LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department of Computer Science
Database Systems Group
Prof. Dr. Thomas Seidl

Bachelor's Thesis

Designing and Developing a Modular, Web-Based User Interface for Query Processing and Simulation in Road Networks for the MARiO Framework

Rainer Wittmann
r-wittmann@outlook.de

Supervised by: Gregor Jossé
PD Dr. Matthias Schubert
Submitted on: August 19, 2016

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in this thesis.

Unterhaching, August 19, 2016

. . . . . .

Contents

1 Introduction

One big part of the theoretical and practical work, done by the Database Systems Group (DBS) at the Ludwig-Maximilians-Universität in Munich (LMU) regards graph data and algorithms that solve problems on graphs, in particular but not limited to the search for shortest paths from one node to another. The algorithms which are solving this issue, like Dijkstra's algorithm and A*-search for example, are constantly refined to perform faster, less error prone and more precise and also find use in other algorithms to solve more complicated tasks. In 2011 the Multi Attribute Routing in Open Street Map Project (MARiO) [3], was initiated to further enhance the development process, the implementation and the visualisation of new or updated algorithms and methodologies.

"MARiO is an open source project combining functionalities for data integration and preprocessing, implementation of new algorithms and performance evaluation" [3]. It was developed in Java 1.6 and follows a plugin architecture, i.e. allows the implementation of new algorithms and functionalities without the need to alter the original code base. While at the time of development Java 1.6 and the plugin architecture where modern and advanced technologies, today, due to the advancements in programming languages and software patterns, MARiO is outdated, takes extensive amounts of time to download, compile and start on a local machine. Further information about MARiO will be given in section 3.

Nonetheless, MARiO is an important, frequently used and regularly extended framework. As recently as two weeks ago Markus Rohm added functionality to MARiO integrating stochastic cost criteria into road network graphs in his project thesis 'Hazard-Aware Routing'.

The task of this bachelor's thesis was to design and develop a modular, web-based user interface for query processing and simulation in road networks for the MARiO framework which is achieved through a visualisation for the parallelly implemented backend server at the DBS. This type of architecture is called server-client architecture which separates the representation on the client side from the processing and storing of large amounts of data and the implementation of algorithms on the server side. Implementing the backend was not part of this thesis.

The task started out as a interface for querying and visualising route calculations with algorithms implemented on the backend. Over time it evolved into a full blown mapping application with features like route calculations, point of interests, and many more.

At the beginning of this thesis in section 2, the reader will be introduced to underlying technologies, AngularJS and Leaflet, to geospatial encoding formate GeoJSON and to HERE, a company offering a large variety of map designs and mapping technologies. All frameworks and specifications will be considered from various standpoints, alternatives will shortly be mentioned and benefits and drawbacks discussed.

The frontend, developed in AngularJS for large browser and platform compatibility, enables the user to request route calculations from the server and displays the response on a modern and easy to use mapping application with additional route information like travel time and energy consumption. Supplemental features were implemented to round out the application, which will be introduced in detail in section 4. Furthermore the development process, the frontend performance and legal considerations will also be discussed at that point.

As previously mentioned, MARiO was constructed following a plugin architecture which could also be described as modular. Most modern design patterns in software development follow a principle called separation of concerns and follow design goals of low coupling and high cohesion. Section 5 gives a introduction to modularity in software development and expands the idea to web development and it's use in this thesis' developed application.

To conclude this thesis a brief summary and recapitulation will be given in section 6. Interesting future work, which is identified throughout this thesis, will also be acknowledged.

The frontend application can be accessed for testing or just to have a look at https://r-wittmann.github.io/mario. Furthermore the code of the project can be cloned from my Git repository at https://github.com/r-wittmann/mario for analysis and further development.

2 State of the Art

As the software developed in the course of this thesis depends on various frameworks, specifications and other projects, this section will illustrate and determine the boundaries of the underlying technologies used. First an overview of AngularJS, a JavaScript framework for dynamic views is given, followed by an introduction to the Leaflet project for mapping applications. The third section covers GeoJson as a standard for geospatial data and finally the HERE API for map imagery and additional map data will be introduced.

2.1 AngularJS

When implementing a web application, HTML, CSS and JavaScript are the most common technologies of choice. HTML is great for declaring static documents and CSS for styling, but both lack the ability to react to dynamic changes of values and views and are hence not up to the challenges of a modern web application. As HTML and CSS are rather old technologies various developers generated a multitude of ways to compensate said shortcomings and JavaScript was one of them. Developed in 1995 by Netscape employee Brendan Eich, JavaScript has been in a continuos development process, is now implemented in all modern web browsers without the need for plugins and almost all websites depend on it. JavaScript is standardised by Ecma International in the latest ECMA-262 language specification [18].

But still there was room for improvement. In the past few years, with web 2.0, excessive use of mobile browsers and the resulting higher user interaction with websites, additional functionalities were needed and consequently developed. But non of these frameworks addressed the basic problem with HTML, namely that it was not designed for dynamic views. AngularJS on the other hand extends the existing HTML vocabulary and allows for expressive, readable and quick to develop code. Based on HTML and JavaScript, AngularJS is a framework which allows developers to have full access to standard syntax and facilitates the dynamization of web applications.

Normally when a user visits a website, the HTML template is requested from the server and loaded into the web browser, including CSS for styling and all JavaScript files needed for the utilisation of the website. AngularJS applications differ from that process. In the bootstrapping phase, which is started right after the HTML markup is loaded, the core AngularJS functionalities are merged with the template to provide basic features. After the AngularJS code is available in the web browser, the module referenced from the HTML template is generated and all dependencies are injected into and made available to this module. Also the scope is introduced to the application allowing the initially generated static view to be replaced by AngularJS's dynamic view.

AngularJS is based on the model view controller pattern (MVC) which compels a strict separation of the underlying data structure, the visualisation and the implemented business logic. The so-called scope, which is used as the model, is able to bind view and controller, as it is accessible from both components. Each AngularJS application has exactly one root scope which may have several child scopes as every directory creates its own scope. The child scopes inherit from their parent prototypically. Controllers can use services, factories and providers for additional functionalities which are part of the above mentioned dependencies injected into the controller.

The ability to bind view elements to variables and functions on the scope and vice versa is called two way data binding and is one of the main reasons AngularJS is favoured over many other frameworks. All changes to the view, for example via input fields, are immediately reflected to the scope and all changes to variables on the scope are propagated to the view keeping them both up to date. All functions stored on the scope are accessible from the view elements allowing for easy and fast implementation of click listeners and other business logic.

As JavaScript is an untyped language, which means that variables don't have to be declared as one particular type, the developer receives almost no errors while compiling the code and is therefore often confronted with the issue of having to find errors by himself. AngularJS has built-in features for testing to simplify the process. The two basic concepts of testing, unit tests and end-to-end tests, will be explained in the following two paragraphs.

Unit tests are about testing small units of code without building the whole project around it. This ensures locating errors exactly were they occur and minimise the risk of cross contamination between modules and controllers. Karma and Jasmine are two favoured tools for unit testing in AngularJS applications. Both tools are easy to use and allow for structured tests of small units of code. Especially Jasmine makes it easy for the developer to write readable code which is easy to understand and maintain.

Once an application becomes bigger, manual unit testing becomes exponentially more time consuming and harrowing, hence it's no longer a viable recommendation. Additionally some issues may occur when integrating components or resulting from interplay of more than one module. At this point the developer has to consider end-to-end testing which can be executed automatically. The recommended tool for end-to-end testing is called Protractor which, using the same syntax as Jasmine, allow for browser testing and the simulation of user interaction.

For this application AngularJS version 1.5.5 was used, since Angular 2 was not yet at the point of a stable release during this project. AngularJS is mainly maintained by Google and by a community of individuals and cooperations which contribute to the open source project.

2.2 Leaflet

The goal of this thesis was to enable users to request routes using various algorithms and methodologies implemented on the server at DBS and display the servers response on client side. For this purpose a mapping framework was needed to handle all issues regarding the download of map tile imagery, user interaction and overlay handling. The API for requesting map imagery will be described in 2.4.

"Web based mapping has evolved rapidly over the last two decades, from MapQuest and Google to real-time location information on our phones' mapping apps" [1]. Consequently various alternatives are available for mapping applications. After evaluating the possibilities, Leaflet stood out for its light weight of about 33 kB of JavaScript, the compatibility with almost all platforms, the vast amount of third party plugins and the simplicity, usability and performance of the API.

Other projects and products under consideration were, for example, MapBox and the Google Maps API. MapBox is a company offering a variety of services, such as the designing and hosting of maps, geocoding, routing and many more. Unfortunately the terms of use are rather complex and restricting. 2013 MapBox and Leaflet joined forces and MapBox is now using Leaflet for their frontend integrations [20]. These were the main reasons MapBox was not chosen for this project. The Google Maps API is even more restrictive than MapBox. Only Goggle Maps imagery can be used with the API and it's not allowed to display third party information, such as the calculated routes from the server of this project, which rendered it obsolete for this project.

This paragraph will shortly summarise the history of Leaflet. After discovering the complexity and bulkiness of the then industry standard for mapping, OpenLayers, in 2008, Vladimir Agafonkin decided to implement a simple, lightweight mapping library. Albeit the resistance of his superiors at Cloud Made, his employer at the time, and criticism from the community of map developers, he succeeded in developing the currently most popular framework for mapping applications [12]. Since it's initial release in May 2011, which bore limited functionalities and the absence of support for major open standards such as the Web Map Service (WMS) [27] and the Web Feature Service (WFS) [26], the Leaflet framework has matured considerably. As the second release candidate 1.0-rc2 is out of development, major version update 1.0 is soon to be deployed with even more features and improved mobile friendliness.

For the development of this project the most recent stable Leaflet version 0.7.7 (published October 26, 2015) was implemented. Leaflet is maintained by now MapBox employee and original creator Vladimir Agafonkin who is widely supported by a large base of contributors and benefits from the expertise of open source developers implementing third party plugins.

To leverage the combination of the power of AngularJS and the simplicity of Leaflet the Angular-Leaflet-Directive (ALD, developed by David Rubert), was introduced to this project for an easy integration of Leaflet and bi-directional binding between the map and AngularJS's scope. With ALD it is possible to include a Leaflet map with just one line of HTML code.

ALD version 0.9.0 was used for this project. It is still maintained by original creator David Rubert with major help from the AngularUI project.

2.3 GeoJSON

Developing a map application, one needs to stipulate a common file format to transfer data between the involved parties. Even though there are many suitable and well documented file formats like KML or TopoJSON, the decision for GeoJSON was made because of the direct parseability of GeoJSON to map objects in Leaflet. Plugins are available to convert many other formats to GeoJSON, but because it has to be converted, the performance is not comparable to the native compatibility of GeoJSON to Leaflet. This section will introduce GeoJSON as a geospatial format and shortly summarise its use in this project.

Before going into detail on the GeoJSON formalities, it would be expedient to account the JavaScript Object Notation (JSON). The JSON Data Interchange Format, as standardised by Ecma International in ECMA-404 [19], makes use of JavaScripts object literals,

as specified in ECMA-262 [18], which was intended to and is now widely used for the interchange of data between all programming languages. In comparison to other file structures, such as XML or YAML, JSON is considerably lighter, more human readable and better parseable to JavaScript than all other formats.

