Central Frontend for Segment Routing Applications

Bachelor Thesis

Department of Computer Science
OST – University of Applied Sciences
Campus Rapperswil-Jona

Spring Term 2022

Author(s):	Davor Gajic, Leonard Obernhuber
Advisor:	Prof. Laurent Metzger
Co-Advisor:	Severin Dellsperger
Project Partner:	Cisco represented by Bruce McDougall
External Co-Examiner:	Laurent Billas FUB
Internal Co-Examiner:	Stefan Keller

Central Frontend

for Segment Routing applications

Graduate

Davor Gajic

Leonard Obernhuber

Initial Situation: Many segment routing (SR) applications are developed and maintained at the INS and each of those applications has to create its own UI, usually from scratch. The implementation of these user interfaces is never the main focus during the development, resulting in additional effort for each SR app project and reimplementation of existing components without maintaining a unified look and feel for each frontend.

Objective: The goal is to create a centralised user interface that on the one hand dynamically connects and launches existing SR apps, and on the other hand makes the process of creating new user interfaces in a unified way very accessible. With modularity and reusability of the components in mind, we want to create a process as well as a product to create and maintain the ecosystem.

To solve this, a Micro Frontend approach should be implemented, that allows for certain components to be hosted on a standalone web server and being rendered in the central web application.

The Central Frontend for Segment Routing Applications should contain a landing page that renders and clusters up to a thousand nodes in less than 10 seconds onto a geographical map. These performance requirements are achievable by using a graphical processing unit (GPU) supporting graph plotting framework. The landing page should be interchangeable with any other Micro Frontend.

Printed graphs should support a point clustering mechanism that groups nodes into clusters to maintain visibility depending on the zoom level of the map. The nodes and edges should be interactive and show the corresponding information coming from the Jalapeno API Gateway.

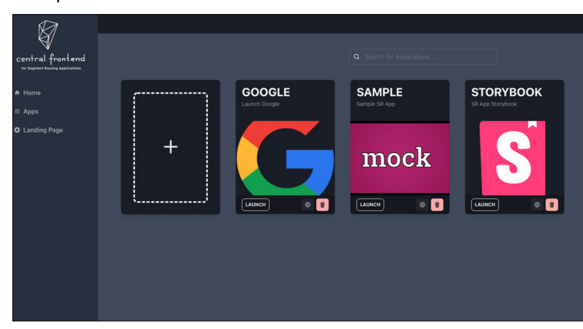
In addition, the Central Frontend should allow other SR apps to be launched and displayed inside a container on the same page.

Result: The Central Frontend has been implemented as a React frontend (with TypeScript) that communicates with a Go API to manage and display the incorporated Micro Frontends.

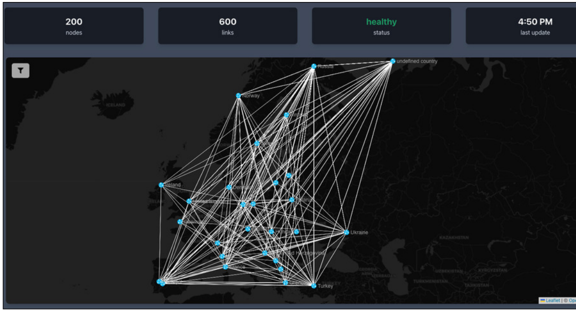
The landing page Micro Frontend communicates with the Jalapeno API Gateway via gRPC-web and handles visualisation of the network onto a Leaflet map using the framework SigmaJS which supports webGL rendering. Additionally, the landing page clusters the network topology geographically to improve the visibility and user experience.

The Central Frontend covers all defined mandatory features and use cases, and the project was a success.

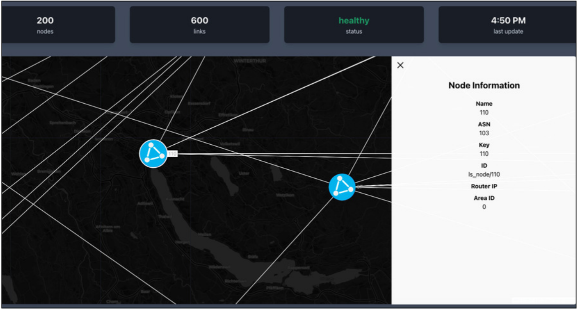
Central Frontend: Standalone and Micro Frontend List
Own presentment



Landing Page: Clustered Network Topology Graph
Own presentment



Landing Page: Unclustered Node with Informationbar
Own presentment



Examiner
Prof. Laurent Metzger

Co-Advisor
Laurent Billas, Führungsunterstützung sbasis FUB, Bern, BE

Subject Area
Internet Technologies and Applications

Project Partner
Cisco Systems, Machelen, Belgien

Management Summary

Central Frontend

for Segment Routing Applications

Authors	Davor Gajic, Leonard Obernhuber
Examinator	Laurent Metzger
Expert	Laurent Billas
Themengebiet	Segment Routing
Project Partner	Cisco Systems

Initial Situation

The Segment Routing (SR) technology brings countless possibilities for applications in the network sector which monitor or make use of this protocol.

Segment Routing Applications are developed by many different teams in the segment routing community. There are no guidelines on what the frontends need to look like, and the UI is usually rushed and just a second priority. Additionally, when deploying many UIs, it is difficult to keep an overview of all the frontends. Furthermore, developing a standalone user interface for each of the different Segment Routing Applications adds a significant amount of effort in each project. Until now there was no centralised solution available in any form.

Approach

To simplify the process of adding a new SR Application frontend to the repertoire and aggregate consisting apps, we developed the Central Frontend.

It implements the cutting-edge approach called Micro Frontends, which allows for certain components of the UI to be deployed independently and being rendered dynamically into a container application. SR developers should be able register their Micro Frontend in the Central Frontend and then users could launch the various Micro Frontends in the Central Frontend.

In addition, the Central Frontend should provide a landing page Micro Frontend which will be rendered when the Central Frontend is launched. It should render the SR network topology onto a map and draw the nodes onto their geolocation. Moreover, it should cluster the nodes together depending on the maps zoom level.

Results

The Central Frontend and its different components have been successfully created in this project.

The product allows to build sustainable segment routing ecosystems by combining all the unified apps into one single frontend and give an overview over the current network topology with a clustered graphical representation on the landing page.

