# Technical Documentation

**Project**

**Activity based route planner**

In fulfillment of

the course

Agile web development

by

Oliver Schnieders

Felipe Lopez

March 2011

## Inhaltsverzeichnis

# Introduction

This document gives an overview of the application ActivityRoutePlanner.

It describes the application, the layout, the functionality of the controllers and model classes and how it fits together.

Finally it gives an overview of the database's structure and how is it possible to use OpenStreetMap[1] (OSM) Data for routing.

## Overview

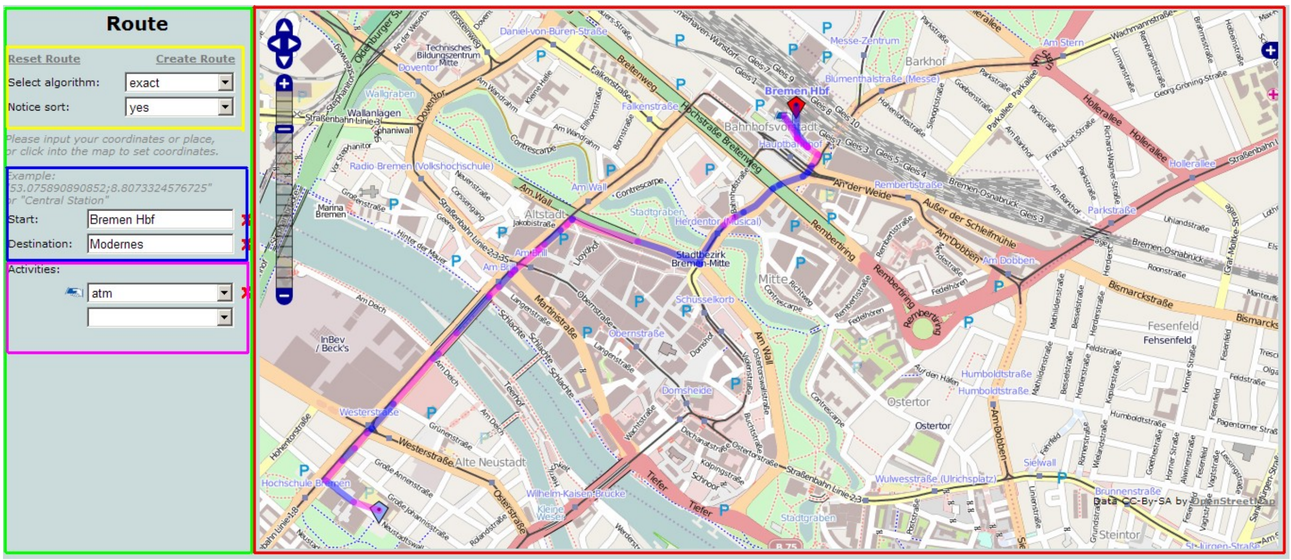
The application is an activity based route planner.

The user sets a start and an end point. Furthermore the user can add activities to a list. The planner searches points where the user can do this activities and returns a route from start to end point passing by the founded points.

The application has been developed with Ruby on Rails[2].

For the routing task the application uses PostgreSQL[3] 8.4, PostGIS[4], osm2po[5] and pgRouting[6].

To display the map, points and routes it uses OpenLayers[7], Yahoo! User Interface Library[8] and JavaScript.



Route

Reset Route Create Route

Select algorithm: exact

Notice sort: yes

Please input your coordinates or place, or click into the map to set coordinates.

Example: "53.075890890852;8.8073324576725" or "Central Station"

Start: Bremen Hbf

Destination: Modernes

Activities:

atm

*Figure 1: Screenshot Application*

Figure 1 shows the different areas of the application. They are included in the file init/index.html.erb.

In the following section is described which area is rendered in which section:

- Red: Content. View: init/_content. The map will be loaded with javascript.
- Green: Menu. View: init/_menu
- Yellow: Algorithm. View: generate_place/_algorithmus_form
- Blue: Points. View: init/_form
- Purple: Activities. View: activity/_index

The views will be described in the next chapter.

## Views

In this application each view has his own task. The following list gives an overview of all views.