Wrapped in an object, which consists of key value pares, JSON values can either be objects, arrays, strings, integers, booleans or null. A simple example could look like this:

```
{
  "organisation": "Institut für Informatik",
  "address": {
    "streetname": "Oettingenstraße",
    "streetnumber": 67,
    "postalcode": 80538,
    "city": "München"
  },
  "groups": [
    "Kommunikationssysteme und Systemprogrammierung",
    "Datenbanksysteme",
    "Programmierung und Softwaretechnik",
    "Programmier- und Modellierungssprachen",
    "Theoretische Informatik",
    "Bioinformatik",
    "Medieninformatik"
  ]
}
```

"[JSON] does not attempt to impose ECMAScript's internal data representations on other programming languages. Instead, it shares a small subset of ECMAScript's textual representations with all other programming languages." [19]. JSON is supported by virtually all modern programming languages.

Using all the advantages of JSON, the GeoJSON format is intended and used for encoding, storing and transferring data about geographical structures [16]. Points, LineStrings and Polygons are the simplest geometries supported by GeoJSON which can be extended to MultiPoints, MultiLineStrings and MultiPolygons. In most cases one wants to attach certain information other then the geographical extend to objects. For this use case Features are introduced which consist of a geometry, i.e. one of the above mentioned objects like Points or MultiPolygons, and an additional properties object. The properties attached to a feature can be any kind of JSON object. "That said, given the fact that no other prominent geospatial standard supports nested values, usually the properties object consists of single-depth key -> value mappings" [21]. The most commonly used object type is FeatureCollection, which is, as the name suggests, a collection of Features.

Requests for route calculations from the developed frontend application to the backend server at DBS are encoded in JSON format, whereas the servers response is in GeoJSON format allowing an easy and fast parseability to ensure an enhanced user experience for the client. All responses are FeatureCollections containing at least one LineString with the properties object heavily used for route information. File sizes vary from 1KB for very short routes to about 1.5 MB for large simulations (see 4.4.5).

2.4 HERE API

As Leaflet offers just the wire frame application handling all user interaction and supplies a framework for mapping, a provider of map imagery, also called tiles, is needed to provide the actual map data. The standard implementation of Leaflet uses tiles from the OpenStreetMap Foundation (OSM) which is both convenient and sufficient for most mapping applications. Fortunately Leaflet offers the possibility to effortlessly switch to a different tiles provider. This section will shortly summarise the history of the HERE company, describe the HERE APIs, give reasons for the decision to use HERE and uncover various benefits and shortcomings of the technology.

The origins of HERE go back to a company called Karlin & Collins, Inc. founded in 1985, later called Navigation Technologies Corporation and ultimately NAVTEQ. NAVTEQ was one of the first companies providing turn-by-turn navigation for the San Francisco metro area. Switching business model to licensing map materials to hardware providers instead of building the hardware itself, accelerated the rise of the company which invested heavily in the development of mobile mapping technologies. In October, 2007 NAVTEQ was acquired by Nokia [24] to merge it with its own mapping technology. To streamline map services Nokia started the brand HERE in October, 2011, which later was sold to a consortium of German car manufacturers in August, 2015. Now HERE offers smartphone and web map applications including routing, points of interest, real time traffic information and geocoding.

Resulting from the in section 4.4.1 described wish for a points of interest search (POI) and the subsequentiell decision to use HERE as POI API, the legal terms determined that the so called Places service can only be used with and displayed on HERE map tiles. Fortunately HERE offers a large variety of differently styled map imagery including satellite images and a dark, unlabelled version which would be important for the in section 6 described future work of displaying visited nodes.

Actually HERE offers not only map tiles and point of interest search, but also a geocoding API to translate location data into physical addresses (see section 4.4.4) which concludes the use of HERE APIs in this project. The ultimate decision to stay with HERE and use tiles, Places and geocoding from their offering was that everything would be managed within one account which eases the maintenance of the project. Not using HERE would have resulted in creating three separate accounts with different API providers adding unnecessary inconvenience and complexity.

The download and replacement of map imagery is completely handled by Leaflet, resulting in only basic configuration of the map mode and providing the access credentials. With Places and geocoding the developer comes into close contact with the inner workings and structure of the APIs. To provide a similar response structure across all APIs HERE returns rather complex, deep nested compositions of either XML or JSON. Unfortunately GeoJSON is not supported by the HERE APIs but using JSON as response type was the obvious choice for the compatibility and parseability to JavaScript.

This project is subscribed to the Basic Plan free of cost, which results in limitations both in transactions and features. It allows the application to cause up to 50.000 transactions per month and limits the features to map imagery, geocoding, the places service and car and pedestrian routing. Various other plans are available including a free 90 days trial of the entire platform. If this project should ever be considered as a public offering

the rate limitation to 50.000 transactions per month will be reached rather quickly. A plan for universities and other non profit organisation is available, which could be taken under consideration.

3 Description of the Backend

The MARiO project is an important research tool for the DBS as it allows the implementation and testing of new algorithms and methodologies on graph data. Additionally the performance of the calculations can be measured with regards to speed, accuracy and susceptibility to errors. This section introduces the MARiO project in two parts. Subsection 3.1 will point out functionalities and features of MARiO and subsection 3.2 will deal with the modernisation of the backend including the newly implemented API contact points.

3.1 Backend Functionalities

This section will give an overview over the various functionalities of the MARiO project. MARiO is implemented in Java 1.6 following a plug in architecture allowing researchers at the DBS to add new functionality to the project without altering the original code base. For further information the original publication regarding MARiO [3] is highly recommended as it gives a more detailed description of the implementation.

The necessary map data is provided by the OpenStreetMap Foundation (OSM). Since it contains irrelevant data, like floor plans of buildings for example, and vital information for multi criteria route computation, like traffic lights, is missing, preprocessing of the data is an essential functionality. Accordingly the original OSM map is relieved of unnecessary information and traffic light data and topological information from NASAs SRTM[1] program are merged into the edges of the graph. Analysis has shown that it is additionally possible to remove certain nodes from the graph which are irrelevant for route calculations [3].

For route calculation based on a single cost criteria, MARiO implements Dijkstra's algorithm and A*-search. The A*-search algorithm can optimise on distance as cost criteria, whereas Dijkstra's algorithm is implemented to take distance, time or energy consumption as criteria for optimisation. MARiO also includes a route skyline query implemented with the ARSC algorithm (see [7]), which computes the set of all pareto optimal solutions between two given locations regarding monotonic combination of the selected cost criteria, and a linear route skyline algorithm (see [10]) computing only pareto optimal solutions regarding linear combination of the selected cost.

Dijkstra's algorithm is also used to offer intermodal route calculations as an additional feature, which is also used in this thesis (see section 4.4.3). This methodology takes start and end point and calculates the route using not only the street network, but also means of public transportation. Unfortunately it is experimental and only available for the Berlin metro area.

The last component of MARiO is the also Java based frontend view displayed in figure 3.1. It enables the user to post queries by selecting start, end, algorithm and algorithm properties; resulting in the visualisation of the outcome on the map.

As skyline calculations return multiple routes for one query, MARiO also implements clustering algorithms. The result of said clustering can be seen in the bottom left corner of figure 3.1.

[1] http://www2.jpl.nasa.gov/srtm/

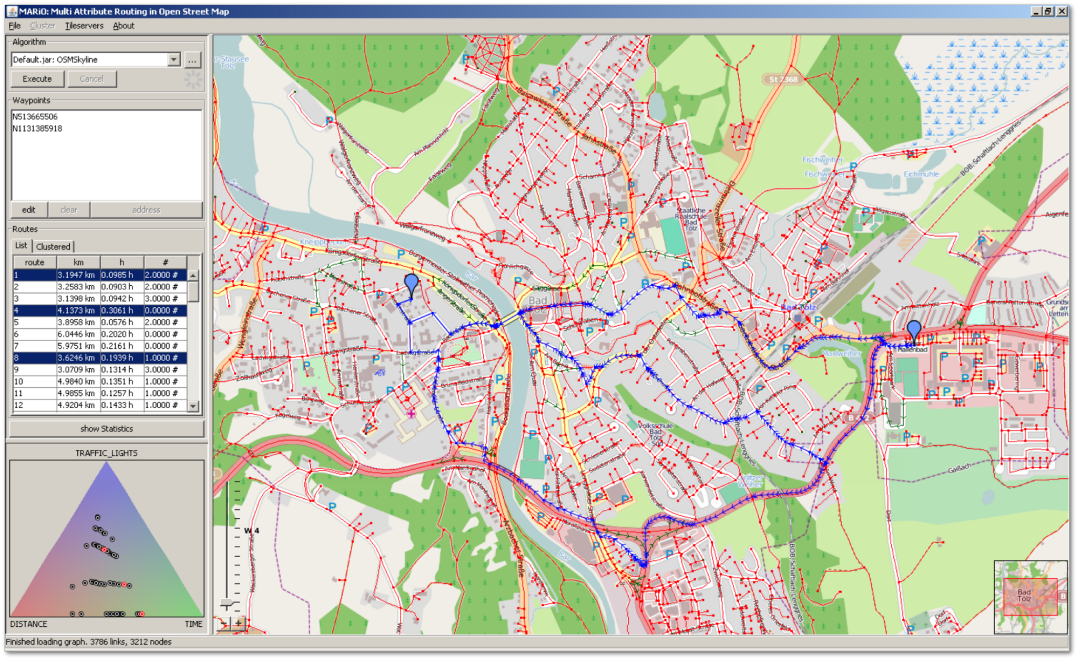

Figure 3.1: Screenshot of the graphical user interface of the MARiO project

3.2 Modernisation of the Backend

As mentioned in the previous subsection, MARiO was developed as a Java project following a plugin architecture [3]. As the project grew over the last few years, MARiO lost its MVC separation and became a bulky, hard to handle project. This fact is also due to the multitude of contributors to the project, letting it grow without much of a quality control system as, for instance, in modern open source projects. This subsection takes into consideration the process of consolidation of features as well as the modernisation of the graphical interface.

The modernisation started with a bachelors thesis about shared fleets of electrical vehicles written by Ludwig Zellner at the DBS in 2015 [6]. In collaboration with Gregor Jossé, Zellner proposes a management tool for shared e-fleets providing features for both the user of the vehicles and the operator of the fleet. Quite similar to this thesis, the user was enabled to request route calculations from his position to arbitrary locations with the possibility to include a charging point in the route. As the driver used a mobile application for this service it was infeasible to install the whole MARiO project on each mobile device.

By consolidating the needed functionality of route calculations and implementing them on a stand alone server which was accessible via a single API contact point the user was able to request route calculations by providing start and end point and some additional information via a http-request. This was the first step of the separation of the visualisation from the implemented functionalities which would be further evolved during the development phase of this thesis.

The implemented server is separated from the clients by a REST (REpresentational State Transfer) service. REST is an approach focused on the communication between server and client often used in the development of web services. It is a perfect fit for applications communicating with a server because of the small overhead necessary to

request and send data over the internet.

As mentioned, REST services are like a façade for the client as it allows him to communicate with a server and vice versa, without the need to directly reference each other. This kind of separation of concerns will be explained in detail in section 5.

The REST service implemented for the new MARiO backend offers two points of contact for clients to post queries and receive data from. One is the for the shared e-fleet used API easyEV, which can be contacted at `http://129.187.228.18:8080/restservices_path/webresources/easyev?` and expects the selected algorithm, an array of costs and, most important, start and end point of the requested route.

The second point of contact is for intermodal route calculations, which is accessible at `http://129.187.228.18:8080/restservices_inter/webresources/intermodal?` and expects the departure time, the range, one is willing to go or drive in the route network, and start and end point of the route. Two features which aren't available yet are the restriction of changes and the amount of means of public transportation. Therefore the highest integer number (2,147,483,647) is transferred for that attributes. With a round trip time (RTT) of only 35 ms to the server and about 50 ms to the specific port respectively, both REST services are highly responsive.

