Customizable Navigation on Mobile Devices for Physically Impaired People

Christoph Boßmann and Bettina Harriehausen-Mühlbauer
Dept. of Computer Science, Univ. of Applied Sciences, Schöfferstraße 8B, 64295 Darmstadt, Germany

Keywords: WheelScout, Wheelchair, Navigation, Mobile, Barrier-free, Impaired, Mobility-impaired, Routing, Outdoor, Customization, Profile.

Abstract: Physical disability means for affected persons a substantial restriction in their everyday life. A significant number of people throughout Germany were physically disabled, a not inconsiderable proportion of whom are in wheelchairs. Current market-relevant navigational systems such as Google Maps still do not provide profiles for mobility-impaired people or people in wheelchairs. With *WheelScout* it aims to create a navigational system for mobile devices that enables these people to make their way alone, meaning without being routed across barriers such as stairways or uneven surfaces. In this paper, it is shown how such a system can be developed and how its essential features, including the ability of individual customization for its users, can be realized.

1 INTRODUCTION

In 2019, more than 4.6 million people are physically disabled across Germany (Destatis Statistisches Bundesamt, 2020). Among, approximately 1.4 million people are in wheelchairs. This corresponds to 17.7% of the severely disabled and 1.68% of the total population in Germany (nullbarriere.de, nd).

In article 9 of the *United Nations Convention on the Rights of Persons with Disabilities*, the declaration states that the member states shall take appropriate measures to ensure that persons with disabilities have access to the physical environment and transportation as well as facilities and services that are open or provided to the public (United Nations, 2006). That about half of the more than 1700 reported barriers on the map called *Weg mit den Barrieren*, in English *Get rid of the barriers*, is caused by lack of mobility, makes an otherwise sobering impression concerning those claims (Sozialverband VdK Deutschland e.V., 2016).

Further, article 20 states that member nations shall take adequate measures to ensure personal mobility with the highest possible independence for persons with disabilities. That also includes facilitation of new technologies (United Nations, 2006). If disabled persons are familiar with their surroundings, they can benefit from structural changes towards accessibility. However, in many cases, people want to be able to be mobile independently, even in unfamiliar areas. Although using a conventional navigational system such as *Google Maps* is convenient for citizens without any impairment, it might be somewhat frustrating for individuals with physical impairments, as those apps do not yet provide profiles, designated for people in wheelchairs.

Therefore, with *WheelScout* it was the aim to create an alternative to conventional navigational systems in terms of the accessibility of route for physically impaired people. Thus, in a way, the development of a navigational system for physically impaired people seeks to take the independence of personal mobility by assistive technologies into account which is claimed in article 20 of the *United Nations Convention on the Rights of Persons with Disabilities* mentioned above.

In the development of this project, Darmstadt University of Applied Sciences is working closely with the Independent Association of Civil Invalids (LAPIC-UVZ Onlus: Independent Association of Civil Invalids) in Bolzano, which is providing advisory as well as financial support for the development of this navigational system named *WheelScout*.

Boßmann, C. and Harriehausen-Mühlbauer, B.
Customizable Navigation on Mobile Devices for Physically Impaired People.
DOI: 10.5220/0010527002440255
In *Proceedings of the 7th International Conference on Information and Communication Technologies for Ageing Well and e-Health (ICT4AWE 2021)*, pages 244-255
ISBN: 978-989-758-506-7

2 RELATED WORK

In this section related products as well as useful components involved in the development of this navigational system are introduced and described.

2.1 OpenStreetMap

Fundamental for the realization of a navigation system is a suitable map material. OpenStreetMap (OSM) has proven to be a suitable source for the purpose. It is a free project that collects and structures geodata and makes them available for use by anyone for any purpose (Open Data). The map material is created by its diverse, passionate, and every day growing community (Community Driven) (OpenStreetMap, 2021).

Besides that, the major advantage of this particular purpose is a large amount of data about potential barriers such as stairways or uneven surfaces. The share of the most popular surface types in OSM worldwide among highways is illustrated in figure 1. Highways are any type of path, road, or street in OSM (OpenStreetMap contributors, 2021).

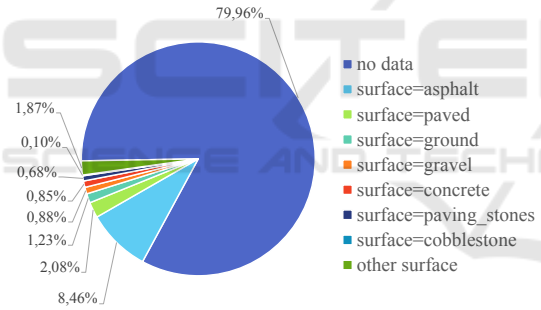

Figure 1: *Own visualization of data from https://taginfo.openstreetmap.org/*: Proportion of surface information among highways worldwide on OSM.

It is noted that for a proportion of about twenty percent details about the surface type are entered in OSM. Comparing figure 2 and figure 3, it can be further seen that in Darmstadt, Germany, for example, many stairways are tagged but further details, such as the number of steps or the presence of a handrail, are not entered in most cases.

In conclusion, even taking into account the fact that the information on roads and trails is far from complete, OSM data can still be used as a source of data regarding barrier information.

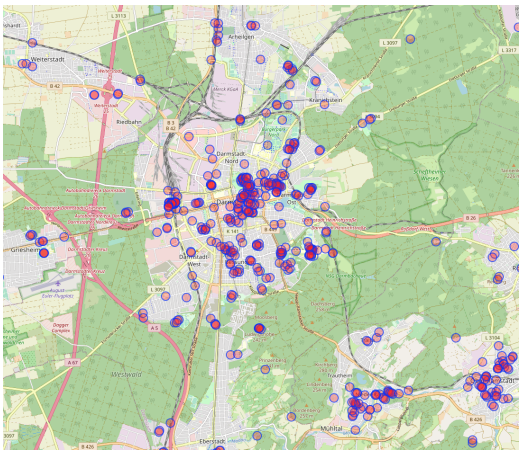

Figure 2: *Overpass Turbo*: Stairways in Darmstadt and surroundings.