- activity
  - _form.html.erb
  - _index.htmlerb
  - updateActivity.jsp.erb
- calculate_route
  - calculate_route.jsp.erb
- generate_place
  - _algorithmus_form.jsp.erb
  - set_algorithmus.jsp.erb
  - update_place.js.erb
- init
  - _content.html.erb
  - _form.html.erb
  - _header.html.erb
  - _menu.html.erb
  - index.html.erb
- layouts
  - application.html.erb
- place
  - _place_form.html.erb

Each *.jsp.erb file will handle the action for each Ajax-Call after the controller has finished his tasks. Each *.jsp.erb file will finally render the menu.

### Activity

At this views, the user can add, update or delete an activity. The view index.html.erb parses for each selected activity a select box, where the user can choose another activity or delete that activity. The user can only choose an activity once. Every time the _form.html.erb is called, the view will get the possible activity list from the activity class.

The icon of an activity will be searched at the Hash Image_URL, which is defined in class global.rb, with the Key TAG$VALUE.

Finally an empty select box (activity) will be added.

### *Calculate Route*

In this view the controller will find the shortest path and the Ajax-Callback will show this at the map.

### *Generate Place*

In this view the user is able to select the route algorithm and is able to select, if the algorithm has to notice sorting of activities.

Finally, the JS file update_places.js.erb will handle the Ajax-Callback after setting, updating or deleting start- or end point.

### *Init*

This view builds the application layout. Index.html.erb sets the views menu.html.erb and content.html.erb. Menu will sets the form for selecting algorithm and activities, content sets the map.

### *Layouts*

The view application.html.erb sets the header of the HTML page.

### *Place*

The view _place_form.html.erb renders the form for a point (start or end point).

## Controllers

Controllers handle the user inputs and verifies the data. In the following listing you can see all controllers in this application:

- activity_controller.rb
- calculate_route_controller.rb
- generate_place_controller.rb
- init_controller.rb

### *Activity Controller*

The activity controller handle the action to set a new activity, update a activity or delete an activity. Finally the controller will execute the updateActivity.jsp.erb file.

### *Calculate Route Controller*

The calculate route controller handles the routes finding. The controller writes the route nin an kml file with name “kmlRoute_”SESSIONID.kml and will be stored at the “public” directory. At the end of his tasks, the controller will execute the calculate_route.jsp.erb file.

### *Generate Place Controller*

This controller handles the user input to set, update or delete a point (start- or end point). Finally, the controller will execute the update_place.js.erb file after setting, updating or deleting a point, or execute the set_algorithmus.jsp.erb file after setting the algorithm or sorting of activities.

If the user has not set start- and end point, it will not be able to select activities

### *Init Controller*

The init controller is called at the start of the application and sets necessary objects.

## Classes

In the following listing you can see all classes of this application.

- activity.rb
- geo_result.rb
- global.rb
- point.rb
- route_generator.rb
- route.rb

All classes are stored in the directory “models”.

### *Activity*

The activity class got the attributes tag, value and result. Result is the nearest point(start- or end int) to that activity.

### *Geo Result*

This class represents an route result and stores them.

### *Global*

This class saves all global attributes. The attributes are valid all over the application.

### *Point*

This class represents an point (start- or end point) and controls the database attributes to get points.

### *Route Generator*

This class generate has SQL-Statements for searching the route in the database.

### *Route*

This class represents the route self.

## JavaScript

The application uses two custom JavaScript Files, util.js and yuiUtils.js. Utils.js is for setting, updating and removing markers and route from the map, yuiUtils.js is for loading wall for user inputs.

### *Util.js*

This script uses some global variables. In the following listing you can see all global variables

- map: The OpenStreetMap object.
- zoom: The default zoom at the start of the application.
- markerHash: An Hash to save markers. The key if start for start point, end for end point and for activities the icon name.
- Route: The route layer, which is shown after creating route.
- LayerMarkers; The marker to show activities and points.

Methods:

The script got the following methods:

- loadMap(): Loads the map at the start of the application and calls init().
- handleMapClick(event): Sets the context menu for clicking into the map.
- loadRoute(fileName): Loads a route and show them at the map.