4 Development of the Frontend

The goal of this bachelor's thesis is to develop a user facing browser application to allow route calculation requests and various other features (see 4.4). The process of developing this software, an overview over the used tools and frameworks to support development, a description of the implemented application and its performance and a specification of the included features will be given in this section.

4.1 Build Tools and Frameworks

To ease and support software development a multitude of tools were developed in recent years. This subsection will introduce Git as a tool for version control and GitHub-Pages as deployment environment. Additionally JavaScript build and dependency management tools npm and bower are delineated and agile software development is defined.

4.1.1 Version Control and Deployment

While developing software there comes a point of impasse, during which an error sneaked into the code or the implementation was simply wrong or unnecessary. In such cases version control systems allow the developer to go back in time to a previously saved condition, revert changes to the software and compare modifications to the source code. Version control, e.g. a logical way to organise and control revisions and changes, is a practice that outdates software development, but in the course of the latter has grown into a technically more complex and relevant tool. Wether for collaborative development processes or smaller projects, "[v]ersion management is essential to software development and is considered the most critical component of any development environment" [15].

Like many other technologies, version control systems matured considerably in the last twenty years from locking files, so only one developer or writer can make changes, like RCS and SCCS to rather complex, distributed tools allowing simultaneous changes to files and merging the changes afterwards, like Mercurial and Git. Recently Git has become the tool of choice for modern projects in software development.

A popular web based Git repository is the hosting service GitHub, a company founded in 2007, which incorporates all Git functionalities and adds features like an issue tracker and task management. One particular feature, named GitHub-Pages, will be explained in the next paragraph. GitHub currently hosts, according to their own statements, more than 35 million repositories [25] making it the largest source code host world wide [22]. For this project Git was used for version control on GitHub with a workflow described in 4.1.3.

To be accessible for the public, web applications have to be hosted on a web server with an associated IP address and URL. GitHub-Pages (GHP) come in handy for this purpose, albeit it is not the originally intended use of GHP. GitHub provides every repository with the ability to host a publicly available project website accessible at `<repository-owner>.github.io/<repository-name>` to describe the project and offer sales pitches for chargeable software. GHP was alienated as productive deployment environment for this project.

4.1.2 Dependency Management

"You can only get so far using the language features and core functions. That's why most programming platforms have a system in place that allows you to download, install, and manage third-party modules" [11]. Node.js's package manager npm has established itself as the go to technology for dependency management in server, as well as frontend JavaScript application development. Node.js is a popular JavaScript framework for server implementations using non-blocking input/output streams facilitating efficient backend applications.

Npm allows developers to share their code and consume others as dependencies to their project. More than 300.000 packages are available from npm with a growth rate of about 500 per day [17].

To ease the development process and reduce memory space of the Git repository it was decided to split dependencies in development and display dependencies. All third-party packages for the development process, like a local http-server and a css post-processor for vendor prefixes, are managed by npm, whereas all dependencies needed for the actual display of the frontend, like AngularJS and Leaflet, are installed by bower, another powerful JavaScript build tool. Only the bower managed dependencies are committed to the code repository as the development dependencies can be installed by the developer himself.

Several scripts are implemented to start various tasks concerning the development process. The following examples assume a command line tool is pointing at the right directory and has administrator permissions. Running 'npm install' installs all npm-packages the software is depending on, including for example bower and postcss-cli. This step can take a few seconds to up to a minute on the first run, depending on how many packages have to be downloaded and installed. After installing all dependencies running 'npm run develop' starts a development environment by installing all bower dependencies, starting the application on a local http-server and opening the standard browser to the configured localhost port to load the application. The command 'npm run build:css' was implemented to use autoprefixer to include all css vendor prefixes for the display in recent versions of all web browsers.

4.1.3 Agile Software Development

Software development in todays fast moving business environment is bound to be able to react to changes in specification quickly. Agile development methods are designed as an iterative process containing specification, development and delivery of software in short iterations. The next paragraphs will introduce Scrum as an agile development method and describe the method used in the development phase of this thesis.

Introduced by Ken Schwaber and Jeff Sutherland at the OOPSLA conference in Austin, Texas (US) 1995 and now manly promoted by Jeff Sutherland, Scrum is more of a process managing framework than a method of software development. "SCRUM is a management, enhancement and maintenance methodology for an existing system or productive prototype. It assumes existing design and code which is virtually always the case in object-oriented development due to the presence of class libraries" [9]. The development is divided into so called sprints which last for one to three weeks and iteratively implement the most important steps of a classic waterfall model in software development. Starting

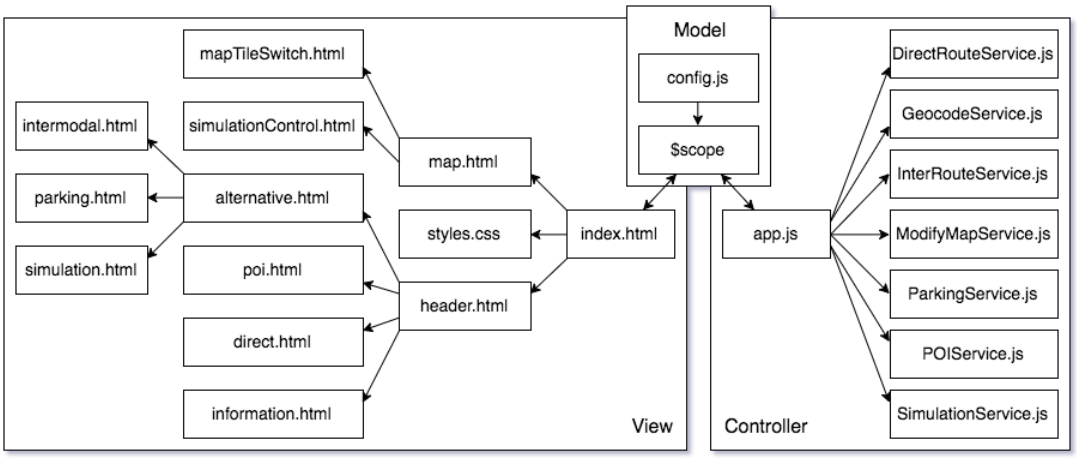

Figure 4.1: Diagram of the application structure

with the sprint planning, i.e. the software specification, followed by the development phase concluding with a demonstration of the implemented features. At the end of each sprint the customer receives a working software to evaluate and gives feedback, which will be incorporated into the next sprint. This process guarantees close collaboration of both the interdisciplinary development team and the customer and ensures an end product to the customers satisfaction.

The development process of the MARiO frontend can be described as close to Scrum. Every two weeks the state of the project was demonstrated to the supervisor which in turn gave feedback on the software and the implemented features. A feature driven approach was maintained with the main goal of implementing one of the in section 4.4 described features per sprint. For big features, like the Route Calculations (subsection 4.4.2) two sprints were scheduled to allow a thorough treatment. Every feature was developed on a development branch and only merged into the master branch after functionality was guaranteed leaving an always working version on the master branch. Once a feature was successfully integrated into the master branch, it was deployed to the gh-pages branch which is the foundation of the GitHub-Pages service.

4.2 Implemented Application

As previously mentioned, the goal of this thesis is to develop a browser application allowing a client to request routes in a street network using algorithms and methodologies implemented by the DBS. The following paragraphs will characterise this application and the use of the in section 2 introduced technologies and describe the interface design process.

Following the recommendations of AngularJS, the application implements a strict MVC separation where AngularJS's scope is the model which binds the view to the controller. The view consists of several HTML files forming a hierarchical, tree like structure of components starting with index.html as root. The controller is implemented in the JavaScript file app.js which depends on several services. Each service handles all API calls and implements business logic concerning one of the in subsection 4.4 detailed features. The

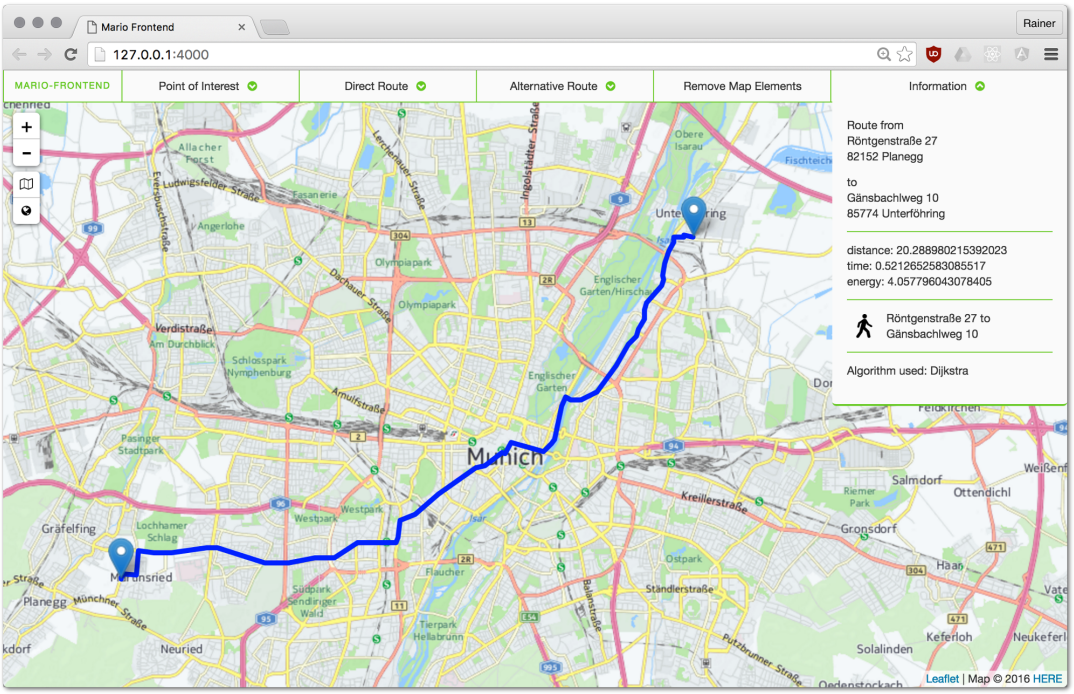

Figure 4.2: Frontend with calculated route and route informations

ModifyMapService handles everything concerning modifications of the map, e.g. the positioning of the marker and the switch of map imagery to satellite images. The controller itself can rather be seen as a façade delegating all user interactions to the appropriate service.

The model receives its initial configuration, containing mostly data to boot Leaflet, from the config.js file where it is stored in a JavaScript object literal (see 2.3 for more information on JavaScript object literals). The file structure of the application can be seen in a diagram in figure 4.1 clearly showing the MVC separation.

AngularJS allows the developer to create so called directories which than can be used like standard HTML syntax, many of which are published by their developers via GitHub or npm (both explained in subsections 4.1.1 and 4.1.2 respectively). One of these directories, the Angular-Leaflet-Directory was used to merge the AngularJS architecture with Leaflet. The <map>-tag, which renders the Leaflet map, expects attributes to bind image source, map centre, markers and paths to AngularJS's scope.

Much time and effort was put into the user interface design to ensure usability and a great user experience (see figure 4.2). A horizontal navigation bar was implemented in the top area of the application to maximise the space available for the map which covers about 85 to 95% of the available screen size, depending on the device, compared to a screen coverage of about 70 to 80% using a vertical navigation. The design is responsive and was tested on small mobile devices, through various tablet and laptop devices up to displays with 4K resolution (3840 x 2160 pixels).