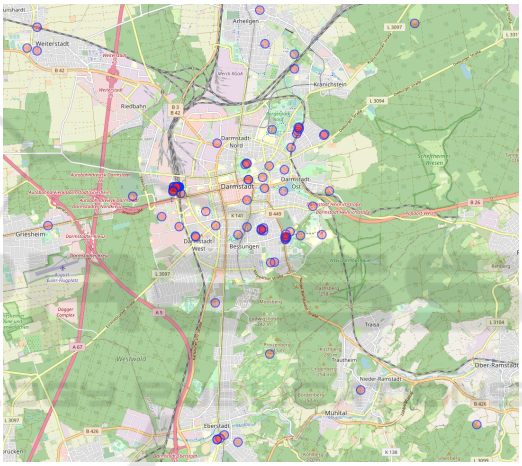

Figure 3: *Overpass Turbo*: Stairways in Darmstadt and surroundings tagged either with handrail or step count information.

2.2 GraphHopper

Further, it is necessary for the realization of a navigation system to use a suitable routing engine in order to process routing requests accordingly. With regard to the realization of a navigation system for people with limited mobility, it is advisable to use an engine that works with OSM data to take advantage from its features described in section 2.1.

GraphHopper is a fast and memory-efficient routing engine programmed in Java that uses OSM data to compute routes, which makes the routing engine platform independently usable. It is licensed with Apache license so it can be used modified and embedded into other applications free of charge. It supports several routing algorithms such as Dijkstra and A* as well as the integration of elevation providers to consider inclines during route computation. A major advantage

of GraphHopper is its customizability concerning the creation of different profiles for instance to represent different vehicles as different profiles in GraphHopper (Karich, 2019b)(Karich, 2019a). For example, a profile can be created that corresponds to the pedestrian profile but additionally avoids all stairways that are tagged as such in OSM.

2.3 Open Route Service

Open Route service, developed and provided by Heidelberg Institute for Geoinformation Technology (HeiGIT), offers routing services by using user-generated, collaboratively collected free geographic data from OSM (Heidelberg Institute for Geoinformation Technology, 2019). The platform provides the function of routing, based on profiles, including a wheelchair profile, also through its newly developed mobile application. This wheelchair profile is customizable by its users, directly on the platform. For instance, it can be customized to avoid cobblestone or flattered cobblestone as ground surface during routing. Parameters such as the width or the incline of a path can be customized as well. Furthermore, it is possible to examine the percentage of a path consisting of a particular type of surface. However it does currently not support adding new barriers to the system which is a major requirement four the work, see section 3, and the profile settings are not fully customizable (Google, 2021b).

2.4 Wheelmap and Accessibility.cloud

Wheelmap is a charity-based navigational service available to the public for reading requests. The map helps to find wheelchair-accessible places similar to points of interest on Google Maps. Lots of places such as grocery stores, shops, hotels, bars, restaurants, pharmacies, public parking garages, and many more are listed in four different categories depending on whether the place is entirely, partially, or not at all accessible or if the accessibility is unknown (Krauthausen, 2010). While *Wheelmap* itself serves as the frontend, the API named *accessibility.cloud* is both backend and data source of it. The accessibility information from this API is provided to the public to be used in other applications as well. For scientific purposes, its usage is free of charge. Both projects, *Wheelmap* and *accessibility.cloud* are initiatives of the non-profit organization SOZIALHELDEN e.V. (accessibility.cloud, 2021).

Even though it is shown whether the place itself is wheelchair accessible or not, the map does not show if a wheelchair can handle the way to the place. Although it hence cannot be used for routing directly, it can help to find barrier-free points of interest such as restrooms or stores that the user can navigate to.

2.5 Google Maps

The most popular navigation service, at least in the western world, is *Google Maps*. Today, it is an indispensable companion for almost anyone in daily life (Poleshova, 2020). Although its usage is convenient for vehicle drivers, cyclists, or just pedestrians since suitable profile exists for each of those different ways of transport, no such profile is available for wheelchair users. This is a big limitation, considering that stairways, uneven surfaces, and other insuperable barriers may require users to find alternative ways around them on their own. Even though *Google* introduced a feature to find wheelchair accessible routes, the service is currently only available in six cities, and none of them is in Germany (Akasaka, 2020). Bringing this together with the fact, that their API usage is not free of charge (Google, 2021a), it is currently neither utilizable as a source for map material nor as a navigational system for physically limited people itself.

3 METHODOLOGY

In this section, the feature requirements, the strategy for the development process, and its resulting design and architecture is explained.

3.1 Requirement Analysis

Since the navigation system is being developed in collaboration with this LAPIC association, mentioned in section 1, the desired features were evaluated primarily through surveys with mobility-impaired individuals from the association. Based on the result of the surveys the following core features were elaborated:

- Routing under consideration of barriers such as stairways, uneven surfaces, bottlenecks, and inclines.
- Creation of individual profiles to take into account the physical limitations of the user in the routing process accordingly.
- Adding new barriers or modification of existing barriers to the system by its users. This includes as well temporarily existing barriers such as fallen trees or road constructions.

Since it turned out, as described in section 2.1, that the database of OSM already contains a lot of relevant

data for finding an accessible route, the integration of the data of OSM is also a central requirement.

3.2 Design and Architecture

In this section, the design and architecture of *WheelScout* is described. The basis for the design process of the navigation system for *WheelScout* is the chosen routing engine GraphHopper. Therefore, all design, architectural and technical decisions must fit the technical design and properties of GraphHopper. It is important to point out that due to newly gained knowledge concerning the technical possibilities of GraphHopper, the design of *WheelScout* has changed multiple times. Corresponding designs were always implemented to test the technical possibilities of GraphHopper and the functionality accordingly.