Contents

1 Technical Documentation 10
1.1 Introduction and Overview . . . 10
1.1.1 Initial situation . . . 10
1.1.2 Problem description . . . 10
1.1.3 Goal . . . 10
1.1.4 Scope and limitations . . . 11
1.2 Requirements . . . 12
1.2.1 Actors . . . 12
1.2.1.1 Software Engineer . . . 12
1.2.1.2 Network Operator . . . 12
1.2.2 Use Case Diagram . . . 12
1.2.3 Use Cases Brief . . . 13
1.2.4 NFRs (Non functional requirements) . . . 14

2 Analysis 15
2.1 Clustering . . . 15
2.1.1 DBSCAN . . . 15
2.1.2 K-means . . . 15
2.1.3 Nominatim . . . 15
2.1.3.1 Zoom parameter . . . 16
2.1.3.2 Hosting Nominatim . . . 16
2.1.3.3 Decision . . . 17
2.1.3.4 Problems . . . 17
2.2 Network graph visualisation framework . . . 19
2.2.1 Scalability Tests . . . 19
2.2.2 Advantages . . . 19
2.2.3 Downsides . . . 19
2.3 Other SR Apps and Micro Frontends . . . 20
2.3.1 Mono Repo Approach . . . 20
2.3.2 API Approach . . . 20
2.3.3 Micro Frontend Approach . . . 20
2.3.3.1 Initial Research . . . 21
2.3.3.2 Micro Frontends Proof of concept . . . 21
2.3.3.3 Further research and Micro Frontends with TypeScript React apps 24
2.3.3.4 Conclusion of Micro Frontends . . . 24
2.3.3.5 Storybook proof of concept . . . 25
2.3.3.6 Conclusion of the Storybook proof of concept . . . 27
2.4 React UI Frameworks . . . 28
2.4.1 Chakra UI . . . 28
2.4.2 Material UI (mui) . . . 28
2.4.3 Decision . . . 29

3 Architecture 30
3.1 Overview . . . 30
3.2 Components . . . 30
3.2.1 Central Frontend . . . 30
3.2.2 Landing Page . . . 30

3.2.3 API . . . 31
3.2.4 Central Frontend UI Library . . . 31
3.3 External components . . . 31
3.3.1 Other SRApps/Microfrontends . . . 31
3.3.2 Nominatim . . . 31
3.3.3 MongoDB . . . 31
3.3.4 Jalapeno API Gateway . . . 31
3.4 Technologies . . . 32
3.4.1 Languages . . . 32
3.4.2 Data Serialisation languages . . . 32
3.4.3 External Libraries . . . 32
3.4.4 Browser Support (tested) . . . 32
3.5 Design . . . 33
3.5.1 Wireframes . . . 34
3.6 Deployment . . . 37
3.7 Infrastructure . . . 37

4 Design decisions **38**
4.1 Clustering . . . 38
4.1.1 Reasons . . . 38
4.1.2 Proof of concept / Usability test . . . 38
4.1.3 Data structure . . . 38
4.2 User Interface Framework . . . 39
4.2.1 Reasons . . . 39
4.2.2 Proof of concept / Usability test . . . 39
4.3 Micro Frontends . . . 39
4.3.1 Reasons . . . 39

5 Implementation **41**
5.1 Repositories . . . 41
5.2 Central Frontend . . . 42
5.2.1 Description . . . 42
5.2.2 Components . . . 43
5.2.2.1 Chakra UI (Choc UI) . . . 43
5.2.2.2 Config . . . 43
5.2.2.3 API Client . . . 43
5.2.2.4 PageController . . . 43
5.2.2.5 MicroFrontend . . . 45
5.2.2.6 Pages . . . 46
5.2.3 Docker . . . 47
5.2.4 CI CD . . . 47
5.3 Central Frontend API . . . 48
5.3.1 Persistence . . . 48
5.3.2 Model . . . 48
5.3.3 Routes . . . 48
5.3.4 SRApp controller . . . 49
5.3.5 SRApp Image controller . . . 49
5.3.6 Landingpage controller . . . 50
5.3.7 Docker . . . 50

5.3.8 CI CD 50
5.4 Helm Chart 50
5.4.1 Limitations 50
5.5 srapp-lib 51
5.5.1 Story 51
5.6 Create React App Template 51
5.6.1 config-overrides.js 51
5.6.2 Chart 52
5.6.3 microfrontend.tsx 52
5.6.3.1 API 52
5.6.4 Limitation 52
5.7 Landing Page 53
5.7.1 Components 53
5.7.1.1 SigmaMap 53
5.7.1.2 SigmaGraph 53
5.7.1.3 DashboardInfoCard 54
5.7.1.4 NodeInfo, MultiNodeInfo, EdgeInfo, MultiEdgeInfo 54
5.7.1.5 FilterCriteria 54
5.7.1.6 Home page 55
5.7.1.7 Landing page 55
5.7.2 Point clustering 56
5.7.2.1 API client 56
5.7.2.2 Custom Node Type 56
5.7.2.3 Graphology camera ratio 56
5.7.2.4 Node Point Clustering 57
5.7.2.5 Edge Clustering 57
5.7.3 Filtering 58
5.7.3.1 View Filtering 58
5.7.3.2 Data Filtering 58
5.7.4 Leaflet / Sigma synchronisation 59
5.8 Sample SR App 60
5.8.1 Golang Sample SRApp 60
5.8.2 Micro Frontend Sample SRApp 60
5.9 Review 61
5.9.1 Central Frontend 61
5.9.1.1 Micro Frontends 61
5.9.1.2 Chakra UI 61
5.9.2 API 61
5.9.2.1 MongoDB Golang Driver 61
5.9.2.2 Images and GridFS 61
5.9.3 Micro Frontends 61
5.9.3.1 UI library 61
5.9.3.2 create-react-app template 62
5.9.3.3 CORS 62
5.9.3.4 CSS and config.js 62
5.9.4 Landing Page 62
5.9.4.1 Clustering 62
5.9.4.2 Filtering 62
5.9.4.3 Node/Edge Information 63

5.10 Improvements . . . 64
5.10.1 Central Frontend . . . 64
5.10.1.1 CSS . . . 64
5.10.2 Micro Frontends . . . 64
5.10.2.1 CORS . . . 64
5.10.3 Landing Page . . . 64
5.10.3.1 Clustered Edges onClick event . . . 64
5.10.3.2 Maximum/Minimum Zoom . . . 64
5.10.3.3 Empty Address Model Values . . . 64
5.10.3.4 Zoom Level Rebound . . . 64
5.10.3.5 Centroid Placement . . . 65
5.10.4 Deployment . . . 65
5.10.4.1 Central Frontend . . . 65
5.10.4.2 MongoDB . . . 65
5.10.4.3 Bypassing issues . . . 65

6 Conclusions 66
6.1 Results . . . 66
6.1.1 Remark regarding Deployment . . . 66
6.1.1.1 MongoDB . . . 66
6.1.1.2 Nginx . . . 66
6.1.2 Use Cases . . . 67
6.1.3 NFRs (Non functional requirements) . . . 67
6.1.4 Speedtest . . . 68
6.1.5 System Test . . . 69
6.1.5.1 Open Central Frontend . . . 69
6.1.6 Click Central Frontend logo . . . 69
6.1.6.1 Click Home in Navbar . . . 69
6.1.6.2 Click Apps in Navbar . . . 69
6.1.6.3 Click Landing Page in Navbar . . . 70
6.1.6.4 Click Configure on not registered Landing Page . . . 70
6.1.6.5 Register Landing Page . . . 70
6.1.6.6 Landing Page zoom out . . . 70
6.1.6.7 Landing Page zoom in . . . 71
6.1.6.8 Landing Page click node . . . 71
6.1.6.9 Landing Page click edge . . . 71
6.1.6.10 Landing Page click multi edge . . . 71
6.1.6.11 Landing Page disable clustering . . . 72
6.1.6.12 Landing Page re enable clustering . . . 72
6.1.6.13 Landing Page disable edges . . . 72
6.1.6.14 Landing Page re enable edges . . . 72
6.1.6.15 Landing Page filter node . . . 73
6.1.6.16 Landing Page filter area . . . 73
6.1.6.17 Landing Page filter asn . . . 73
6.1.6.18 Update Landing Page . . . 73
6.1.6.19 Delete Landing Page . . . 74
6.1.6.20 Register SR App . . . 74
6.1.6.21 Launch SR App . . . 74
6.1.6.22 Update SR App . . . 74

6.1.6.23 Delete SR App 75
6.1.7 Screenshots 76
6.1.8 Outlook 82
6.1.8.1 Central Frontend 82
6.1.8.2 Landingpage 82

7 List of Figures 83

8 List of Tables 83

Glossary 84

1 Technical Documentation

1.1 Introduction and Overview

1.1.1 Initial situation

As part of the term thesis, the base for the Central Frontend application was implemented which includes functionality for rendering a Segment Routing network, including LsNodes and LsEdges, onto a Leaflet map and cluster them together, resulting in greater overall visibility.