To not overexert the users eyes but still provide reasonably high contrast the background color used for the navigation bar and dropdown menus is not white, but slightly darkened to a hex code of #FAFAFA and text color was slightly lightened from black to

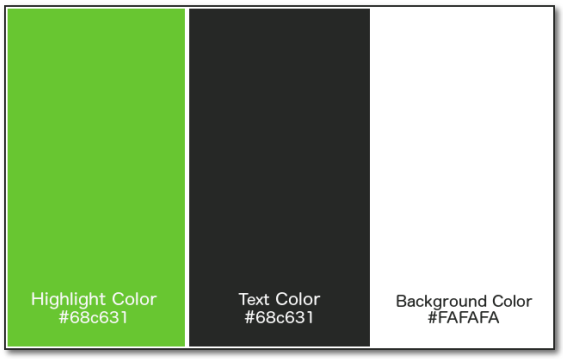

Figure 4.3: Color palette

URL	First Byte	Start Render	DOM Elements
r-wittmann.github.io/mario	0.164 s	1.491 s	216
maps.google.de	0.453 s	1.500 s	514
www.openstreetmap.org	0.470 s	3.206 s	383
wego.here.com	0.773 s	0.995 s	352

Table 4.1: Basic Performance of frontend compared to other applications

a hex code of #262826. Out of four alternatives a light green (hex code #68C631) was chosen as a primary color, used for example for several icons, the separators and the route highlighting (see subsection 4.4.2). Colors can be seen in all screenshots included in this thesis and in the color palette in figure 4.3.

4.3 Performance

Performance played a big role in the development of the frontend application to ensure great usability. That is why file sizes where kept to a minimum, html files are only loaded, once they are needed and AngularJS and Leaflet were chosen as technologies (see 2.1 and 2.2). This subsection will compare the newly developed frontend to established mapping applications, like Google Maps and OpenStreetMap in various dimensions.

The first measurement, as seen in table 4.1, is mostly about how fast an application starts to load. The time to first byte however, is not related to the application itself, but to the deployment environment. It is a measure for how fast the server response starts to arrive at the clients browser after requesting a page load. The GitHub-Pages server is quite quick in that perspective, being three to four times faster than the other servers.

The next column displays the time, until the application starts to render the first elements. With 1.491 seconds the MARiO frontend is in second place only beaten by wego.here.com, which starts after a little less than a second. Noticeable is the very short time span of 0.222 seconds between first byte and start render for the HERE application. Optimising the render process of the MARiO frontend would certainly reduce the time between first byte and start render significantly.

The last column of table 4.1 is the amount of Document Object Model (DOM) elements the website is composed of, once it is fully loaded and rendered. The low number of 216 DOM elements is mostly the consequence of file and structure optimisation and

	Document Complete		
URL	Time	Request	Bytes In
r-wittmann.github.io/mario	0.904 s	23	183 KB
maps.google.de	6.265 s	105	494 KB
www.openstreetmap.org	3.621 s	28	995 KB
wego.here.com	5.783 s	24	1298 KB

Table 4.2: Performance of frontend in regard to completed document load

	Fully Loaded		
URL	Time	Requests	Bytes In
r-wittmann.github.io/mario	2.176 s	47	689 KB
maps.google.de	7.569 s	122	1816 KB
www.openstreetmap.org	4.807 s	34	1109 KB
wego.here.com	7.937 s	50	2743 KB

Table 4.3: Performance of frontend until application is fully loaded

also impacts loading and rendering times.

After the first byte arrived in the browser and the application started rendering, at some point, the whole content originally requested from the server is available to the browser. Table 4.2 shows the comparison between the MARiO frontend and the same mapping applications as before in regards to the complete document download.

The time until the document is downloaded completely mostly depends on the speed of the internet connection, in particular the upload speed of the server and the download speed of the client. As the tests were performed with the same internet connection and during different times of day, the download speed to the client can be disregarded. Similar to this, we can assume that Google and the other providers have a pretty fast internet connection, too, as their business model depends on it.

The MARiO frontend is loaded in 0.904 seconds and is thereby three to seven times faster than the other applications. As we excluded the internet connection on both client and server side as a high influence to loading times, the difference originates mostly from the small amount of data which has to be transferred to the client's browser. With 183 KB of HTML, JavaScript and css, the MARiO application is far smaller than the competitors.

The original complete loading speed and following efficiency and speed of operationalisation is key to perceived performance value for the destination user. Similar to the previous tables, table 4.3 also shows the MARiO frontend as the clear winner. After only 2.176 seconds the page is completely loaded and rendered and is thereby more than twice as fast as OpenStreetMap, which comes in second.

The poor performance of Google and HERE can be explained by the high number of requests or the high amount of bytes that need to be transferred. 45 of Googles request are for the little blue hexagons marking the German autobahns only, even though they?re already included in the map imagery. HERE, on the other hand, accessorise their map with high resolution map imagery, tenfold the size the MARiO application receives from the HERE API. Both Google and HERE could improve their offering significantly, by

preventing unnecessary requests and reducing the resolution of map imagery, respectively.

All performance tests were conducted with WebPagetest[2], a testing framework primarily developed and run by Google. It allows the user to test performance whilst emulgating real life conditions in various locations around the globe.

The tables above are the result of 5 tests with three runs each, performed in one hour intervals on August 16 and 17, 2016. Testing location was in Frankfurt, Germany with an internet connection of 5 Mbps and a round trip time (RTT) of 28ms. All tests were performed with disabled caching. Allowing the applications to cache resources reduces both the time until the document is complete and until it is fully loaded to 40 to 60% of the original values.

4.4 Implemented Features

This subsection will contain all implemented features, describe them from a usability and technical standpoint and uncover shortcomings which will be an essential part of chapter 6. The original features this thesis was constructed for, namely the calculation of routes using various algorithms and the calculation of intermodal routes including public transportation will be explained in 4.4.2 and 4.4.3. The remaining features are the points of interest search (see 4.4.1), geocoding addresses (see 4.4.4) and the animated display of simulations of cars and thunderstorms (see 4.4.5).

4.4.1 Points of Interest

A feature that was not originally intended to be implemented in the project, which was derived from the use case of displaying open parking spots at a given location, is the search for points of interest. Because there is no real data about parking spots available yet, this would have been implemented using simulation data from a different project at the DBS and would have included route calculations from origin point A to a parking spot near the destination point B [5]. As the display of parking spaces on a map has common elements with the displaying of any points of interest at any given location, the Point of Interest search was implemented (see figure 4.4). This paragraph will describe the implementation of the feature and will identify technical difficulties and shortcomings.

To use this feature the user has to set one marker on the map to specify the location he is interested in. If two markers are set or a route is displayed both the route and the first marker are removed from the map once the Go! button is pressed. The user is able to select various categories like 'Eat & Drink', 'Sights & Museums', 'Shopping' and many more and even combining more than one category, to choose which information he is interested in. The data will then be displayed as small dots on the map with a popup opened by clicking on them, containing name, category and vicinity.

The technical integration of this feature was achieved by using one of the services of the HERE-API, the 'Places' search. The request sent to the API has to include most importantly the location in latitude and longitude, the categories one is interested in, the search radius and a few other attributes mainly concerning the response type and format.

[2] www.webpagetest.org

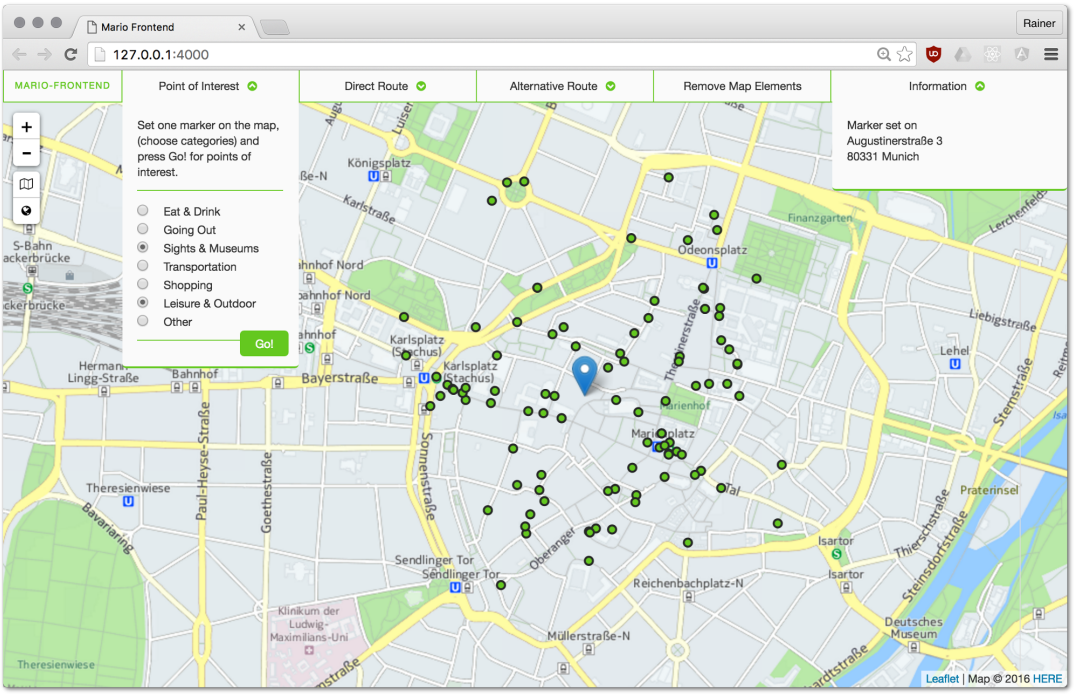

Figure 4.4: Frontend with points of interest and open dropdown

If no category is selected by the user and consequently no selection is sent to the API, a list of points of interest in all categories is returned.

Up to twenty points of interest are included in one response. When more than twenty points of interest are available in the surrounding of the location a link to twenty more points is provided. This pattern of responses allow for up to five responses and therefore a maximum of 100 points of interest for one request, which are automatically added to the map. Leaving all categories unselected about 90 percent of the responses are associated with the 'Eat & Drink' category. Fortunately the response includes the corresponding category, the name of the point of interest and the physical address. Because of the included address this feature, for a short period of time, was evaluated for the in 4.4.4 introduced reverse geocoding API but was dismissed for the lack of sufficient data in rural areas.

The performance of the 'Places' endpoint was the most unreliable of the HERE services used in this project. In the majority of cases, the response time was around 300 ms for the first request to be answered and about 200 ms for additional points of interest via the above mentioned response pattern. In very few cases the performance was exceptional, receiving five responses in about one second and sometimes even less. On the other hand there were more exceptions leaving the user to wait for up to 15 seconds until all points of interest were rendered on the map.

Similar to the above mentioned route calculations to a parking spot one could implement an algorithm which calculates a route traversing a given number of points of interest on the way. Both topics will be discussed in chapter 6.

