in T:
13                 del line['line_hash']
14                 T[line_hash] = line
15                 T[line_hash]['line_id'] = len(T)
16             else:
17                 T[line_hash]['weight'] += 1
18                 T[line_hash]['distance'] += line['distance']
19                 T[line_hash]['duration'] += line['duration']
```

Listing 5. Lines conversion.

5.1.4 Link Lines to Points

Since we have our data extracted now it is important to have references from lines to points. The first approach was to store in lines CSV file also point identifiers so in case we would need to get all lines for given points we would just go through the lines table an select only those lines which have ID of the point. It was revealed that this approach is quite slow since each line can be associated with up to 2000 points and it would take $O(N \times M)$ time complexity to get all lines where N is the size of the lines table and M

the length of the array points. The second approach was to store lines identifiers inside points table, which resulted in significant improvement and we could access all lines for $O(N)$. In combination with Pandas library [13], which is used for storing lines table in memory, method with storing lines in points table results in even more considerable speed improvement. On the other hand, the points dictionary grew in size and started to occupy more than 8 Gb of the disc space. To overcome this issue the shelve [14] data type was chosen. Shelve is a simple file-based database with object-like interface, which is fully compatible with pickle. Thus without serious code modifications the memory usage was dropped from 8 Gb to 200 Mb.

5.1.5 Transformation to Vector Tiles

Once the data is transformed to convenient form, the next step is convert it to vector tiles. For this purpose the tippecanoe [15] tool was selected which is developed by Mapbox team as open source project. The drawback of this instrument is that to generate tiles with several layers one would need to have separate GeoJSON file for each tile layer. In our application we will need 128 tile layers to render all weights, colors and also apply filtering. Thus, we first need to generate 128 GeoJSON files for each point. Then we could easily convert set of GeoJSON files to tiles.

5.1.6 Results

The resulting preprocessing scheme is demonstrated on Figure 8. First, we transform raw data to intermediate state which is shelve of points with links to lines and table of lines with properties. Next, for each point we generate a folder which contains GoeJSON file describing set of lines associated with given point (all directions from one point on the grid to others), then GeoJSON files are transformed to vector tiles. In course of the preprocessing there were other approaches, for instance when rendering was not utilizing vector tiles, but just GeoJSON files the was improvement which allowed merging adjacent lines together to reduce the size of the occupied space, but once the the implementation started to be based in MBTiles lines merging was rejected. It is important to note that initially the tiles generation was planned to implement in real time, but it was discovered that real-time processing takes significant amount of time which results in delays in server responses.

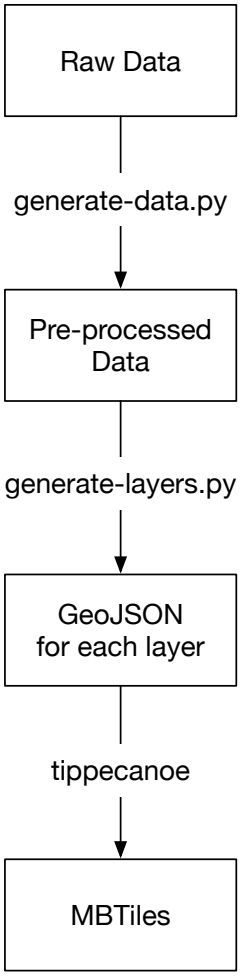

Figure 8. The process of the data transformation.

5.2 Web Interface

In this section the main architectural pattern used for building the interface is described including diagrams showing hierarchy of components, the selection of map drawing library is made.