Additionally, the functionality of launching, registering and searching different Segment Routing Application frontends has been mocked.

The goal of this bachelor thesis is to expand these functionalities.

Also, there were a few use cases from our term thesis which were optional and therefore left out due to lack of time and could now be implemented in the course of this bachelor thesis. Those were updated and added to the current definition of use cases in a later chapter. Moreover, an overhaul of the clustering algorithm, user interface design and app management are needed.

1.1.2 Problem description

Each Segment Routing Application developed at INS needs to create and maintain its own user interface. This leads to a number of problems:

- The user interface of each Segment Routing Application has never been the focus, resulting in a suboptimal user interface and user experience.
- Each Segment Routing Application currently builds the user interface from scratch, adding overhead to each project.
- Frontends of Segment Routing Applications do not have a consistent look and feel.
- Each frontend needs roughly the same UI components (e.g. drawing a network).
- Frontend deployments are scattered around with different URLs. It is hard to keep an overview.

1.1.3 Goal

The thesis results will enable future Segment Routing Application developers to easily use a centralised and modular UI platform, so they can easily develop their frontends using predefined elements, which in turn can be launched inside the Central Frontend. Therefore, the conceptual steps need to be well-thought-out. The goals are divided into five main parts:

- Optimise the clustering algorithm, so it scales better and contributes to a more seamless experience. This may also be done by using static city lists as cluster centroids for easier handling.
- Add functionality in form of a filtering mechanism to only display a subset of nodes grouped by various attributes. Not only node attributes can be filter criteria, but also link usage/health may be used.

- Implement the Central Frontend in such a way, that future Segment Routing Application frontends can reuse the components and easily be added and removed from the Central Frontend.
- Give the actors an overall appealing experience by documenting the Central Frontend and its components well and designing the UI in a logical and visually pleasing manner.

The UI should be as modular as possible, easily extensible and provide high abstraction and reusability. For example, the Central Frontend should provide the functionality to easily draw a network so that future Segment Routing frontends can use that feature.

When you click on a link or node, you should get information about the object itself. This is already implemented but should be visualised differently.

The network map should ideally have color differentiated visual elements with icons to be more self-explanatory and more clearly arranged.

Optional

Identity and Access Management Application administrators can create/update/delete users and give them permissions to different functionality and Segment Routing Applications. Additionally, the application can be connected to different identity providers through OIDC.

Responsive CSS

The application can be used on mobile phones and tablets.

1.1.4 Scope and limitations

Currently, there is no way to get the coordinates of the Segment Routing (SR) nodes in Jalapeno. The API needs to be extended for this. However, this is not in the scope of this project and mocked coordinates will be used and they work as if the functionality is present in the API.

Multiple edges between two nodes and curved edges are not yet possible in SigmaJS. A visual workaround will be implemented which indicates that multiple links are available.

1.2 Requirements

1.2.1 Actors

The Central Frontend has two main actors.

1.2.1.1 Software Engineer

The actor should be able to get information on how to develop, deploy and connect his SR app frontend and launch it from the Central Frontend

1.2.1.2 Network Operator

The Central Frontend should be able to show a landing page to this actor and give the possibility to switch to different SR apps. The actor should be able to view the status of the network and filter for certain entities. They should also be able to show single node/link information on demand.

1.2.2 Use Case Diagram

The use case diagram visualises all the use cases and how they relate to different usecases and actors.

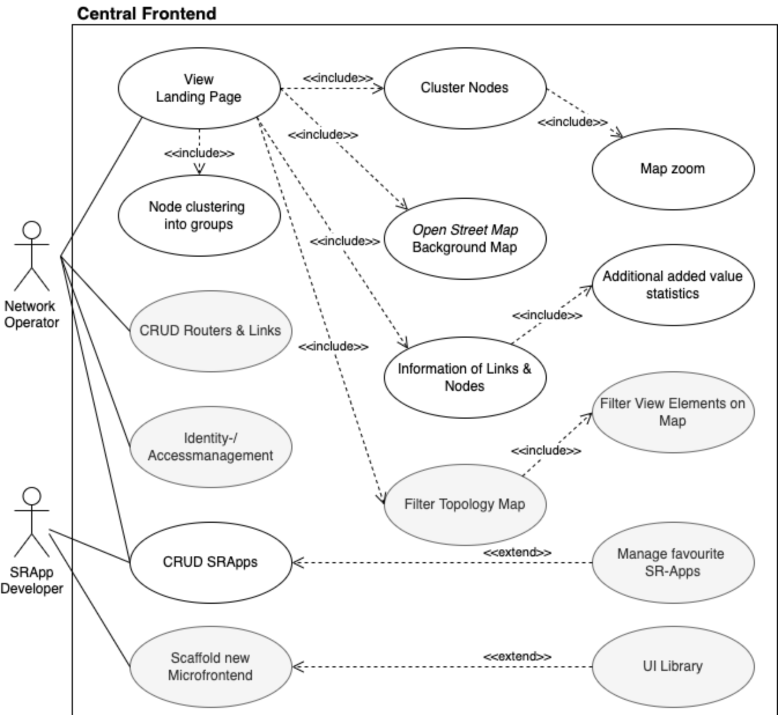


Figure 1: Use Case Diagram

1.2.3 Use Cases Brief

The use cases were formed in a brief format so that a compact overview over all the functionality is available. They are summarised into main and sub use cases by functional groups.