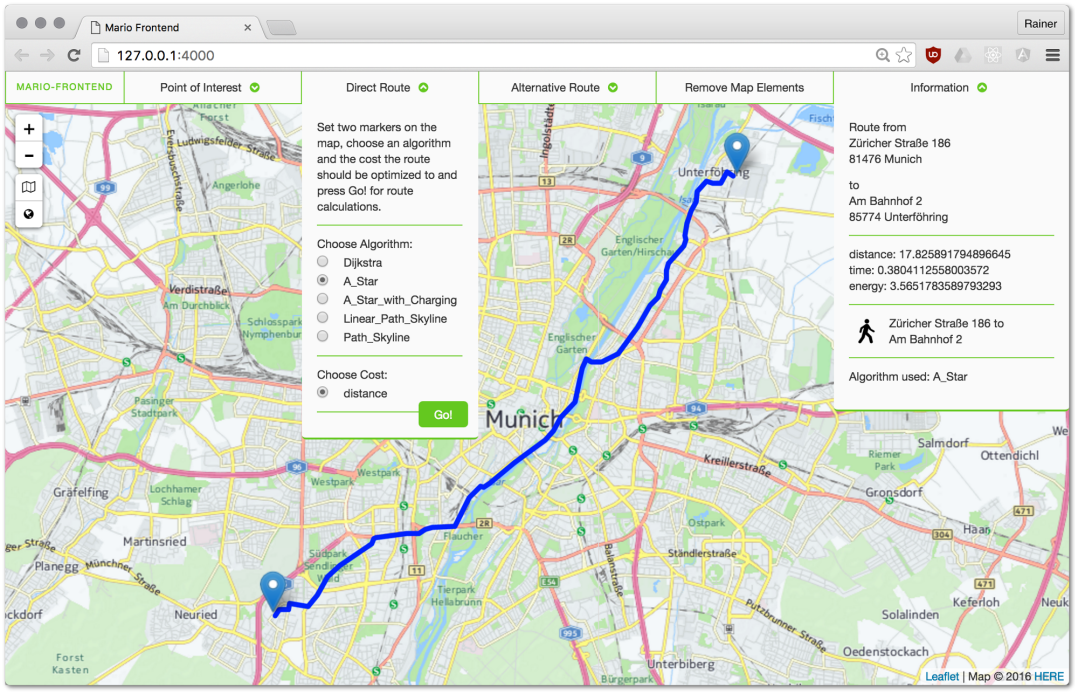

Figure 4.5: Frontend with calculated route and open dropdown

4.4.2 Route Calculations

The first of the two main features of this project is also the first feature to contact the server at DBS described in chapter 3. It enables the user to request the calculation of a route between previously defined origin and destination from the server which in turn is displayed on the map with additional information available on the information panel. Via a dropdown the user can choose one of the in 3.1 defined algorithms and additionally allows the selection of the costs, which the route should be optimised to (see figure 4.5).

As new algorithms and new implementations are frequently introduced to the server by other projects, partly resulting from other students writing their theses at the DBS, a mechanism was needed to keep the information about algorithms up to date. For that reason an additional API call was planned to be triggered to the backend, requesting a file containing all implemented algorithms and their attributes. All new algorithms implemented on the server would have to be included in that file for the frontend to make them available for selection in the above mentioned dropdown. Unfortunately this functionality was not implemented on server side, which is why it is simulated by loading a file with predefined algorithms from the deployment server.

Once the user has set two markers for origin and destination on the map and selected algorithm and costs, a request containing all information is send to the server for the calculation of the optimal route. The route is than sent back to the frontend in GeoJSON format and added to the map. Right after the route is added to the map a popup is bound to it containing the addresses of origin and destination. The route can be highlighted on hover over the route itself or the route description in the information panel. Both features are especially helpful for intermodal route calculations introduced in 4.4.3.

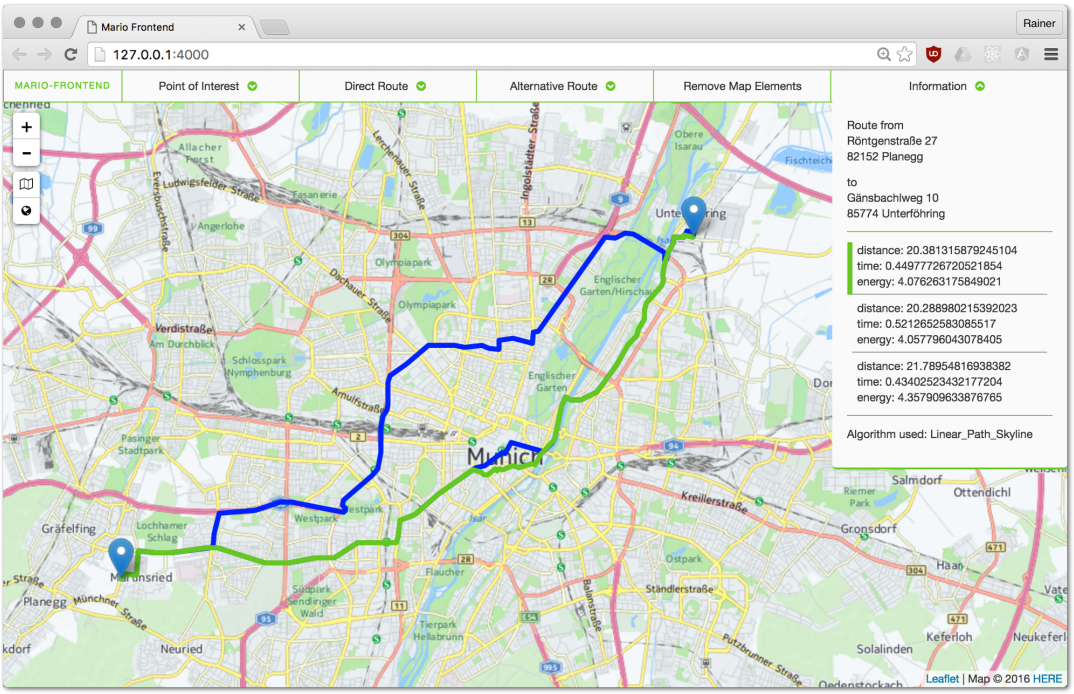

Figure 4.6: Frontend with highlighted route

At this moment, there are two cases receiving different treatment than described above. On of them is the algorithm called a-star-with-charging which was implemented for electrical vehicles, as described in 3.2. The response to a request with this algorithm selected contains a route with a charging station on the way and a point, where the station is located. The route is added as above and the charging station is added as a special marker to the map.

The second case comprises all algorithms from the skyline family. As delineated in 3.1 the skyline algorithms are able to optimise route calculations to more than one cost. That's why the user is able to select more than one or even all of the available costs in the direct route dropdown. The server returns a collection of routes in geojson format with the associated costs attached to the features. Adding the costs to the information panel of the frontend instead of the addresses of the origin and destination allows the user to compare different routes in regard to their efficiency. The above mentioned route highlighting helps to differentiate between the displayed routes (see figure 4.6).

One additional feature would have been implemented for further analysis of the algorithms, if time would have allowed it. The displaying of visited nodes in the graph to better understand how the algorithms work will be described in chapter 6.

4.4.3 Intermodal Route Calculations

The second main feature is the calculation of intermodal routes using public transportation such as trains and busses. This service is experimental and currently only available for the Berlin metro area. Similar to route calculations in 4.4.2 the user has to set two markers on the map for origin and destination. Because the intermodal route calculations are executed using an implementation of Dijkstra's algorithm optimised to time as cost,

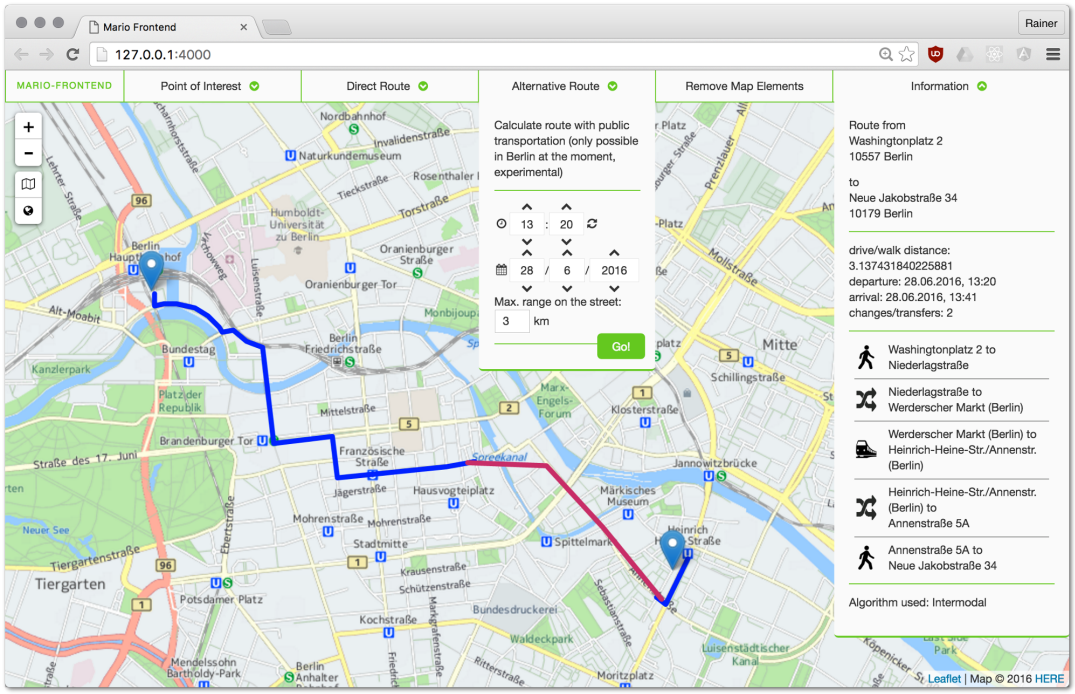

Figure 4.7: Frontend with intermodal route and open dropdown

the user is neither able to select algorithms nor costs. Additional to origin and destination the user has to specify the time of departure and the range in km he is willing to go by foot, bike or car. The departure time is automatically set to the current system time on page load and is updated only on page refresh to allow the user to check varieties of the route by changing the range parameter.

The response is a single route divided into at least five segments including walk or drive segments, train or bus rides and the changes of the mode of transportation. For easier identification of the mode of transportation and understanding of the instructions segments using public transportation are added in red and representative icons are added to the respective segment on the information panel (see figure 4.7). The user also receives the walking distance, departure and arrival time and the number of changes, which are also displayed on the information panel.

The performance of this API endpoint (see section 3.2) is rather uncertain and fluctuates between 400 ms and 30 seconds. This, of course, is not acceptable for any kind of client application and has to be addressed in further work on the backend implementation. Additionally, if the server is not able to find any route corresponding to the users input, no answer is sent to the client application at all, rendering it unable to determine whether the server is slow or no route was found.

In some cases, for yet inexplicable reasons, the server returns erroneous route information. As seen in figure 4.8 the displayed public transport segment of the returned route is one stop longer than it should be. The instructions on the other hand, are correct.