5.2.1 FLUX Architecture

In the process of the development of the web interface the crucial thing is the state management. In 2010 the Backbone.js [16] library was introduced which was supposed to solve the problem of the state management often applying MVC [17] pattern. Although, on large projects there where a lot of cross-dependencies this paradigm resulted to be hard to maintain. Later in 2013 Facebook Inc. introduced component based React [18] library,

which allowed to think of the interface as function which maps state to representation. Although, React was implemented only as library for creating views and was not meant to solve problem of data state management in the application it still had significant success. In 2014 Facebook Inc. suggested FLUX [19] pattern which gained significant popularity. The idea behind FLUX can be described as follows: data comes from the store to the view which renders it, the user can fire an action, for example by pressing a button, the action is goes to the dispatcher which tells to all of the stores that certain action was fired. The stores modify the data and broadcast it to all of the views which are subscribed to this store. All of the views that received new data gets re-rendered. The illustration of this process can be seen on Figure 9.

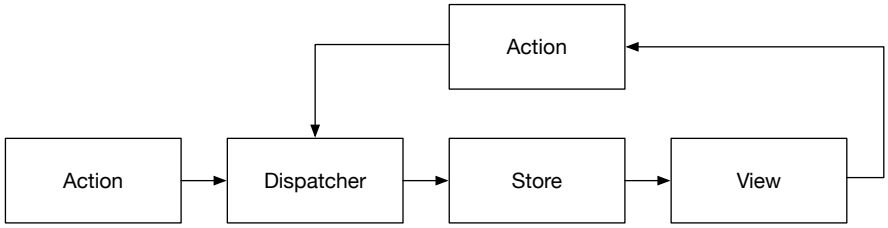

Figure 9. FLUX architecture.

Eventually FLUX evolved in library called Redux [20], which introduced few important modifications such as suppressing dispatcher, using pure functions for data modifications, using only single store where all of the data is accumulated, introducing middlewares for managing side effects. All this features create easier debugging experience and clear separation of concerns. Thus, for the development Redux and React libraries were chosen.

React components were organized accordingly to Figure 10 accordingly to the approach of presentational and container components [21]. The main presentational component is `App` which is the only one connected to the store directly. All other components are "dumb" and receive data as well as actions which are needed to be fired from `App`. The `SpeedMapButton` represents a switch for choosing between coloring lines by speed and by the type of transport. The `Map` components accumulates all of the logic related to drawing maps. The `TransportFilter` shows what kind of transport should be displayed on the map. The `LocationSelector` shows information about selected location. The `ModeSelector` switches between global overview and location information. The positioning of all of the components is illustrated by Figure 11.

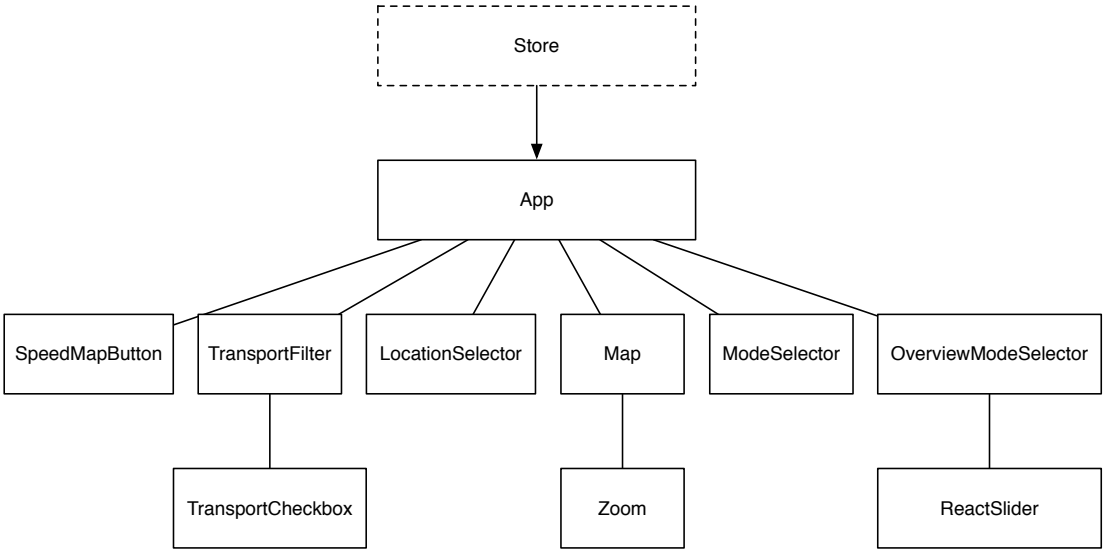

Figure 10. Structure of the components

5.2.2 Drawing Maps

As it was described before there are several the most famous libraries for drawing maps such as Yandex Mapx, Google Maps, Leaflet.js and Mapbox GL JS. To determine which library will suite better for our program, the needed functionality should be defined. The most important features are:

1. Panning and zooming.
2. Support vector tiles.
3. Rendering map from external data source.
4. Support of the vector tiles styling of such properties as line widths, line colors.
5. Ability to render icons at certain location.
6. Support for map styling.
7. Uses open standards.

Among all of the libraries the most suitable is Mapbox GL JS which supports all of the listed features. Although some part of the free Mapbox toolkit has limitations, for example free subscription plan allows not more than 50000 map views per month, but for

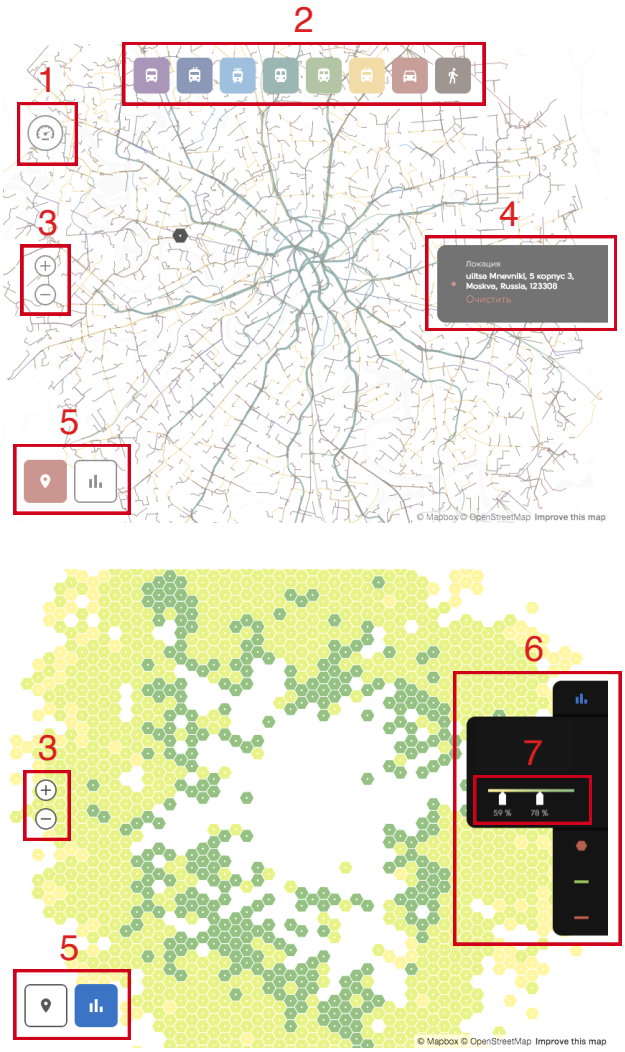

Figure 11. Components positioning. 1 – SpeedMapButton; 2 – TransportFilter; 3 – Zoom; 4 – LocationSelector; 5 – ModeSelector; 6 – OverviewModeSelector; 7 – ReactSlider.

our project that was enough. Important to note that Mapbox uses OpenStreeMap for geographical data provider which is free to use. As for Google Maps and Yandex Maps, they were not supporting maps from external data sources which is needed for rendering custom vector tiles. Although Leaflet is a very popular choice for map drawing, it is not suitable for our particular application since it does not support vector tiles rendering.