3.2.1 First Design Approaches

It is important to point out here, that the underlying routing algorithms of GraphHopper are graph-based, meaning that all settings regarding routing behavior must be set before the graph is rendered. That means that whether a way or path should be considered as a barrier and thus avoided or not must be decided in advance to the rendering of the graph, see section 4.1.1. The first design based on this technical limitation is described below:

- The routing process takes place on the server-side.
- The requirement for complete individual profiles is waived, instead, various fixed profiles are created among which the user can choose.
- Changes to barriers are stored in a database.
- Periodically, the graph is re-rendered to apply the updated barriers to the graph so that they are taken into account in future routing processes.

The disadvantage is, however, that on the server-side, no fine-granular settings are possible, e.g. an exact setting of the maximum number of manageable steps on stairways depending on the existing handrail. Also, the graph must be re-rendered each time barriers are added, which must be considered a major disadvantage with a rendering time of about one hour (Germany).

Another design approach is based on the possibility of setting the accessibility of edges after rendering the graph which is possible through the GraphHopper high-level API. The design for this approach is described below:

- The routing process takes place on the client-side.
- Barrier information are again stored in the server-side database. Pre-rendered graphs are stored on the server-side as well.
- The requirement of complete individual profiles can be full-filled since the client stores the graph on the client. If a user updates his or her profile, the particular barrier information is fetched from the database and applied to the graph locally.

A major disadvantage of this approach is that it requires GraphHopper to run locally on the client which limits its usage on platforms where Java can be executed. That is why worthwhile development would be limited to Android as it is the only market relevant operating system for mobile devices which allows to run Java code.

Therefore, a third and final design approach has been developed where the routing engine has been moved back to the server-side. To take advantage of the 1st and 2nd approaches, i.e. server-side platform-independent route computation together with enabling fine-granular profile adaptation, the design was adjusted again which is described in section 3.2.2.

3.2.2 Final Design

Fundamental for the final design approach is that GraphHopper allows setting accessibility of certain edges of a graph to either true, meaning accessible, or false, meaning not accessible, as already described in section 3.2.1. Accordingly, in advance of each routing request, only the accessibility of those barriers that do not fit the user's physical limitation, e.g., its profile, is set to false. In figure 4 and figure 5 a minimal example visualizes the edges taking normal route and taking a longer route if a certain edge is not accessible e.g. there is a barrier.

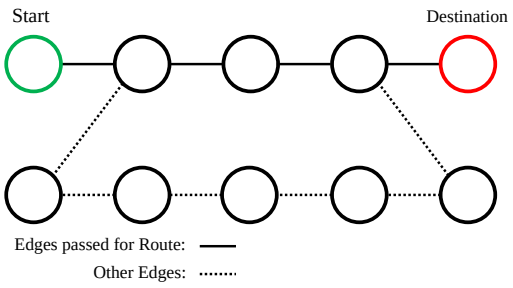

Figure 4: Normal route.

A further issue that needs to be addressed is that updating the accessibility can take multiple seconds depending on the number of edges, e.g., barriers. Furthermore, during this process, the GraphHopper instance cannot compute any other route because the

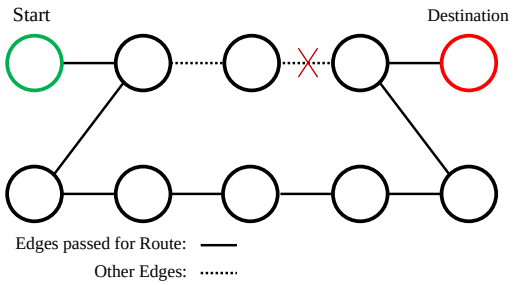

Figure 5: Route if certain edge is not accessible.

updated accessibility would affect all routing requests processed by GraphHopper on the same graph. That is why the accessibility should only be updated for those barriers that are situated in the surroundings of the actual route. Since the exact route is not known before its computation, squares alongside the linear distance with an appropriate buffer are calculated as displayed in figure 6.

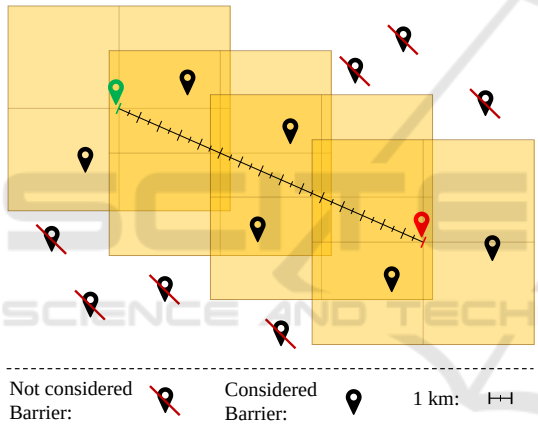

Figure 6: Calculated squares along-side of airline between start and destination of route.

To speed up the process, only barriers within these squares are considered for the route computation. Considering that limited people often find their ways in urban areas, a significant route deviation from the airline was assessed as very unlikely and negligible, based on numerous example routes. If such a doubtful case would occur, it is still possible to inform the user about the impossibility of computing the route. A further approach is to pre-compute the regular route for pedestrians to situate the squares as mentioned above alongside this route to cover a route that deviates too much from the airline. However, this approach has not yet been pursued further. Instead, the mentioned airline approach has been chosen.

3.2.3 Architecture

Under consideration of the feasibility study, described in section 4.2, the in figure 7 visualized architecture for *WheelScout* has been chosen. The architecture consist of a mobile client as frontend that communicates with an application kernel facade on the backend.