Albeit the unveiling of an action plan by the German Bundestag to implement the open data charter established by the G7 countries in September 2014 with the goal to make all government data and all data with public interest accessible to everyone, not many agen-

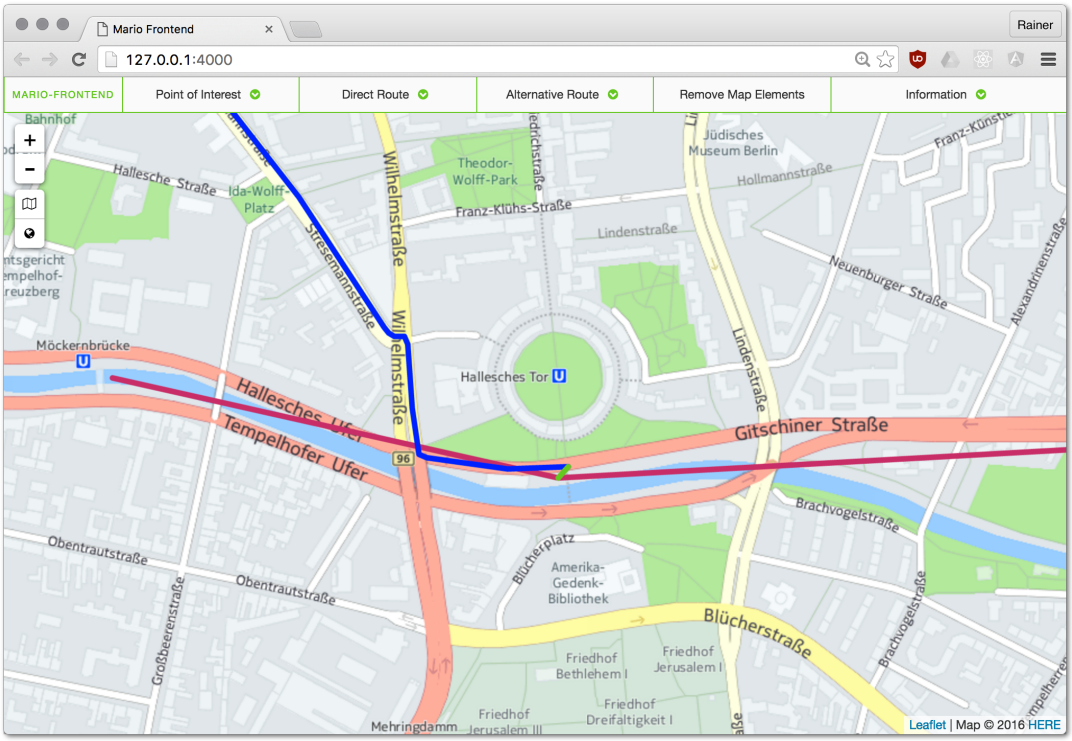

Figure 4.8: Frontend with erroneous intermodal route

cies and companies follow these rules. Although it is highly interesting for the public, one of the only providers of public transportation data is the state Berlin, which offers an API to access departure times, route information, etc while also allowing the download of bulk data sets. The provided data follows the General Transit Feed Specification (GTFS).

For more control over the results and reproductivity the bulk download method was chosen for this project leaving the maintainer to manually download the data from the official source and upload it to the server at the DBS. Because of the enormous file size only two month of data can be held on the servers memory at any given time. This is also an issue that needs to be addressed in future work on the backend implementation.

4.4.4 Geocoding Addresses

For better understanding of route descriptions and set markers a reverse geocoding API was integrated into the project to translate latitude and longitude data into physical addresses (see figure 4.9). This paragraph will outline the implementation of this feature, describe the usage of the HERE-API as reverse geocoding service, compare it to other geocoding services and highlight key features and problems encountered in the development process.

When a marker is set on the map by clicking, two processes are set into motion. Firstly, the image of the marker is loaded and pinned to the location the user clicked on and secondly, the position of the click is saved to the scope in latitude and longitude. In the bootstrapping phase of the application, that is the loading of the core functionality of AngularJS into the HTML document, a watcher is defined to listen to changes in the marker

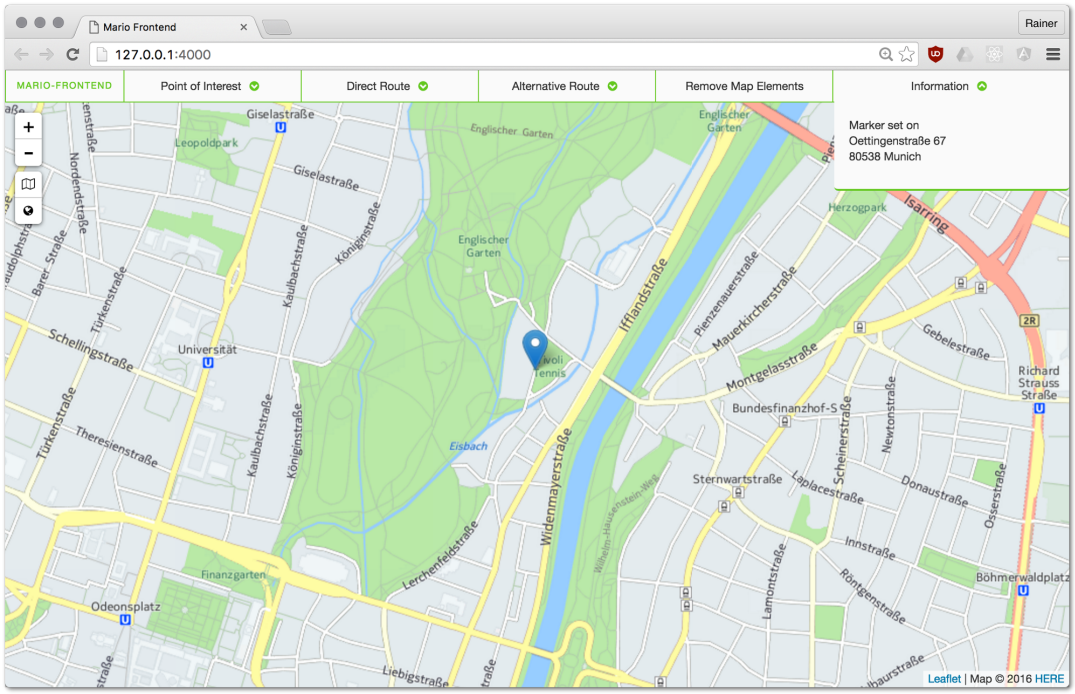

Figure 4.9: Frontend with marker and displayed address

object of the scope. For every change in this object the new or updated location of a marker is sent to the reverse geocoding API and the physical address is sent back to the application, which is than saved to the scope for display. The two way data binding of AngularJS comes in really handy in this instance because otherwise a very complex and complicated process would have to be implemented to generate that functionality.

This feature increases its value in the in 4.4.3 introduced intermodal route calculations. As you know, the route is calculated to include public transportation which implies that the route is split into segments. Each segment has an origin and a destination and for every one of these points the physical address is requested from the reverse geocoding API and displayed as a from-to statement in the route information panel of the application.

The basics of the HERE-API are described in Chapter 2.4 so only the specific properties of the reverse geocoding API will be illustrated in this section. As all other parts of the HERE conglomeration, the geocoding API is only well documented to a certain extend, for the documentation is so bulky that one is better off just implementing and testing it. The error descriptions and actual data responses are far better to understand the workings of the API than just reading the documentation. For historic reasons and to satisfy a similar data structure in all HERE APIs the response data to a reverse geocoding request is strictly hierarchical and multilayered. Especially in this case alternatives to the HERE API such as Nominatim, maintained by the OpenStreetMap Foundation and OpenCage Geocoder run by the OpenCage Data Ltd. would have been preferable but both couldn't be used because of the strict limitation to one request per second.

Implementing the API one can define the mode of the request. For translating latitude and longitude intro physical addresses the modes 'retrieve addresses' and 'track position' are applicable, although the second one is a bit alienated for that purpose. 'Retrieve

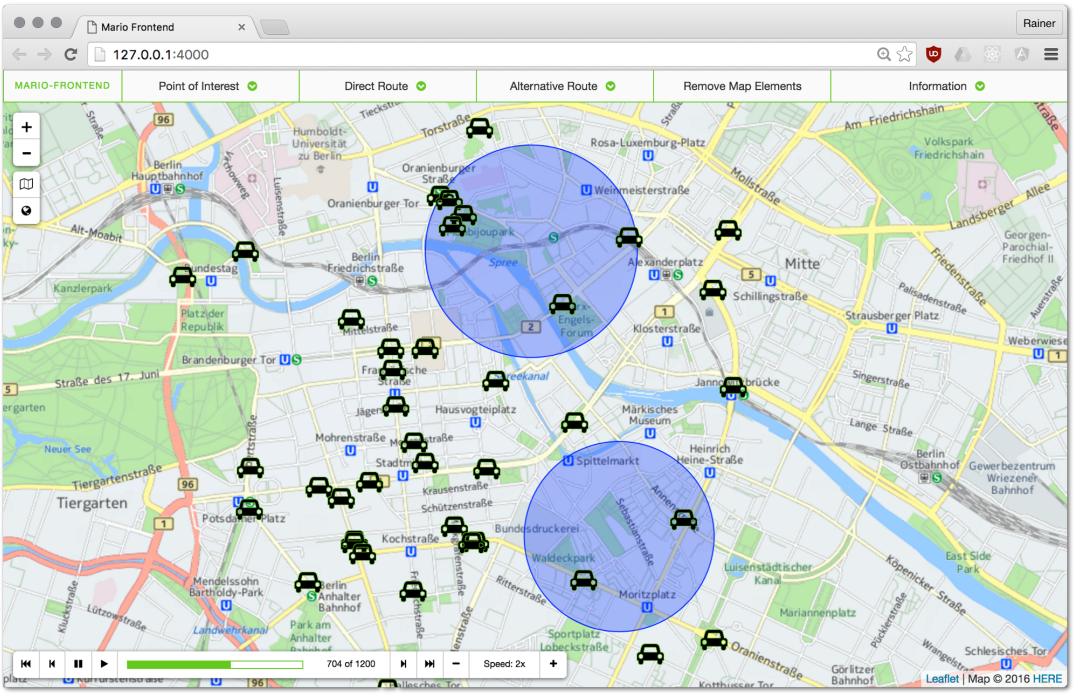

Figure 4.10: Frontend with running simulation of cars and weather

addresses' returns a number of addresses, specified by the developer, which are ordered by the distance to the requested location. For an inexplicable reason in some cases the state or city is returned as nearest address without any information about street or house number. Accordingly the results had to be filtered to satisfy the need for a correct address. 'Track position' is intended to be used for mobile navigation as it incorporates the direction in which the user is moving for the calculations usually returns the better results because it snaps to the nearest address it can find and returns just that. Unfortunately this mode is not reliable in more remote areas and returns no address at all in some cases.

For this project the more dependable mode 'retrieve addresses' was chosen and logic was implemented to select the best quality address from the five responses. The 'match-level' parameter associated with each address was helpful but not reliable because the above mentioned addresses, only containing city or state, sometimes also have the highest match level possible, which is probably a wrong implementation on the providers side. Again, other geocoding APIs would have been preferable but couldn't be used because of rate limitations. Notable is the fast response time of the API which almost never exceeded 100 ms and was thus on average two times faster than Nominatim and almost four times faster than OpenCage Geocoder.

4.4.5 Simulation

Algorithms and methodologies developed at the DBS have the tendency to be abstract and hard to grasp, due to the fact that they are conceived in a highly theoretical and methodological higher education environment. This is especially true for simulations with high data output like the one integrated in this project. Jenny Lauterbach develops and implements a system to simulate vehicles driving through a street network and how they are

influenced by external factors, such as heavy weather simulated as circular storms moving across a simulation area. This is an ongoing project at DBS under the supervision of PD Dr. Matthias Schubert. The next few paragraphs will not describe the developed system but the implementation of simulation data into this project's frontend.

As a simulation of this kind adds time as an additional dimension to this project, a control panel had to be implemented to navigate through the up to 3600 states that where generated by the above mentioned system. This panel (see bottom left corner of figure 4.10) includes play and pause buttons, step forward and backward and jumps forward and backward. Additionally the user is able to jump to a specific part of the simulation by clicking on the control bar. Later in the project the simulation speed control was introduced to specify the play-back speed between 1/4 and four times the normal speed.