#	Use Case	Brief	optional /mandatory
UC1	View Landing Page	User visits Central Frontend and is greeted by a landing page wich shows the current status of nodes and links in the global network topology on a map.	mandatory
UC1-1	Landing Page - Map	The background of the landing page is a map. The map is based on Open Street Map.	mandatory
UC1-2	Landing Page - Information of Links and Nodes	The landing page shows additional information of the network. The links and nodes are placed onto their respective coordinates. The network is correctly drawn and can have multiple links between two nodes. The node symbol is based on the node type (e.g. router is drawn as a router, etc.).	mandatory
UC1-2-1	Additional added value statistics	Node information shows calculated values (e.g. degree of nodes) which add to the functionality of the landing page.	optional
UC1-3	Landing Page - Clustering	The nodes which are roughly at the same location should be hierarchically clustered into groups according to the zoom level. This is implemented with Nominatim OSM library which provides synchronous levels of abstraction to leaflet. The clustering can be seamlessly triggered by zooming in and out.	mandatory
UC1-3-1	Clustering - Map Zoom	The zoom levels of the map should be adapted to the number of subnodes that need to be shown on the screen. They need to be dynamic in a way that at least 5 zoom levels can be shown.	mandatory
UC1-4	Landing Page - Filtering	The landing page allows the user to filter for certain properties (e.g. ASN). The desired selection is either highlighted or the leftover are hidden.	mandatory
UC1-4-1	Filtering - View Filtering	Not only should the data be filtered by attributes but also the view can be filtered by view elements e.g. disable all edges for easier viewing experience.	mandatory
UC5	Register and show/link other SR apps	There are many different SR apps that have or will need an UI. The Central Frontend should be able to either embed those UIs or link to them. The SR apps should be searchable in the UI and the user should be able to "launch" them.	mandatory

UC5-1	CRUD SR apps dynamically	New SR-apps can be CRUD dynamically without touching the code of Central Frontend making it modular. In some cases the user only wants access to certain SR apps and not all of them. Making a ”marketplace” for adding/removing them is implemented.	mandatory
UC6	Port over the Service Programming application	The current UI of the Service Programming SR-App should be ported over to use the newly developed Central Frontend	optional
UC7	Manage favourite SR apps	The user may need the same SR apps often. So a favourite bar or category in the menu could be provided.	optional
UC8	Identity and Access Management	This functionality would allow the application to connect to an identity provider (e.g. Keycloak) and provide/allow different functionality depending on the users permissions.	optional
UC9	Scaffold new Microfrontend	The SR App developers want to set up new microfrontend UIs easily. Create a template for Segment Routing Application developers.	optional
UC10	UI library	Create a UI libary for the Central Frontend and all other Segment Routing Applications.	optional

Table 1: Functional Use Cases

1.2.4 NFRs (Non functional requirements)

#	**Requirement**	**optional /mandatory**
NFR1	Clustering algorithm scales with O(n * log n) or better.	mandatory
NFR2	Documentation and reusable building blocks for developing compatible SR apps exist.	mandatory
NFR3	The application needs to be developed cloud-natively. E.g. the application should be packaged into a Docker container.	mandatory
NFR4	The application needs to easily scale horizontally and be able to be deployed onto any Kubernetes cluster.	mandatory
NFR5	The renderer should display at least 1000 nodes and 10000 links. This can be achieved by using a webGL framework for client side graphics assisted calculation.	mandatory
NFR6	It needs no more than 3 clicks to get to the desired SR app.	mandatory

Table 2: Non Functional Requirements

2 Analysis

The purpose of the Analysis is to formulate the systems requirements. This is accomplished by establishing what the system has to do, according to the requirements.

2.1 Clustering

In the term thesis a first iteration of clustering has already been implemented which resulted in the following shortcomings:

- Cluster centroids overlap to different country or oceans
- Cluster have no indication of how many nodes are summarised
- There are no links between the clusters
- The algorithm is slow and produces jittering view elements when refreshed

2.1.1 DBSCAN

Density-based Spatial Clustering of Applications with Noise is a data clustering algorithm which takes a set of points in some space and groups points that are closely packed together. It marks outliers in low density regions and is very suitable for geolocation clustering.

Outliers get their own cluster centroid which prevents clusters in the ocean, clusters over country borders are still possible but less likely.

Nonetheless it still evaluated as a good alternative due to its ability to keep track of the connections and the general runtime efficiency of `O(log n)`.

2.1.2 K-means

K-means clustering is a method of vector quantisation. It aims to partition n nodes into k clusters and each node gets added to the cluster with the closest mean called centroids, serving as a prototype of a cluster. K-means clustering minimises within-cluster variances. It is relatively simple to implement and scales to large data sets. It is suitable for clusters of different shapes and sizes which applies to geological clusters.

In the term thesis a variant of this algorithm was already implemented and essentially following shortcomings of it were outweighing the efficiency benefits of this approach:

- unpredictable centroid placement
- manual cluster count definition

2.1.3 Nominatim

Nominatim[1] is an API to generate synthetic addresses from Open Street Map Points. It is used to reverse search coordinates and generate an address response from it. The returned address

[1]Nominatim Geocoding Software. 2022. URL: https://nominatim.org/

response contains multiple layers of abstraction that can then be used to form clusters. The following code snippet is the response of a reverse search:

Listing 1: Nominatim Response

```
1  GET https://nominatim.openstreetmap.org/reverse?
2  lat=47.356856769969525&lon=8.112738184067592&zoom=99&format=json
3
4  {
5     "place_id":67865315,
6     "licence":"Data OpenStreetMap contributors, ODbL 1.0.
          ↪ https://osm.org/copyright",
7     "osm_type":"node",
8     "osm_id":6214387944,
9     "lat":"47.3568323",
10    "lon":"8.1127676",
11    "display_name":
12    "9, Quellstrasse, Granichen, Bezirk Aarau, Aargau, 5722, Schweiz",
13    "address":
14    {
15       "house_number":"9",
16       "road":"Quellstrasse",
17       "village":"Granichen",
18       "county":"Bezirk Aarau",
19       "state":"Aargau",
20       "postcode":"5722",
21       "country":"Schweiz",
22       "country_code":"ch"
23    },
24    "boundingbox":["47.3567823","47.3568823","8.1127176","8.1128176"]
25 }
```

As depicted in the listing above the address gets split up into an address type which, depending on the used zoom parameter, shows the lowest abstraction of the address first.

2.1.3.1 Zoom parameter

The Nominatim zoom level correlates to the open street map ratio. That provides a seamless way of clustering nodes with the lowest available abstraction at a given zoom level. The nodes with the same value on top are clustered together.

2.1.3.2 Hosting Nominatim

The already available public API has a limit of 1 req/sec which is not sufficient to initialise all of the nodes in a network in a timely manner. That is why a dedicated server to host the API needs to be set up. The API therefore could be deployed to the `sr-000169` server and all the OSM data for the whole planet could be downloaded with the reduced dataset for reverse search and administrative borders only. This should end up in a 20GB dataset which allows for all reverse searches.

2.1.3.3 Decision

As the intent is to cluster by geographical data a clustering method that supports the generation of synthetic addresses by supplying coordinates with a latitude and longitude parameter is needed. In this case Nominatim provides a perfect base for our clustering algorithm. This will be discussed in detail in the design decisions.

2.1.3.4 Problems

Memory

To be able to load all the nodes, ways and relations a high amount of memory is needed. The buffer fills with data as the Postgres database writes the data too slow. When it reaches maximum capacity, it just crashes. To have a successfully running task it is essential to provide enough memory. The initial memory was set to 8GB. On further tests INS provided 16GB.

Disk Space

The .osm.pbf files are a compressed data format which needs to be unpacked first. The relatively small file sizes leads to some problems in the beginning as they turned out to be way larger. The planet.osm.pbf file only is 62.8GB at the time, but unpacked it can take up to 950GB of space.