This system consists of several clusters on the backend side, where each cluster is responsible for a different area of the offered map material. An evident subdivision is, for instance, the division by countries. Accordingly, a cluster corresponds to a single country. The reason for preferring this approach is the in section 3.2.2 already mentioned computation time of a route. If a cluster's load is expected to be higher, a cluster can also be equipped with more than one routing engine to enable parallel computation of more than one routing request.

Further details concerning the implementation of processing routing requests are outlined in section 4.

3.3 Testing

To verify the functionality of the system's logic, regarding its correct computation of the route with corresponding profile settings, the developed mobile client is used by analyzing various computed routes for various profile settings. However, to verify the accuracy of computed routes without yet an existing client, a tool named *umap* was used. *umap* enables the visual representation of geographic information in the GeoJSON format (OpenStreetMap contributors, 2019). The results of computed routes are presented and discussed in section 5.

4 IMPLEMENTATION

In this section, the implementation of *WheelScout* is outlined. First of all, the required adjustments to the chosen routing engine GraphHopper are explained in order to realize the in section 3.1 mentioned requirements of custom profiles. Then its integration into a web server and the database for storing the barrier data will be explained.

4.1 Routing Engine

GraphHopper and thus also OSM are set as components, due to their advantages pointed out in sections 2.1 and 2.2. As the underlying algorithm of GraphHopper for computation of routes is graph-based, the

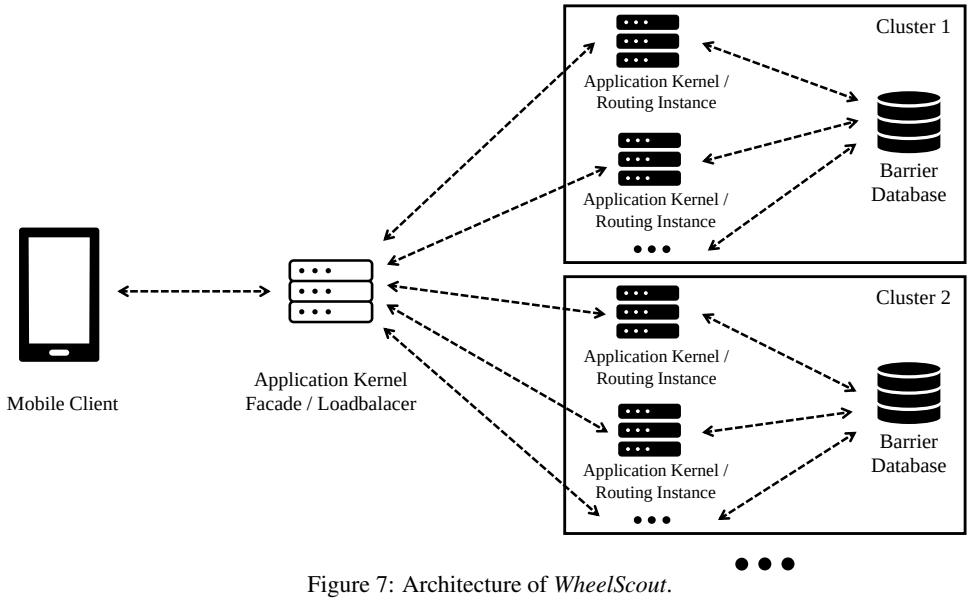

Figure 7: Architecture of *WheelScout*.

graph needs to be rendered in advance to route computation. GraphHopper organizes its graph in edges and nodes whereby each edge is connected to two nodes or one node if there is no further connected edge meaning the edge is an end. Contracted Hierarchies (CH) are an additional mode that allows to speed up the route computation process for longer routes by creating so-called shortcut edges. However they must be disabled in the usercase of *WheelScout*, due to pre-procession the graph. It might result in wrong routes if certain edges are being marked as not accessible but CH is enabled 3.2.1 (Karich, 2020b).

4.1.1 Graph Rendering and Extraction of Barrier Information

The creation of a custom profile is done by creating a new *FlagEncoder* class that extends from another *FlagEncoder* (Karich, 2019a) The method *getAccess* can be overridden to define the accessibility for the certain flagEncoder in the resulting graph based on OSM tags which are consumed as parameters in the method, as shown in the code extract below:

```
public EncodingManager.Access
    getAccess(ReaderWay way) {
    //[...]
    if (way.hasTag("foot",
        intendedValues))
        return EncodingManager.Access.WAY;
    //[...]
    if (!allowedHighwayTags.contains(
        highwayValue))
        return EncodingManager.Access.
            CAN_SKIP;
    //[...]
    return EncodingManager.Access.WAY;
```

To accomplish the in 3.1 demanded possibility of individual adjustable user profiles on client side based on their physical capabilities certain profile settings should not be applied on the graph directly. That is why potential barriers information are extracted during the rendering process of the graph to be stored in a database and then indivdually fetched and applied for each routing request. To achieve that, the process of extracting barriers through the method called *applyWayTags* is provided by the *FlagEncoder* as well. This method is called for each edge together with its particular associated OSM-Way that contains all the parameters from OSM. In the code extract below the extraction of a stairway and its number of steps is exemplarily shown:

```
public void applyWayTags(ReaderWay way,
    EdgeIteratorState edge) {
    if(way.hasTag("highway", "steps")) {
        Integer edgeId = edge.getEdge();
        Integer baseNode = edge.getBaseNode();
        Integer adjNode = edge.getAdjNode();
        Long osmWayId = way.getId();
        Integer number_of_steps = null;
        if(way.hasTag("step_count")) {
           number_of_steps = Integer.
           valueOf(way.getTag("step_count"));
        }
        //[...]
        //Write values into temp csv file
    }
}
```

After finishing the graph's rendering process, but prior to the upload of them to the database, the coordinates, meaning latitude and longitude values of each edge, need to be extracted from the created graph.