5.3 Tile Server

For rendering all of the directions the tile server would need to be chosen. The tasks in our application for this component are quite trivial:

1. Fast.
2. Support of the MBTiles format.
3. Easy to setup and configure.
4. Free and open-source.

The great candidate for this role was Tessera tile server [22] which is written in Node.js and based on Tilelive interface (developed by Mapbox). Moreover, Tessera is really easy to install using npm package manager. The configuration file should be written in JSON format and can be easily generated from our data. The only needed parameters are source path to the data and the URL on which particular vector tiles will be available.

6 INTEGRATION

Development of the software is very related to the problems of the distribution and setting up development environment. In this section the methods which were used for building client code will be described. Another section will be devoted to the deploying process.

6.1 Development Environment

The tools on the client side are developed with enormous speed. If previously the

6.2 Building a Project

6.3 Deploying a Project

7 CONCLUSION

REFERENCES

[1] Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2016. [Online; accessed 19-April-2016].

[2] Allan Doyle Sean Gillies Tim Schaub Christopher Schmidt Howard Butler, Martin Daly. The geojson format specification. http://geojson.org/geojson-spec.html, 2008. [Online; accessed 19-April-2016].

[3] Map types: Tile coordinates. https://developers.google.com/maps/documentation/javascript/maptypes?hl=en#TileCoordinates, 2016. [Online; accessed 19-April-2016].

[4] Vector tiles. https://www.mapbox.com/vector-tiles/. [Online; accessed 19-April-2016].

[5] Google maps javascript api: Browser support. https://developers.google.com/maps/documentation/javascript/browsersupport, 2016. [Online; accessed 20-April-2016].

[6] Troubleshooting: Browser support. https://www.mapbox.com/help/mapbox-browser-support. [Online; accessed 20-April-2016].

[7] Google maps directions api. https://developers.google.com/maps/documentation/directions/, 2016. [Online; accessed 20-April-2016].

[8] World geodetic system. https://en.wikipedia.org/wiki/World_Geodetic_System, 2016. [Online; accessed 20-April-2016].

[9] Google maps directions api: Encoded polyline algorithm format. https://developers.google.com/maps/documentation/utilities/polylinealgorithm, 2016. [Online; accessed 20-April-2016].

[10] Geohash. https://en.wikipedia.org/wiki/Geohash, 2016. [Online; accessed 21-April-2016].

[11] python-geohash 0.8.5. https://pypi.python.org/pypi/python-geohash/0.8.5. [Online; accessed 21-April-2016].

[12] 12.1. pickle — python object serialization. https://docs.python.org/3/library/pickle.html. [Online; accessed 25-April-2016].

[13] Python data analysis library. http://pandas.pydata.org/. [Online; accessed 21-April-2016].

[14] 12.3. shelve — python object persistence. https://docs.python.org/3/library/shelve.html. [Online; accessed 25-April-2016].

[15] Build vector tilesets from large collections of geojson features. https://github.com/mapbox/tippecanoe. [Online; accessed 25-April-2016].

[16] Backbone.js. http://backbonejs.org/. [Online; accessed 25-April-2016].

[17] Client-side mvc with backbone.js. http://www.slideshare.net/iloveigloo/clientside-mvc-with-backbonejs. [Online; accessed 25-April-2016].

[18] React: A javascript library for building user interfaces. https://facebook.github.io/react/index.html. [Online; accessed 25-April-2016].

[19] Flux: Application architecture for building user interfaces. https://facebook.github.io/flux/. [Online; accessed 25-April-2016].

[20] Predictable state container for javascript apps. https://github.com/reactjs/redux. [Online; accessed 25-April-2016].

[21] Dan Abramov. Presentational and container components. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.2vyf9ikcr, 2015. [Online; accessed 26-April-2016].

[22] Tessera: A tilelive-based tile server. https://github.com/mojodna/tessera. [Online; accessed 26-April-2016].