Currently the user is not able to select an area for the simulation as there is no web service or API provided by Jenny Lauterbach's project. The temporary workaround is to generate simulation files, converting them to GeoJson, adding them manually to the project and later load them from the web server, the frontend is hosted at. At this point this workaround is probably the only way to display simulation data produced by the system because generating the file takes up to several minutes, depending on the number of cars and storms and the simulation duration.

The GeoJson file is structured into cars and storms which are grouped as GeoJson-Features according to their timestamp. Each vehicle has a position in latitude and longitude associated to it which makes it easy to add them as markers to the map. The icon of a car is used as marker image to clarify what they symbolise. Storms on the other hand have to be treated otherwise since they have a spatial component. These are added as so called circular paths with the radius provided in the file. On first load of the simulation data the first state of cars and storms, identified by the timestamp of 0, are added to the map.

The animation of moving elements is essentially achieved by removing all vehicles and storms and adding the ones of the next step. These, again, are identified by the timestamp associated with the data. On normal speed, the scene is repainted about twenty times per second and can be slowed down to five and speedup to about eighty repaints per second. In a few instances a slight flickering of the storms was noticed but unfortunately it was neither possible to detect the source of the error nor to determine the specific system setup the led to it.

4.5 Legal Basis and Copyright

Using third party code and projects brought a certain legal complexity into this project concerning licenses and rights to use. Several projects, including AngularJS and bower, are published under MIT, others under more restrictive licences. The following section will introduce the for the software developed for this thesis relevant licences and summarise legal considerations in general.

Copyright law dates back to the early 18th century and grants the creator of an original work the exclusive right for use and distribution. This exclusive right is granted for limited time (70 years after the creators death or 120 years after creation for not identifiable creators) and only protects the product itself and not the general idea behind it. In software

development copyright law is highly relevant "[b]ecause commercial software is made, as a general matter, by large teams of people and requires the substantial expenditure of capital, the resulting work is 'work for hire'" [8] and the copyrights therefore belong to the corporation commissioning and founding the development process. Since the beginning of the open source movement in the early 90th, developers needed and subsequently wrote open source licenses. The three most important and most widely used licences will be introduced in the next paragraphs.

As of 2015 the most popular open source software licence used in GitHub projects is the MIT license authored and published by the Massachusetts Institute of Technology [13] which was also used to licence the software of this project. According to the Black Duck Software Inc. MIT is also the most used licence for open source software in general, followed by GNU GPL and Apache License [14] which will be explained in the next paragraphs. "The 'BSD-like' licenses such as the BSD, MIT, and Apache licenses are extremely permissive, requiring little more than attributing the original portions of the licensed code to the original developers in your own code and/or documentation" [23].

One of the first licenses granting the enduser the right to run, share and modify source code was the GNU General Public License (GPL), written by Richard Stallman in 1989 for the use in the GNU project. It originated from previously written licenses for specific parts of GNU that were not transferable to other projects. Stallman had the intention to write a general licence, which than would be applied to many projects without specification, allowing them to share code without a breach of licence agreements [29]. GPL specifically allows the licensee to redistribute even derivative versions and is able to charge for his service. This allowance of commercial use separates the GPL from several other licences. Several sub dependencies of this project are published under GPL version 2.0 and version 3.0.

The Apache Software Foundation (ASF) is a non-profit, donation founded cooperation overseeing more than 350 leading open source projects including the Apache HTTP-Server, Apache Cassandra and Apache OpenOffice. The ASF published a bundle of licenses to accept and redistribute contributions from individuals or cooperations and grants of existing software products. On of these licenses is the Apache License (ASL) which is, as the two licences above, a permissive free software license [23]. The ASL was originally intended to be used solely for ASF projects and products but is now, since the update to version 2.0 in 2004, also used for unassociated open source projects.

For further information on licensing of open source and copyright law Andrew Laurent's "Understanding Open Source and Free Software Licensing" [8] is highly recommended further reading.

Leaflet and HERE demand attribution of their use in any software project, which is given in the bottom right corner of the map application. HERE also imposed several other legal requests, such as the in 4.4.1 mentioned restriction to only use the Places service on HERE map imagery. Legal considerations were one of the main reasons to rely on one single source of services rather than several.

5 Modularity

The idea behind the architecture of the MARiO project, as described in section 3, was a plug in system, which also can be described as a modular approach. To add new functionality a new module is developed and added to the code base without altering it. But what are modules and what is modularity in general? The next sections will introduce modularity as an important software engineering principle and subsection 5.2 will take a closer look at modularity in modern web development and it's use in this project.

5.1 Modularity in Software Development

'Separation of concerns' is a much used key word in object oriented programming and incorporates a 'divide and conquer' (d&c) approach, which is a basic technique for problem solving in computer engineering. The idea behind d&c is to split a problem into smaller sub problems, solve the sub problems individually and combine the results to a solution for the original problem. Many algorithms for various problems have been implemented following that principle. Separation of concerns however is a more general principle allowing for more heterogenous problems.

Modularity is a practical implementation of the separation of concerns as it allows to split a complex system into smaller and simpler modules. Imagine a standard coffee maker as an example. The coffee maker consists of various modules - pot, filter, water heater, etc. - each with its own responsibilities. Separating them makes it easier to think about the functionality of each module and facilitates enhancements but only together they can work as intended and brew coffee.

The same can be said about software development. The model view controller pattern described in section 2.1 and 4.2 is an example for a high level separation of concerns. View, model and controller are strictly separated and only join forces at previously defined anchor points. But the separation can be driven much further, splitting the view in smaller components and detaching functionalities and features to form services for example.

"Modules are technically connected to one another. The measure of inter-module relation is known as coupling. Design goals require modules to have low-coupling and high cohesion. Cohesion is a measure of the inter-relatedness of elements (statements, procedures, declarations) within a module. A module is said to have high cohesion if all the elements in the module are strongly connected with one another. Tight coupling of modules makes analysis, understanding, modification and testing of modules difficult. Reuse of modules is also hindered." [28].

Almost all software design patterns implement some kind of modularity. The MVC pattern, as previously mentioned, is the obvious example, but several others do it as well. The Decorator pattern for example permits to add functionality to classes without the need of subclasses, allowing isolation of core functionality inside. The Mediator Pattern specifically promotes lose coupling by offering a central place of communication between objects without the need of referencing each other directly. A pattern which perfectly implements modularity is the Composite pattern. Objects are composed into tree structures forming whole-part relationships, as seen in a class diagram in figure 5.1, like the coffee maker example [2].

As shown in the last paragraphs, separation of concerns or modularity is omnipresent in software development in an object oriented environment. And it's not a bad thing either.

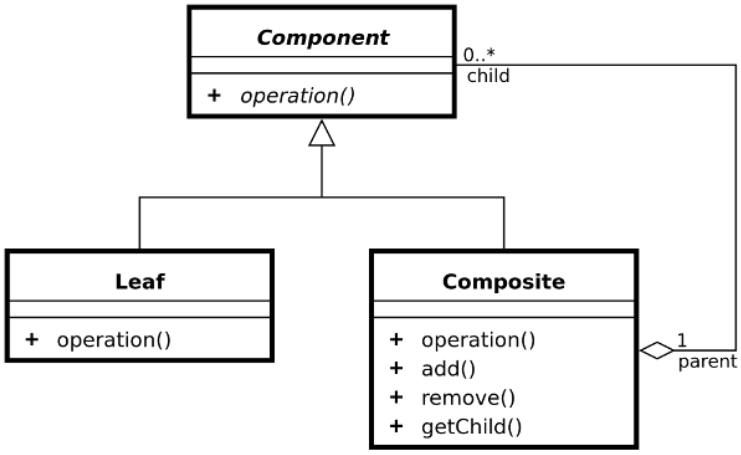

Figure 5.1: Class diagram of a Composite pattern

The developer is enabled to enhance small parts of his code without the need to alter the whole project. The partitioning of projects into smaller, independent parts also facilitates software development in teams, even if they work remote. The communication between modules has to be agreed upon and it should be clear which functionality a module needs to add to a project, but other than that the developer is free to implement the module the way he wants to.

5.2 Modularity in modern Web Development

Two types of modularity have to be considered in this section. One is modularity from a web design point of view, dividing a design into atomic elements and building web sites by combining these elements. This will be shortly introduced before considering the second point, modularity resulting from the in subsection 4.1.2 introduced package managers like npm and bower.

Web applications can be divided into elements, like navigation, forms, buttons, etc. Each of these elements can be further split down until you reach so called atomic elements. Taking these elements and combining them to components, combining components to pages and pages to web sites without loosing the independence of the atomic elements is called modularity in web design. The best implementation of this approach is the Block-Element-Modifier (BEM) methodology, where blocks are independent entities, simple or compound, elements are context-dependent parts of blocks performing a certain function and modifiers are properties to change style and behaviour of blocks or elements. BEM offers a framework for styling of small components to than add them to bigger ones without loosing the ability to style the elements independently. Bootstrap is a well known implementation of BEM offering predefined objects to use for web design. The advantage of this approach is the reusability of components.

Reusability is also one of the main reasons why package managers like npm and bower gained in popularity. NodeJS is a framework to use JavaScript in a backend environment.

As JavaScript is primarily used in frontend development, NodeJS is favoured by many frontend developers over other technologies. NodeJS's package manager npm is an administrative tool for dependency management, which can also be used as a build tool in software development by implementing scripts to build and run software. Npm also gives access to the largest ecosystem of open source software in the world.

Npm packages are mostly small components that developers can integrate into their projects without adding much complexity. Most open source frameworks, like AngularJS and Leaflet, offer a integration via npm or other package managers to ease the development process for developers using it. The result of this method is a highly modularised, tree like structure because this packages also have dependencies to other projects as well.

Taking on dependencies of third party code can be problematic as well. In March 2016, many of high profile npm packages like React, Node and Babel broke because one programmer wasn't happy with how npm has handled a legal dispute between him and a company requesting the name of one of his packages. After npm granted the packages name to the requesting party he unpublished all of his packages from the npm registry including one called left-pad. Left-pad is a module consisting of 11 lines of code to left pad a string with a certain character. Thousands of projects had left-pad as an indirect dependancy which, after unpublished, wasn't available for download anymore [30].

Having a look at the source code written in the course of this thesis again, dependencies were kept to a minimum to avoid such problems. Other than MVC modularity can be seen in the structure of the HTML files as well as the controller-services structure (see figure 4.1). The dropdown menus are all implemented in their own file to separate them from each other. The Alternative Route dropdown even has different files for each of the alternatives displayed.

The advantage of this approach is the easy extensibility of the project. A new feature just needs a HTML file for the dropdown and a JavaScript service implementing the needed logic wired into the controller. As mentioned in subsection 4.4.1 about the point of interest search, open parking spots and route calculations to a parking spot near the destination point could be implemented in the future (see section 6). For the purpose of easy integration parking.html and ParkingService.js were created as examples as how to integrate a new feature into the project.

6 Recapitulation and Future Work

This thesis introduced the reader to various components of the MARiO project operated by the DBS at LMU. MARiO is a open source mapping application written in Java which allows the scientists and researchers at the DBS to implement new algorithms and methodologies regarding graph data and street networks. As MARiO was implemented in 2011 and is thereby already outdated, MARiO was refined to a modern server-client architecture where the backend server can be accessed via a REST API. In the course of the software development accompanying this thesis a browser mapping application was implemented enabling users to access this API to request route calculations with various algorithms. The server returns the calculated routes and additional route informations to the frontend, where it is displayed in a user friendly way.