The extraction is done through the usage of *NodeAccess* as displayed below:

```
GraphHopper hopper; //object reference
NodeAccess nodeAccess = hopper.
   getGraphHopperStorage().
   getNodeAccess();
double baseNodeLat = Double.valueOf(
   nodeAccess.getLat(baseNode)).
   doubleValue();
double baseNodeLon =
   Double.valueOf(nodeAccess.
     getLon(baseNode)).
     doubleValue();
double adjNodeLat =
   Double.valueOf(nodeAccess.
     getLat(adjNode)).
     doubleValue();
double adjNodeLon =
   Double.valueOf(nodeAccess.
     getLon(adjNode)).
     doubleValue();
double latitude =
   Double.valueOf(
   (baseNodeLat + adjNodeLat) / 2.0D));
double longitude =
   Double.valueOf(
   (baseNodeLon + adjNodeLon) / 2.0D));
```

4.1.2 Apply Barrier Information on the Graph during Routing

In order to consider barriers during a routing request, the particular barriers need to be applied prior to the processing of the routing request. After the routing request has been processed, the prior application of barriers needs to be undone. The technical implementation is that the edge information stored in the database, which consists of the edge id, the base node id, and adjective node id, is first fetched from the database. Subsequently, the accessibility of all fetched barriers or edges is set to false. Those barriers that belong to the edges will be considered as not passable in the following route computation. After the route computation, the process is reversed accordingly. This whole procedure is visualized in the flow chart diagram in figure 8.

In the code extract below the implementation of the edge accessibility adjustment under usage of GraphHopper is shown:

```
GraphHopper hopper; //GraphHopper reference
FlagEncoder encoder; //FlagEncoder reference

private static EdgeExplorer edgeExplorer =
  hopper.getGraphHopperStorage().
  createEdgeExplorer();

private static BooleanEncodedValue e =
    encoder.getAccessEnc();
```

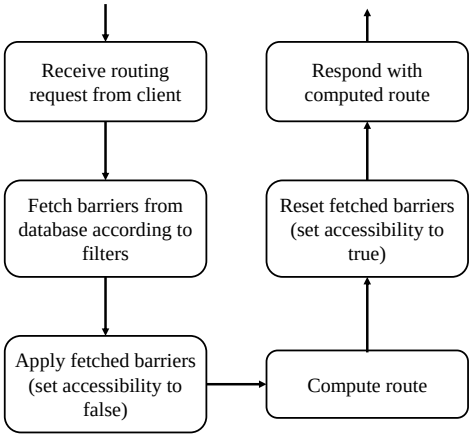

Figure 8: Processing a routing request including applying individual profile settings.

```
public static EdgeIteratorState getEdge(
  int baseNode, int adjNode {
  EdgeIterator iter = edgeExplorer.
    setBaseNode(baseNode);
      do {
          if(iter.getAdjNode() ==
            adjNode) {
              return iter.getEdge();
           }
      }
     while(iter.next());
         throw new Exception(
   "adjNode not found!");
  }
}

public static setAccessibility(
  int baseNode, int adjNode,
  boolean access) {
     EdgeIteratorState edge =
       getEdge(baseNode, adjNode
     edge.set(e, access);
     //forward
     edge.setReverse(e, access);
     //backward
}
```

4.1.3 Handling of Barrier Related Information from OSM

In this section, the handling of relevant information is explained. Currently, *Stairway* and *Uneven Surface* are the two considered barrier types in *WheelScout*. A stairway is tagged as *Highway:steps*. The attributes, considered as relevant for stairways are: *step_count*, *handrail*, *ramp*, *width*, *step_depth* and *step_height*. Potential uneven surfaces are tagged with either *Key:surface* or *Key:smoothness*. *surface* corresponds to the type of surface and *smoothness* is the passability of the way or path (OpenStreetMap con-

tributors, 2021).

While the stairways' attributes are suitable to be filtered directly, it has been found that another attribute *Key:tracktype*, especially in the countryside, is clearly more present. Although not as accurate as *Key:surface*, it is useful to consider this attribute as a substitute when *Key:surface* or *Key:smoothness* are not available.

For this reason, a filter called *intell-surface-type* is developed to assign each *intell-surface-type* a suitable value for *Key:tracktype* and *Key:smoothness* to each *surface-type*.

4.2 Database

To store barrier information in order to fulfill the requirements as described in section 3.1, the relational database MariaDB, which is well established on the market, was chosen for this purpose.

In the following, the core functionalities that the database provides for the navigation system are briefly summarized:

- Adding new and modifying existing barriers
- Removal of barriers
- Receiving barriers for either displaying them on the client or applying them to GraphHopper, see section 4.1.2.
- Importing barriers extracted during the graph rendering process into the database.

4.3 Application Layer

In this section, the application layer consisting of the application kernel and the application kernel facade / loadbalancer is described.

4.3.1 Application Kernel

In addition to smaller, less essential functions, two necessary functionalities are provided by the application kernel. One is the functionality to provide the client with the interfaces, mentioned already in section 4.2, to access the barrier information stored in the database. Client in this context refers to the end-user, e.g., the mobile application, but also to the maintainer, for instance, to update the barriers after updating the map material. Second, the provision of an interface for accepting and responding to routing requests by forwarding them to the routing engine. Since the access of the barrier database, is also used by the routing engine GraphHopper, it was decided to combine the entire logic, including GraphHopper, in one component. In figure 7, this entire logic is labeled as *Application Kernel / Routing Instance*.

The interfaces were implemented as REST API using the embedded Webserver Jetty. As data exchange format for barriers, routing requests, and other minor functionalities JSON (JavaScript Object Notation) has been chosen. The combination of REST and JSON is a common way to exchange data between client and server in business applications (Afsari, Kereshmeh et al., 2016).