Time

It takes a lot of time to unpack, write the data and index everything. For the entire planet dataset it takes about 6 days with the provided hardware. Due to the limited timeframe of this thesis a reduced input was used, so the implementation phase can be started. At the moment only the Europe dataset is deployed and it took 3 days for it to finish. This might change in the course of the thesis if a solution is found.

Cache size

After a few attempts to find a suitable cache size, it was set to about 5 GB, which resulted in a successful import run of the data.

Level of detail

With a parameter called `NOMINATIM_IMPORT_STYLE` the level of detail that the reverse search will provide could be controlled. Following values are allowed:

style	import time	DB size	drop
admin	4h	215 GB	20 GB
street	22h	440 GB	185 GB
address	36h	545 GB	260 GB
full	54h	640 GB	330 GB

Table 3: Nominatim Import Styles

For using only reverse searches we can drop a lot of information and end up with the used disk space like stated in the last column. We decided for admin as it shows administrative boundaries down to the city.

Final import

The used import statement:

```
nominatim import -reverse-only -no-updates -threads 4 -osm2pgsql-cache 5000 -osm-file
/var/dl/planet-latest.osm.pbf 2>&1
```

2.2 Network graph visualisation framework

To be able to display a network graph a JavaScript visualisation framework is needed. It was already decided to use the open source framework sigmaJS v2 in the term thesis, as it provides all the necessary tooling to visualise nodes and edges in a variety of ways.

The reason against sigmaJS v1 is that the plugins and documentation are outdated and no longer officially supported. Additionally, the INS will not have to migrate to version 2 in the near future. This comes with a few caveats which will be described in this chapter.

The reason for the decision against other libraries like d3.js or vis.js is that sigmaJS integrates webGL natively. This means it is performing better than other SVG or canvas based rendering libraries out of the box. This is essential to the purpose of the Central Frontend application because the topologies which should be displayed can contain thousands of nodes and links.

There also were different frameworks which had webGL support, but they are not free or open source which eliminated them from the evaluation process altogether.

2.2.1 Scalability Tests

Network topologies can have a large number of nodes and edges that need to be displayed. This can have a big effect on performance of our product. To verify the capabilities of sigmaJS we did performance tests with varying numbers of nodes and edges both with webGL disabled and enabled.

#	No. Nodes/Edges	Time with webGL	Time without webGL
1	1000 Nodes / 0 Edges	519ms	1.91s
2	1000 Nodes / 1000 Edges	866ms	3.23s
3	10000 Nodes / 0 Edges	1.23s	5.52s
4	10000 Nodes / 10000 Edges	1.85s	10.1s

Table 4: Scalability Test

Inferring from the above tests there will be no problem displaying thousands of nodes without waiting longer than 1-2 seconds. With webGL enabled the requirements of this project can be satisfied, and this framework could therefore be used performance-wise.

2.2.2 Advantages

SigmaJS v2 has a big community and a high activity in its open source repository. Questions regarding feature development of the newly released version of Sigma could be established by contacting one of the maintainers Alexis Jacomy.

2.2.3 Downsides

As this library was released during the last term it is quite new and has little plugins written for it. Luckily it is still possible to write the so-called programs for Sigma which can add certain

functionalities to the rendering process. This gives a lot of room to customise the aspects of the graph which need to be met.

There is no support for integrating OpenStreetMap or Google Maps as a synchronised background layer. This has to be implemented by us with no integrated support of the Sigma framework. This will be described in the implementation process.

Multiple edges between nodes are possible, but they overlap. This feature has not yet been integrated yet.

2.3 Other SR Apps and Micro Frontends

One of the main requirements for the Central Frontend is to allow other Segment Routing Applications frontends to be developed and included in the Central Frontend independently.

To solve this problem, it was necessary to conduct extensive research into possible solutions.

2.3.1 Mono Repo Approach

The simplest approach would be to have the Central Frontend as a standalone monolith. All the code would be hosted in the Central Frontend's repository and if a new UI for a new SR app is to be developed, the developers would need to create a new merge request and merge to the master branch in the Central Frontend's repository.

The components and the specific frontends of all the different Segment Routing Applications would be saved in the file structure in the repository.

While being really simple to develop, this approach has a number of issues:

- While adding new frontend the Central Frontend would need to be redeployed
- If there are two different teams working on separate frontends, merge conflicts may arise
- Organizations that deploy the Central Frontend might not want all the frontends that exist, but only a subset of them

This approach was set as a backup solution, if there were no other working approaches to be found. A proof of concept for this approach will not be pursued for this thesis.

2.3.2 API Approach

This approach would consist of writing an API where external frontends can register themselves to the Central Frontend. The Central Frontend can then display links to those. The other frontends are complete, standalone frontends.

The API approach is generally a good solution, because it allows for a centralised point of contact to manage frontends in a database and we will expand on it with the Micro Frontend approach.

2.3.3 Micro Frontend Approach

During the term thesis a blog post was discovered that introduced us to the term "Micro Frontend".

Inspired by microservices, a Micro Frontend is a piece of the frontend which is developed, tested and deployed independently as a unit. The Micro Frontends are then "glued" together in a central Container web app. The process of glueing is later explained in the implementation. The end user only sees a website, even though the single pieces of the frontend are self-contained. This is a cutting edge approach of developing frontends in large companies with a lot of teams. It caught our interest immediately due to seeming like an appropriate solution to the problem, and it was decided to conduct further research into this approach.

2.3.3.1 Initial Research

The first step was to get familiar with the aforementioned blog on how to include a Micro Frontend into a container app.

The author shared how to share UI components between specific Micro Frontendsby using a service called Bit. An evaluation of different tools resulted in Bit being a fit tool for the case, but sadly it did have costs attached to it. Luckily, there is a proper free and open-source solution in Storybook. It was decided to conduct a proof of concept for Storybook. The result can be found further down.

The author then goes to explain how he includes a React Micro Frontend into a React Container application.

As a start, all the teams developing on this frontend need to agree on the entry point of each Micro Frontend. In the example, the entry point is defined in the asset-manifest.json and called **main.js**.

The author then implemented a React component called "Microfrontend" that represents a Micro Frontend. The component fetches the **main.js** file from a certain host (e.g. http//localhost:8081/asset-manifest.json) and mounts the JavaScript files into an own script tag and renders the Micro Frontendin a specific div.

It was decided to go ahead and conduct a proof of concept with this approach with an API to register the Segment Routing Applications dynamically.

2.3.3.2 Micro Frontends Proof of concept

The goal of this proof of concept was to see if it is possible to dynamically add Micro Frontends to the Central Frontend. For this an API is needed, where a single Micro Frontend can be registered and list of all Micro Frontends can be requested with all the necessary information (host, name, etc.).