- addMark(name,lat,lon,type): Adds a mark at the marker layer at the map and saves the marker in layerMarkers. If the marker not exists, createMarker() will be called.
- removeMark(id): Removes a mark from the map and from layerMarkers.
- addActivityMark(name,lat,lon,imagePath,id): Add a activity at the map. If the mark already exists, the mark will be updated at the map.
- createMarker(tooltip,lon,lat,src): Create a marker at the map.
- updateMarker(name,marker,lon,lat,src): Updates an marker at the map.
- init(lat,lon,zoom): Init will be called form loadMap() and sets necessary objects.
- RemoveMarks(): Removes all marks from the map and calls removeRoute().
- RemoveRoute(): Removes the route layer from the map.
- TestInit(): testInit() defines necessary objects for testing the JavaScript code with Jasmine.

### *YuiUtils.js*

This script got following methods:

- initLoadingPanel(): Inits the loading layer for user input.
- ShowWall(): Shows the wall while the system is busy after a user input.
- HideWall(): Hides the wall after the system has finished his job after user input.
- ShowSetPointMenu(): Shows the context menu after clicking into map for selecting points directly in the map.
- HideSetPointMenu(): Hides the context menu after clicking into the map.

## Database

The used database is a PostgreSQL 8.4 with PostGIS and pgRouting extension.

### *Data*

This application uses two types of data. Data to find activities and data to create routes.

The activity data is usual OSM data. It was imported with osm2pgsql. This data does not have the structure to calculate routes. It has information to query geometries for the desired activities.

The routing data come also from OSM but needs to be imported in a different way. The next section describe how to do it.

**In the /doc directory in repository is a backup from the used database!**

### *Tutorial import OSM data for routing*

This section describes how to install a database and import OSM data for routing purposes.

1. Install PostgreSQL 8.4. (pgRouting works only for this version)
2. Install PostGIS
3. Install pgRouting (For windows see http://www.davidgis.fr/documentation/pgrouting-1.02/)
4. Create a database with a postgis template
5. Execute scripts in the database from {pgRoutingDir}/Shared/Contrib in this order:

   routing_core.sql, routing_core_wrappers.sql, routing_topology.sql

   This scripts add functions to the database to use the pgRouting library!

6. Prepare data for routing with osm2po. See http://osm2po.de/download.php?dl=osm2po-free.pdf Page 5.

   Short:

   ```
   java –Xmx256m –jar osm2po.jar bremen.osm prefix=hb
   ```

   This will create a 'hb' directory in the current directory with an SQL script and some other files.

   Execute the script in the database. After executing the script the shell gives 2 commands like:

   ```
   psql -U postgres -q -1 -c "SELECT DropGeometryTable('hb_topo');" -d [dbname]
   psql -U postgres -q -1 -f "{WorkDir}\hb\hb_p3_topo_0.sql" -d [dbname]
   ```

   Execute this 2 commands.

7. After this steps the database has a table with a topology for routing 'hb_topo'.

   PgRouting requests a certain structure with column names to use the algorithms. To adjust the database its needed to create some tables and copy some data.
   Add the following columns with the following values from the 'hb_topo' table:

| New Column name | Data column name (hb_topo table) | Type |
|---|---|---|
| gid | id | Integer |
| the_geom | geom_way | Geometry |
| length | length(geom_way) | Double precision |
| Name | osm_name | Varchar |

   DONE!

   After this is it possible to execute a query like:

```
SELECT rt.gid, asKML(rt.the_geom) AS geojson,length(rt.the_geom) AS length, hb_topo.gid
FROM hb_topo, (SELECT gid, the_geom FROM dijkstra_sp('hb_topo',5841,3526)) as rt
WHERE hb_topo.gid=rt.gid;
```

   It finds the route from the vertice with the ID 5842 to the vertice with the ID 3526!

1 http://www.openstreetmap.de/
2 http://www.rubyonrails.org/
3 http://www.postgresql.de/
4 http://postgis.refractions.net/
5 http://osm2po.de/
6 http://www.pgrouting.org/
7 http://openlayers.org/
8 http://developer.yahoo.com/yui/