The request below, shows an example routing request. The example request asks for a route between two points. The filter parameters attached to the request allow only routes where passed stairways either are equipped with a ramp for wheelchairs or have less than 12 steps and a handrail:

```
curl --location --request GET '{{host}}/route?
point=49.8712,8.6403&point=49.86772,8.64233&
vehicletype=wheelchair&instructions=true&
filter=ramp-wheelchair*EQ*true&filter=or&
filter=number-of-steps*LT*12&
filter=handrail*EQ*true'
```

Below, another example for a request, to receive barriers in a certain radius, here 500 meters, for instance to display them on the client:

```
curl --location --request GET {{host}}/
barriers?point=49.8712,8.6403&distance=0.5'
```

4.3.2 Application Kernel Facade / Load Balancer

As described in the 3.2.3 section, the *WheelScout* architecture supports running different maps on different clusters to take advantage of scaling. For this reason, it is necessary to actually forward all incoming requests from the mobile client to the appropriate cluster, e.g., *Germany* or *Italy*. To achieve that, the facade, which is implemented using Spring Boot needs to be aware of the boundaries of each existing cluster. The boundaries are represented as a polygon. Now, the determination is actually done using an algorithm called *Ray casting algorithm*. The algorithm sends out a ray of the point in the direction to the minimum square border of the polygon and counts the crossings. If the number is odd, the point is situated within the polygon, otherwise not (Taylor, G., 1997). In case of no coordinates are provided in the request, such as modifying an existing barrier by its id, the id has a prefix to identify the cluster.

The second task is, as the name suggests, load balancing if more than one application kernel exists for a certain cluster. As there are no special requirements for load balancing, except that writing requests shall only be processed by the first application kernel in a

cluster, a simple Round-robin scheduling is used for all non-writing requests.

4.4 Client

In section 3.1, a client-side approach that performs the routing on the client has been described. For this approach, it was necessary to use Android as an operating system since Android is currently the only market relevant operating system for mobile devices that allows running Java code on it which is a must to execute GraphHopper.

However, since the final approach requires to run GraphHopper on the server and not on the client, the Cross-Platform Mobile App Development Framework called Ionic had been chosen for the development of the client.

5 RESULTS

In this section, the results gained from the implementation for *WheelScout*, described in section 4, are presented and discussed.

5.1 Routing Results

A great number of different routes was continuously tested to verify the functionality of the system. For example, two different routes in Darmstadt have been tested. One rather short route in the surroundings of Hochschule Darmstadt with three different profiles and one longer route with a single profile from Darmstadt to Darmstadt-Wixhausen. Each route and profile has been tested three times and was evaluated with the number of passed nodes, distance, computing time, and total request time. The measured results of the first route are shown for each profile setting in the tables 1, 2 and 3. The route is also visualized in a different color per profile setting in figure 9. The range of barriers that are applied for the route is visualized as an orange square in figure 10. The measured results for the second route are shown in table 4 and also visualized with its barrier range in figure 11:

Route 1: Start: 49.86779,8.64460; Destination: 49.87047,8.64016

Route 2: Start: 49.86779,8.64460; Destination: 49.9335,8.6571

Table 1: **Route color:** blue, **Applied filters:** No filters.

	Number of passed nodes	Distance	Computing time	Total request time
1	14	580m	177ms	253ms
2	14	580m	188ms	364ms
3	14	580m	176ms	238ms

Table 2: **Route color:** green, **Applied filters:** Number of steps < 12; only always accepted surface types.

	Number of passed nodes	Distance	Computing time	Total request time
1	14	615.85m	231ms	376ms
2	14	615.85m	244ms	380ms
3	14	615.85m	227ms	288ms

Table 3: **Route color:** red, **Applied filters:** Number of steps < 12; accepted surface types: fine_gravel, compacted, pebblestone and always accepted surface types.

	Number of passed nodes	Distance	Computing time	Total request time
1	10	624.6m	236ms	519ms
2	10	624.6m	229ms	368ms
3	10	624.6m	242ms	306ms

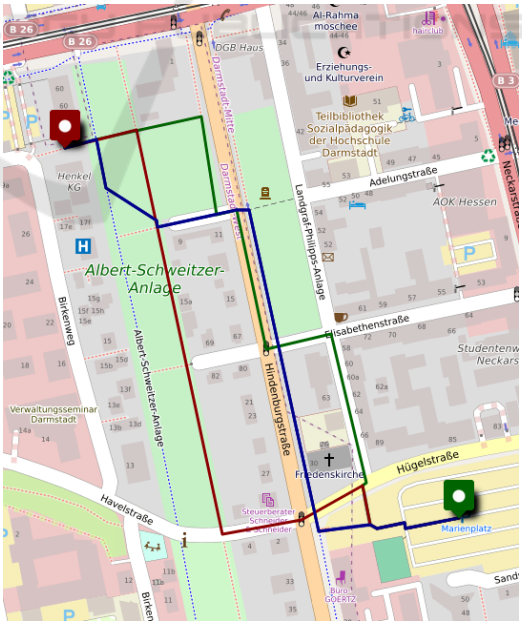

Figure 9: Route 1, first profile: blue, second profile: green, third profile: red - © umap, © OpenStreetMap contributors.

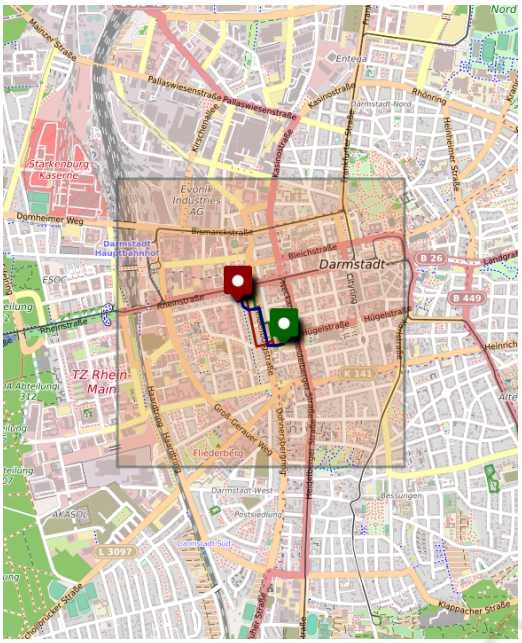

Figure 10: Route 1, three routes, barrier range visualized as orange square - © umap, © OpenStreetMap contributors.