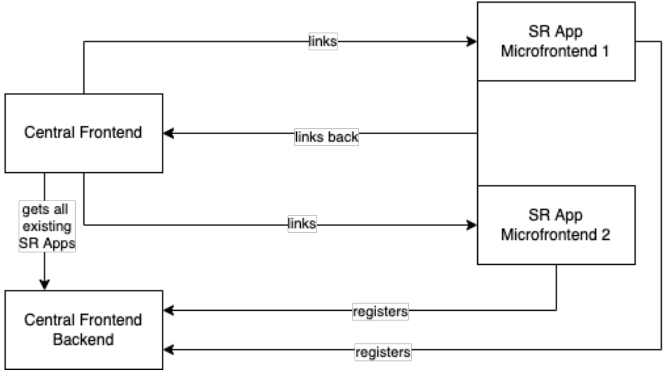


Figure 2: Micro Frontend

A Go API with the highly performant framework Gin was developed. For the persistence, a MongoDB database and the official Go MongoDB Driver was used.

A simple Segment Routing Application model was developed:

```go
package models

import "go.mongodb.org/mongo-driver/bson/primitive"

type SRApp struct {
   Id  primitive.ObjectID ‘json:"id,omitempty"‘
   Name string       ‘json:"name,omitempty" validate:"required"‘
   URL string        ‘json:"url,omitempty" validate:"required"‘
}
```

And simple routes were created:

```go
package routes

import (
   "api/controllers"

   "github.com/gin-gonic/gin"
)

func UserRoute(router *gin.Engine) {
   router.POST("/srapp", controllers.CreateSRApp())
   router.GET("/srapp/:id", controllers.GetSRApp())
   router.PUT("/srapp/:id", controllers.EditSRApp())
   router.DELETE("/srapp/:id", controllers.DeleteSRApp())
   router.GET("/srapp", controllers.GetAllSRApps())
}
```

The API was deployed to one of the developer machines and the MongoDB in a docker container. The API itself was highly responsive and performant, and there were no issues.

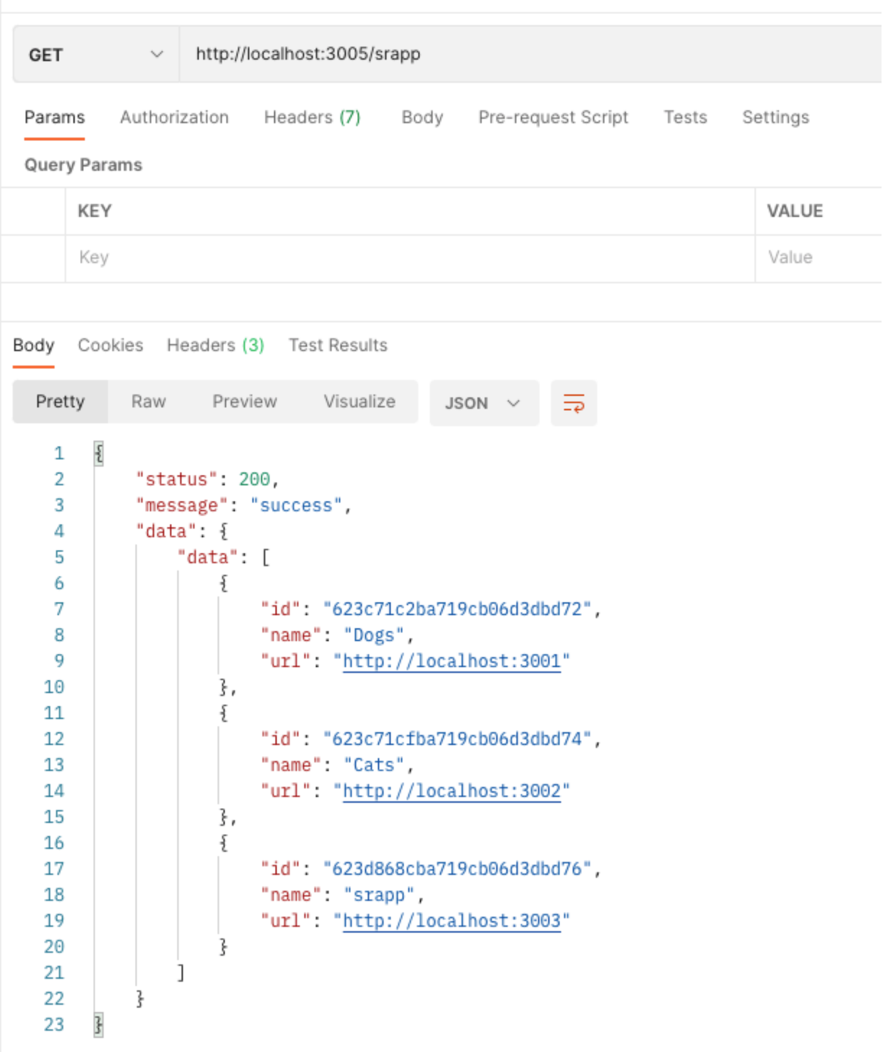


Figure 3: GET Request on the API

After, the examples from the aforementioned blog post were used and a modified version of the Container app was loaded into the Micro Frontend dynamically. Additionally, the UI was changed to look similar to the Central Frontend from the term thesis. Finally, the Container app should keep some of its own components and render the Micro Frontend in an underlying div. The repo can be found here.

Example of the API interaction:

```
const http = axios.create({
  baseURL: "http://localhost:3005/",
  headers: {
    "Content-type": "application/json"
  }
```

```
6 });
7
8 http.get<Array<SrApp>>("/srapp").then((response: any) => {
9    for (const srapp of response.data.data.data) {
10     if (srApps.findIndex(x => x.id === srapp.id) == -1){
11       srApps.push(new SrApp(srapp.id, srapp.name, srapp.url,
             "Calculator_Outline.png", null))
12     }
13     ReactDOM.render(<SrAppList edges={srApps}/>,
           document.getElementById("content"))
14   }
15  }).catch((e: Error) => {
16    console.log(e);
17  });
```

The proof of concept worked quite well. One problem was that it did not work with a TypeScript React app. Because of this, further research on how to solve that issue was conducted, as it was only working with JavaScript React apps.

2.3.3.3 Further research and Micro Frontends with TypeScript React apps

Research was conducted for the topic of Micro Frontend and Micro Frontends with TypeScript React by reading numerous blog posts and watching videos. The research resulted in finding the following npm package[2]. The package allows to bootstrap a TypeScript React Micro Frontend using the template and the following container app[3] which is also based on TypeScript React. The working example can be found here.

The npm package allowed to quickly create new Micro Frontend using the following command:

```
1 $ npx create-react-app my-app --template microfrontend-typescript
```

Going a step further in the bachelor thesis, a clone of the **create-react-app** template could be created to make the life of future Segment Routing Application developers easier when setting up a project.

The container app serves as a good starting point for the Central Frontend.

2.3.3.4 Conclusion of Micro Frontends

Since the proof of concept worked so well. Especially the latter found TypeScript React solution and the API approach. It was decided that Micro Frontends will be used to solve the problem of independent Segment Routing Application frontends.

The Central Frontend will be rewritten as a container web app and future Segment Routing Applications will be able to register themselves via the API.

In case there are unforeseen issues with the Micro Frontends approach that were not discovered during the proof of concept or analysis, a switch back to the API approach would be unproblematic.

[2]https://www.npmjs.com/package/cra-template-microfrontend-typescript
[3]https://github.com/gabrielcerutti/main-spa

2.3.3.5 Storybook proof of concept

As the blog mentioned already, it should be considered to create and deploy a custom UI library based on TypeScript React. For this, a proof of concept Storybook was conducted. To jump start this approach, this video guide[4] was used.