JavaScript framework AngularJS in combination with mapping library Leaflet were used as frontend technologies, which ensure fluid usability and a great user experience through AngularJS' bidirectional data binding and Leaflets simplicity and performance. After giving a detailed description of AngularJS and a explicit introduction to Leaflet, GeoJSON was established as encoding for storage and transmission of geospatial data, for its nativity in JavaScript and, more important, its direct parseability to Leaflet objects. As an external data source the HERE API plays an important role in this project by providing map imagery and permitting the superinduction of additional features.

The application was developed following a MVC separation and a modular approach, which allows, similar to the original MARiO project, the implementation of supplemental features, without altering the code base. The user is enabled to request the calculation of routes using various algorithms implemented in the backend, including intermodal route calculations.

A point of interest search was implemented allowing the user to search for interesting places around a set marker. Various categories like 'Eat & Drink' and 'Transportation' are available to narrow the search. Points of interest are added to the map as circular markers with name, category and address of the venue, available in a popup overlay. For more intelligible route instructions and better user experience, latitude-longitude positions returned by the backend as well as created by clicks on the map, are geocoded to physical addresses. Both features were implemented using HERE APIs.

In parallel to this thesis, Jenny Lauterbach developed and implemented a system to simulate vehicles driving through a street network influenced by external factors. These simulations are integrated in the new MARiO project as a visualisation for testing and analysis and can be speed up or slowed down for replay.

Throughout this thesis various fields of future work are identified and will be described in the next few paragraphs. Some of them are easy to implement, others require a great deal of research and work, although the majority of topics are not possible to implement on frontend side alone, but require the deployment of functionality on the backend.

The feature with the most impact on testability, mostly regarding the comparison between algorithms, would be the display of by the algorithm visited nodes in the graph. All algorithms calculating shortest paths in a graph, visit nodes to determine whether it is usable for a route or not. Nodes in map data are, for example, intersections and crossings. The amount of nodes visited by an algorithm is an indicator for its complexity and performance. Additionally it would allow the discovery of an erroneous implementation. For

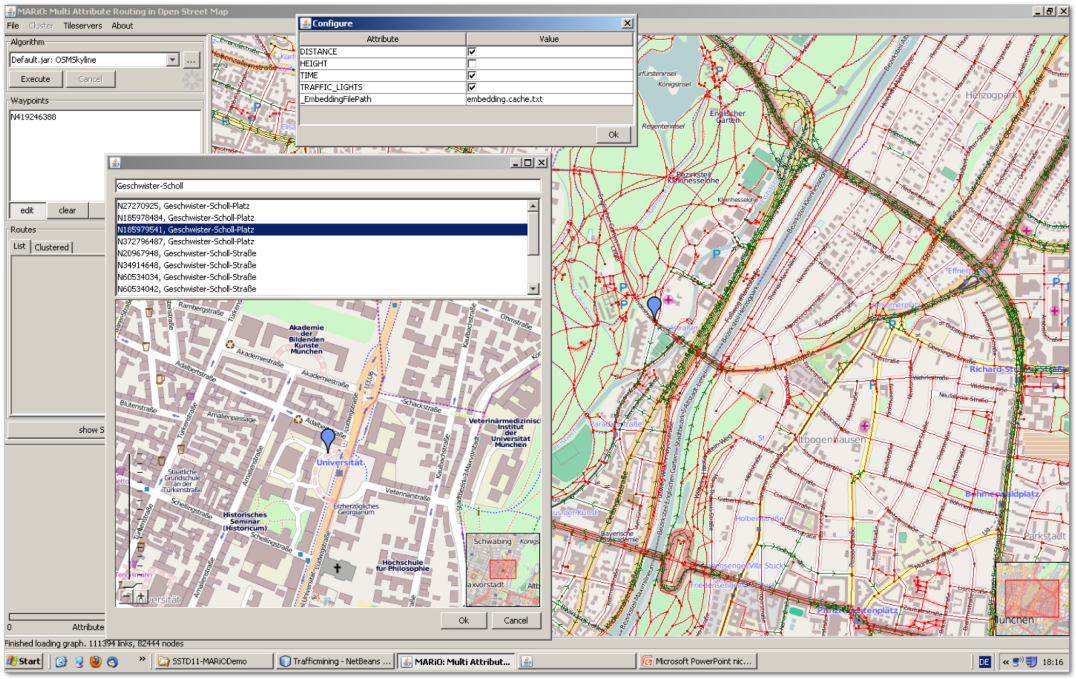

Figure 6.1: Diagram of the application structure

that purpose dark grey, unlabelled map imagery is already prepared in the project. To implement this functionality, the frontend would have to send a flag, indicating whether the visited nodes are requested or not, with its route calculation request and the server would have to send back a list of the location of these nodes. Once the frontend receives the data it can check for the existence of visited nodes and display it to the user accordingly.

Various publications of the DBS qualify themselves to be implemented in the MARiO project. Two of them where already mentioned in subsection 4.4.1 about the points of interest search. A driver usually needs a parking spot in the vicinity of his destination, which can prove quite difficult in unfamiliar urban surroundings. As proposed in [5], an algorithm could be implemented, allowing the considerations of unoccupied parking spots in route calculations. Similar, as proposed in [4], tourists could include points of interest in their route calculations to get the most out of a drive through a visited city. Both features would have to return the route and the parking spots and points of interest, respectively, to the frontend where they could be displayed as a combination of the points of interest and the route calculations already available in the developed frontend.

The intermodal route calculation implementation is, as of yet, limited in its applicability because it is restricted to Berlin and a lot of manual work is necessary to operate it on backend side. Once the data is available other cities could be integrated into the service. Additionally an API on the providers side would be more than welcome to reduce the manual labour needed to preprocess and upload the data to the backend. As this is mostly dependent on third party entities, a implementation is not possible at the moment. A quite easy implementable feature, which was already available in the old MARiO frontend, is an address search functionality. To ease usability the user could search for addresses, which would than be marked on the map. Multiple results would have to be displayed in a list. The implementation in the old frontend can be seen in figure 6.1. The implementa-

tion of this functionality would only require an input field as a search bar and the ability to display a list of results. The HERE API for geocoding could be used to translate physical addresses to their associated location.

Both AngularJS and Leaflet will release a major version update in the coming weeks. As of now, AngularJS 2 is at release candidate 5 and Leaflet at release candidate 3. Updating AngularJS from 1.5.5 to 2.0 would further improve the maintainability and performance. Updating Leaflet from 0.7.7 to 1.0 would improve mobile friendliness and performance.

List of Figures

List of Tables

References

[1] Crickard III, P., 2014: *Leaflet.js Essentials*. Packt Publishing Ltd.

[2] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1994: *Design Patterns*. Addison-Wesley.

[3] Graf, F., Kriegel, H.P., Renz, M. and Schubert, M., 2011: MARiO: multi-attribute routing in open street map. In *International Symposium on Spatial and Temporal Databases* (pp. 486-490). Springer Berlin Heidelberg.

[4] Jossé, G., Schmid, K.A., Züfle, A., Skoumas, G., Schubert, M. and Pfoser, D., 2015: Tourismo: A User-Preference Tourist Trip Search Engine. In *International Symposium on Spatial and Temporal Databases* (pp. 514-519). Springer International Publishing.

[5] Jossé, G., Schubert, M. and Kriegel, H.P., 2013: Probabilistic parking queries using aging functions. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (pp. 452-455). ACM.

[6] Jossé, G., Schubert, M. and Zellner, L., 2015: EasyEV: Monitoring and Querying System for Electric Vehicle Fleets Using Smart Car Data. In *International Symposium on Spatial and Temporal Databases* (pp. 497-502). Springer International Publishing.

[7] Kriegel, H.P., Schubert and M., Renz, M., 2010: Route skyline queries: A multi-preference path planning approach. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (pp. 261-272). IEEE.

[8] Laurent, A., 2004: *Understanding Open Source and Free Software Licensing*. O'Reilly Media.

[9] Schwaber, K., 1997: Scrum development process. In *Business Object Design and Implementation* (pp. 117-134). Springer London.

[10] Shekelyan, M., Jossé, G. and Schubert, M., 2015: Linear path skylines in multicriteria networks. In *2015 IEEE 31st International Conference on Data Engineering* (pp. 459-470). IEEE.

[11] Teixeira, P., 2012: *Professional Node.js: Building Javascript based scalable software*. John Wiley & Sons.

Web-References

[12] Agafonkin, V., 2015: *Leaflet story in 13 minutes*, https://youtu.be/NLbyHffKQuU, accessed July 18, 2016

[13] Balter, B., 2015: *Open source license usage on GitHub.com* https://github.com/blog/1964-license-usage-on-github-com, accessed August 2, 2016

[14] Black Duck Software, Inc., 2015: *Top 20 Open Source Licenses*, https://www.blackducksoftware.com/top-open-source-licenses, accessed August 8, 2016

[15] BusinessWire, 2007: *Rapid Subversion Adoption Validates Enterprise Readiness and Challenges Traditional Software Configuration Management Leaders* http://www.businesswire.com/news/home/20070515005055/en/Rapid-Subversion-Adoption-Validates-Enterprise-Readiness-Challenges, accessed July 26, 2016

[16] Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T., Schmidt, C., 2008: *The GeoJSON Format Specification*, http://geojson.org/geojson-spec.html, accessed July 20, 2016

[17] DeBill, E., 2016: *Module Counts*, http://www.modulecounts.com/, accessed July 29, 2016

[18] ECMA, 2016: *Standard ECMA-262 ECMAScript Language Specification, 7th edition*, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, accessed July 18, 2016

[19] ECMA, 2016: *Standard ECMA-404 The JSON Data Interchange Format, 1st edition*, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, accessed July 19, 2016

[20] MacWright, T., 2013: *Announcing MapBox.js 1.0 with Leaflet*, https://www.mapbox.com/blog/mapbox-js-with-leaflet/, accessed July 18, 2016

[21] MacWright, T., 2015: *More than you ever wanted to know about GeoJSON*, http://www.macwright.org/2015/03/23/geojson-second-bite.html, accessed July 20, 2016

[22] Metz, C., 2015: *How GitHub Conquered Google, Microsoft, and Everyone Else*, http://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/, accessed July 25, 2016

[23] New Media Rights, 2008: *Open Source Licensing Guide* http://www.newmediarights.org/open_source/new_media_rights_open_source_licensing_guide, accessed August 2, 2016

[24] Nokia Communication, 2007: *Nokia to acquire NAVTEQ*, http://company.nokia.com/en/news/press-releases/2007/10/01/nokia-to-acquire-navteq, accessed July 25, 2016

[25] Novet, J., 2016: *GitHub changes pricing: Unlimited private repos cost $7 per month for personal accounts, $9 per user per month for organizations* http://venturebeat.com/2016/05/11/github-changes-pricing-unlimited-private-repos-cost-7-per-month-for-personal-accounts-9-per-user-per-month-for-organizations/

[26] OGC, 2014: *OGC Web Feature Service 2.0 Interface Standard*, http://www.opengeospatial.org/standards/wfs, accessed July 18, 2016

[27] OGC, 2006: *OpenGIS Web Map Server Implementation Specification*, http://www.opengeospatial.org/standards/wms, accessed July 18, 2016

[28] Poo, D., 2008: *Seven Strategies for Successful Software Engineering: Part 2*, http://www.brighthub.com/computing/windows-platform/articles/10177.aspx, accessed August 9, 2016

[29] Stallman, R., 2006: *Richard Stallman at the 2nd international GPLv3 conference*, http://fsfe.org/campaigns/gplv3/fisl-rms-transcript.en.html#before-gnu-gpl, accessed August 2, 2016

[30] Williams, C., 2016: *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*, http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/, accessed August 13, 2016