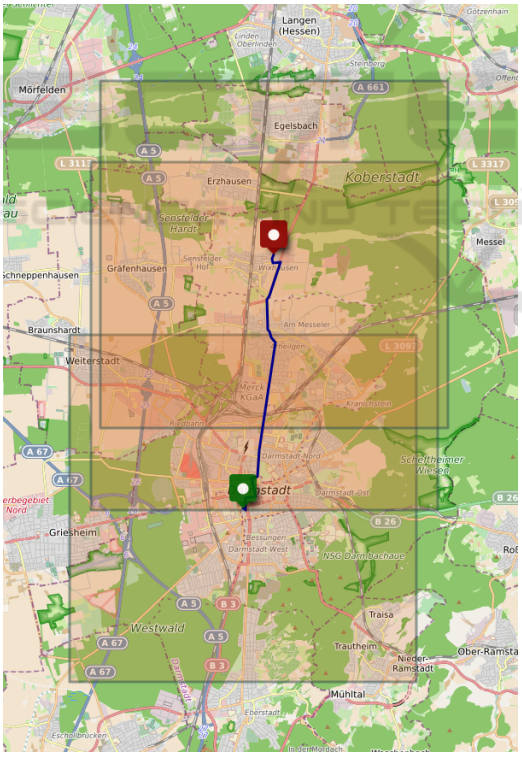

Figure 11: Route 2, barrier range visualized as orange squares - © umap, © OpenStreetMap contributors.

It can be seen that the barrier related information about ways and paths from OSM, as already stated in section 2.1, are sufficient to compute different routes depending on the profile settings. Further, it is noticeable that the computation time clearly increases with the length of the route because of more needed squares and thus more barriers to be considered.

Table 4: **Applied filters:** Number of steps < 12; accepted surface types: fine_gravel, compacted, pebblestone and always accepted surface types.

	Number of passed nodes	Distance	Computing time	Total request time
1	120	8.4km	1366ms	1443ms
2	120	8.4km	1358ms	1503ms
3	120	8.4km	1348ms	1496ms

5.2 Resulting Mobile App

Figures 12, 13 and 14 show some of the core functionalities of the *WheelScout* app. Figure 12 shows the above-introduced route but with uneven surfaces allowed and stairways to be avoided. All existing barriers in the surroundings are displayed by blue markers. The type of barrier depends on the symbol displayed on the marker. Figure 13 shows an example of how the profile settings for stairways can be adjusted in the app. Figure 14 shows how a missing stairway can be added within the app.

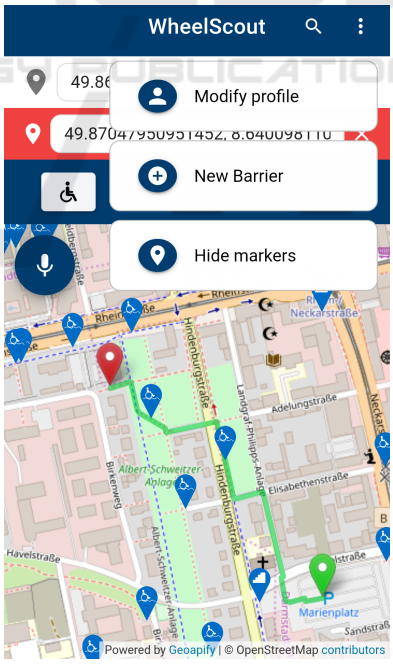

Figure 12: Main screen of the app, route avoiding stairways.

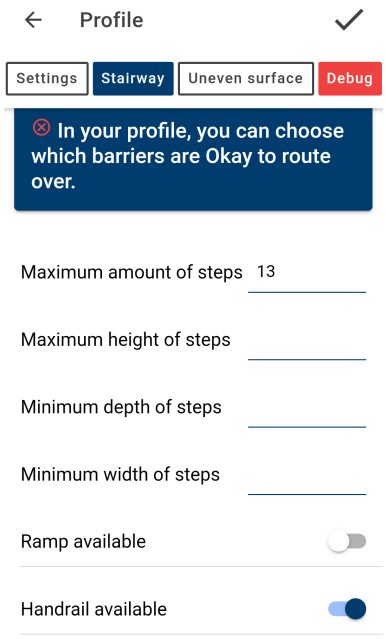

Figure 13: Adjusting profile settings screen, maximum allowed steps 13, stairways must provide a handrail.

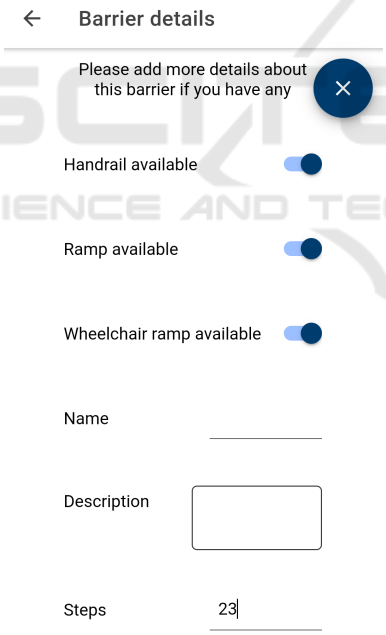

Figure 14: Adding new barrier screen, add a new stairway that has 23 steps and that provides a handrail, a ramp and a ramp for wheelchairs.