The UI library was created by using TSDX. With TSDX a scaffolding for the project can be created by using the **react-with-storybook** project.

Following Segment Routing Application button was added:

```
export interface ButtonProps extends HTMLAttributes<HTMLButtonElement>{
  /** Provide text */
  children: ReactNode;
  variant: "primary" | "secondary";
}

/** SR App Button */
export const Button = ({children, variant = "primary", ...props}:
    ButtonProps) => {
  return (
      <button
         {...props}
         style = {{
            backgroundColor: variant === "primary" ? "blue" : "gray",
            color: "white",
            border: "none",
            borderRadius: 100,
            padding: 10,
            cursor: "pointer",
         }}
      >
         {children}
      </button>
  )
}
```

With the button created, a **.stories.tsx** file can be created, which in turn will be a page on the Storybook UI:

```
import React from "react";
import {Meta, Story} from "@storybook/react"
import {Button, ButtonProps, Props} from "../src/index"
import {action} from "@storybook/addon-actions"

const meta: Meta = {
   title: "Button",
   component: Button,
   argTypes: {
      onClick: { action: "clicked"},
      children: {
         defaultValue: "Default"
      }
```

[4]https://www.youtube.com/watch?v=qSkHRVLcj6U

```
    }
}

export default meta;

const Template: Story<ButtonProps> = (args) => <Button {...args} />;

export const Default = Template.bind({});
export const Secondary = Template.bind({});

Secondary.args = {
    variant: "secondary",
    children: "Secondary",
    onClick: action("Secondary Click"),
}
```

Then a local instance of Storybook can be started:

```
$ npm start
$ npm storybook
```

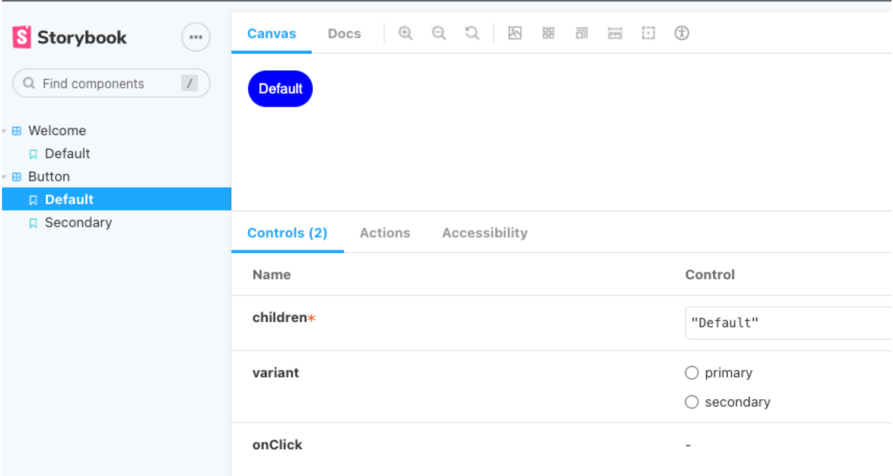


Figure 4: Storybook Overview 1

As the above tests show, all UI components can be developed and tested while also providing great support features like the accessibility checks. The docs page is also great for future Segment Routing Application developers which would like to create their own Micro Frontend.

Finally, in the proof of concept the whole library was packaged and deployed to npm and was used in microfrontend:

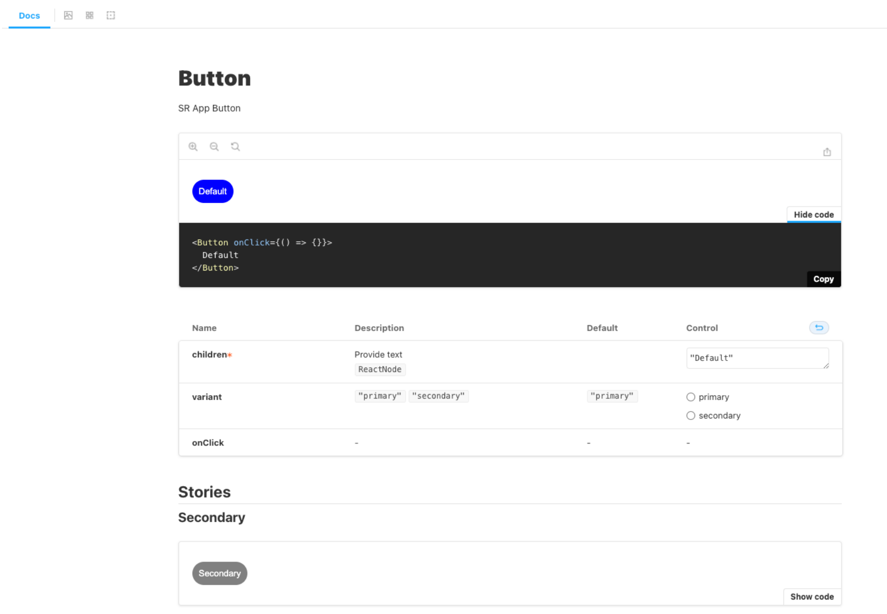


Figure 5: Storybook Overview 2

```
...
import { Button } from '@srapppoc/lib';
...
    <div className="App">
      <Button variant="primary">MY SRAPP BUTTON</Button>
      <Button variant="secondary">OTHER</Button>
    </div>
...
```

2.3.3.6 Conclusion of the Storybook proof of concept

It was decided to use Storybook in conjunction with TSDX to develop and test the UI components for all Segment Routing Application Micro Frontend based on the proof of concept. It showed the great values of Storybook as described above. There will be a separate repository with all the UI components based on TSDX and Storybook and CI/CD pipelines built to allow for smooth development and operations.

2.4 React UI Frameworks

A nicely thought out frontend needs a lot of planning work and HTML or CSS knowledge. This can be mitigated by using prebuilt components or templates which are readily available on the market for free. In this chapter some of them will be evaluated.

2.4.1 Chakra UI

Chakra UI is a simple, modular and accessible component library that provides building blocks to build good looking, modern React applications. There are countless options on prebuilt Chakra components to choose from. Going by looks and the wide collection of components, one of them is really suitable for us.

Choc UI

It provides ready to use code snippets for given UI items like the navigation bar, side bar or card flex layouts. They are highly customisable and provide all the features needed.

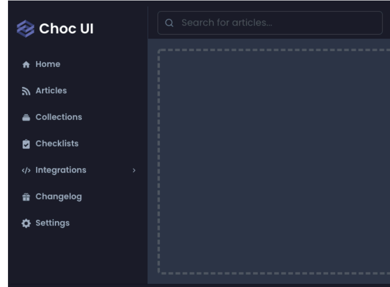


Figure 6: Choc UI - Sidebar
Source: https://choc-ui.com/docs/application-shells/sidebar-layouts

2.4.2 Material UI (mui)

Material UI is also a framework which provides predefined UI elements. It not only does that but also has templates which are ready to be downloaded. This boils just down to preference of the predefined looks compared to Choc UI.