6 DISCUSSION AND OUTLOOK

WheelScout is still in an early stage of development. Nevertheless, this work proved that it is possible not only to develop a navigation system for physically impaired people with GraphHopper but also to make it interactive, i.e., users can collaborate by extending the barrier information. Moreover, it is shown that new barriers are directly taken into account without causing delays, for example, due to the need to rerender the map. This fact is a big advantage since OSM already provides much barrier related data. Anyway, there is still a lot of data missing or not up-to-date, see section 2.1.

The problem caused by the fact that longer routes, i.e., 10km and more, have a longer computation time, is less relevant, since wheelchair users mostly move on short, inner-city routes. Furthermore, the provision of multiple routing engines within a cluster and thus the ability to process parallel routing requests can counteract the issue. Nevertheless, additional routing engines also mean more resource consumption. With an average computation time of 500ms, the number of simultaneously processable routing requests is therefore still limited. That is why the new possibility added by GraphHopper to individually influence the computed route with properties such as allowed surface-types or stairways could be very helpful (Karich, 2020a).

With the integration of Point of Interests (POIs) functionality, which currently under development using data from the Accessibility Cloud, see section 2.4, it will additionally be possible for users to navigate for example to accessible restrooms or stores. This is a great advantage for the user group, as not only the accessibility of the POI itself but also the way to get there can be determined in an uncomplicated way.

The option of using speech to operate *WheelScout* is important as well for the user group since potential users often have limited motoric abilities and therefore cannot use the smartphone in the usual way (Padır, Tasskın, 2015).

Finally, the integration of real-time traffic information is also of high importance, especially for people in wheelchairs, as it can be difficult for them to make their way through crowds in busy pedestrian areas. As a result, traffic can also be a potential barrier (Ding, Dan et al., 2007).

ACKNOWLEDGEMENTS

We thank the LAPIC and its members for their cooperation and support.

REFERENCES

accessibility.cloud (2021). accessibility.cloud: Home. https://accessibility.cloud/. Last checked on Feb 16, 2021.

Afsari, Kereshmeh et al. (2016). Javascript object notation (json) data serialization for ifc schema in web-based bim data exchange. doi:10.1016/j.autcon.2017.01.011. Last checked on Feb 16, 2021.

Akasaka, R. (2020). Introducing "wheelchair accessible" routes in transit navigation. https://www.blog.google/products/maps/introducing-wheelchair-accessible-routes-transit-navigation/. Last checked on Feb 16, 2021.

Destatis Statistisches Bundesamt (2020). Schwerbehinderte menschen am jahresende.

Ding, Dan et al. (2007). Design considerations for a personalized wheelchair navigation system. doi:10.1109/IEMBS.2007.4353411.

Google (2021a). Google Cloud. https://cloud.google.com/. Last checked on Feb 16, 2021.

Google (2021b). OpenRouteService: Services. https://openrouteservice.org/services/. Last checked on Feb 16, 2021.

Heidelberg Institute for Geoinformation Technology (2019). openrouteservice.org: About. https://maps.openrouteservice.org. Last checked on Feb 16, 2021.

Karich, P. (2019a). Graphhopper routing engine: How to create new routing profile aka a new flagencoder? https://github.com/graphhopper/graphhopper/blob/master/docs/core/create-new-flagencoder.md. Last checked on Feb 16, 2021.

Karich, P. (2019b). Graphhopper routing engine: Readme. https://github.com/graphhopper/graphhopper/blob/master/README.md. Last checked on Feb 16, 2021.

Karich, P. (2020a). Examples for customizable routing. https://www.graphhopper.com/blog/2020/05/31/examples-for-customizable-routing. Last checked on Feb 16, 2021.

Karich, P. (2020b). Technical overview of graphhopper. https://github.com/graphhopper/graphhopper/blob/master/docs/core/technical.md. Last checked on Feb 16, 2021.

Krauthausen, R. (2010). Wheelmap.org: Faq. https://news.wheelmap.org/en/faq/. Last checked on Feb 16, 2021.

nullbarriere.de (n.d.). Bedarf an barrierefreien wohnungen in deutschland. https://nullbarriere.de/bedarf-barrierefreie-wohnung.htm#:~:text=Danach%20sind%20insgesamt%20rund%201,in%20Deutschland%20im%20Jahr%202019. Last checked on Feb 16, 2021.

OpenStreetMap (2021). About. https://www.openstreetmap.org/about. Last checked on Feb 16, 2021.

OpenStreetMap contributors (2019). umap/guide/import data files. https://wiki.openstreetmap.org/wiki/UMap/Guide/Import_data_files. Last checked on Feb 16, 2021.

OpenStreetMap contributors (2021). Key:highway. https://wiki.openstreetmap.org/wiki/Key:highway. Last checked on Feb 16, 2021.

Padır, Tasskın (2015). Towards personalized smart wheelchairs: Lessons learned from discovery interviews. doi:10.1109/EMBC.2015.7319518. Last checked on Feb 16, 2021.

Poleshova, A. (2020). accessibility.cloud: Home. https://de.statista.com/statistik/daten/studie/1174791/umfrage/unverzichtbaren-apps-nach-altersgruppen/. Last checked on Feb 16, 2021.

Sozialverband VdK Deutschland e.V. (2016). Barrierefreie mobilität ist den menschen am wichtigsten. https://www.vdk.de/deutschland/pages/72149/barrierefreie_mobilitaet_ist_den_menschen_am_wichtigsten. Last checked on Feb 16, 2021.

Taylor, G. (1997). Point in polygon test. doi:10.31030/2250588.

United Nations (2006). United nations convention on the rights of persons with disabilities. https://www.un.org/disabilities/documents/convention/convention_accessible_pdf.pdf. Last checked on Feb 16, 2021.