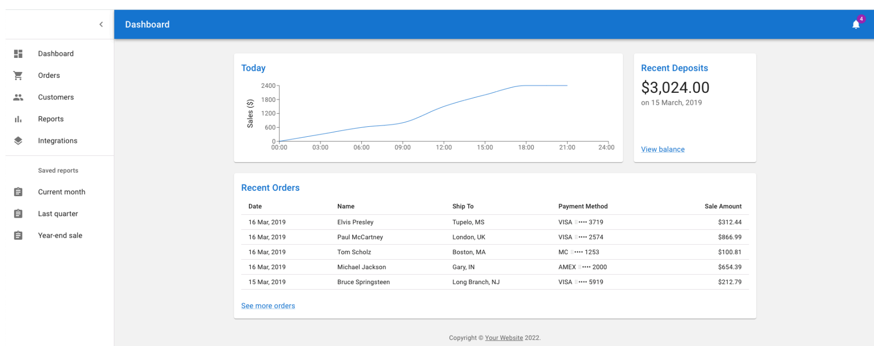


Figure 7: Material UI - Dashboard template
Source: https://mui.com/getting-started/templates/dashboard/

2.4.3 Decision

Going with the Choc UI approach seems easy enough, as customising it for the desired purpose is very user friendly and it also is aesthetically pleasing. We can still move to other designs on the fly as interchangeability is guaranteed for all Chakra UI based frameworks.

3 Architecture

This chapter explains the conceptualisation of the architecture. It explains the software quality characteristics along with the structure of the system. Its value lies in the idea on how to handle functional and non functional aspects which then get realised in a later project phase.

3.1 Overview

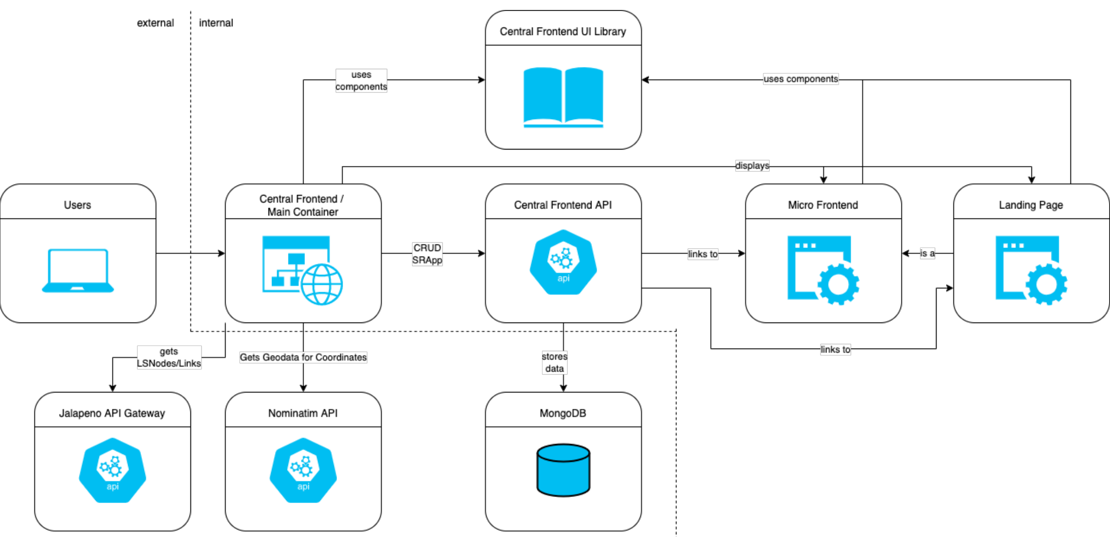


Figure 8: System Overview

3.2 Components

3.2.1 Central Frontend

The Central Frontend component is the main TypeScript React website that acts as the container for the landing page and all other Micro Frontends.

The Central Frontend allows users to register their Micro Frontends and a landing page in the UI and then the Central Frontend can launch and render the said Micro Frontend. The landing page will be rendered by default when the Central Frontend is launched.

3.2.2 Landing Page

As part of this thesis a landing page will be developed. The landing page will show the current network on a Leaflet map and the current status of the network.

The landing page is a Micro Frontend as well, just stored on a specific endpoint in the API. When the Central Frontend is launched, the registered landing page will be rendered. Any Micro Frontend can be registered as the landing page and the user can freely configure it to any other Micro Frontend.

There can only be a single landing page registered in the API, because this is the page that loads initially.

3.2.3 API

The Central Frontend API is a Go based REST API that allows CRUD operations for Segment Routing Application Micro Frontends entries and the landing page entry. It saves the data in a MongoDB database.

3.2.4 Central Frontend UI Library

The Central Frontend UI library is a TypeScript React UI library that is used to have unified-looking components across all Micro Frontends. The library will be scaffolded with TSDX with React + Storybook option and will hold all reusable UI components.

3.3 External components

3.3.1 Other SRApps/Microfrontends

The Micro Frontends are developed and deployed by other Segment Routing Application developers and being registered to the Central Frontend API and being displayed/launched by the Central Frontend.

The API will differentiate between Micro Frontends and Standalone Frontends.

A Micro Frontend will be rendered inside the Central Frontend, whereas a Standalone Frontend can be any website and will be opened in a new tab in the browser.

3.3.2 Nominatim

Nominatim will be installed on a dedicated server as there is a need to reverse search with the API to cluster our nodes on the landing page map.

3.3.3 MongoDB

The MongoDB will be deployed in the Kubernetes cluster and will be used to store the API data.

https://www.mongodb.com/docs/manual/core/gridfs will be used to store images, since the max document size is 16MB in mongoDB. GridFS bypasses that restriction.

3.3.4 Jalapeno API Gateway

The Central Frontend application will be using the Jalapeno API Gateway to interact with the Segment Routing network.

The API is based on gRPC and the protobuf definitions can be found here.

The API documentation can be found here.

The Jalapeno API Gateway was extended to use the envoy proxy, allowing the Central Frontend to use the gRPC-web implementation.

3.4 Technologies

3.4.1 Languages

Central Frontend and UI Library

- TypeScript
- HTML5
- CSS (SASS)
- Micro Frontend template from `https://github.com/gabrielcerutti/main-spa#getting-started-with-main-spa`

Central Frontend API

- Golang

3.4.2 Data Serialisation languages

- YAML
- Helm Templating

3.4.3 External Libraries

Central Frontend

- sigmaJS v2 - A JavaScript library dedicated to graph drawing
- React >= v17.0.0 - A JavaScript library for building user interfaces
- gRPC-web - A gRPC implementation for browsers

Central Frontend API

- gin
- mongo-db driver

Central Frontend UI Library

- React
- Storybook
- TSDX

3.4.4 Browser Support (tested)

- Chrome v96.0.4664.110
- Firefox v95.0.1

- Safari v15.0

3.5 Design

The Central Frontend will receive a new design based on choc-ui. Choc-ui is a framework that offers prebuilt Chakra-UI components. The predefined color palette will be used.

3.5.1 Wireframes

The wireframes have the purpose of creating a baseline for the UI design which will be followed in the implementation phase. The location the wireframes depict can be found in the figure description.

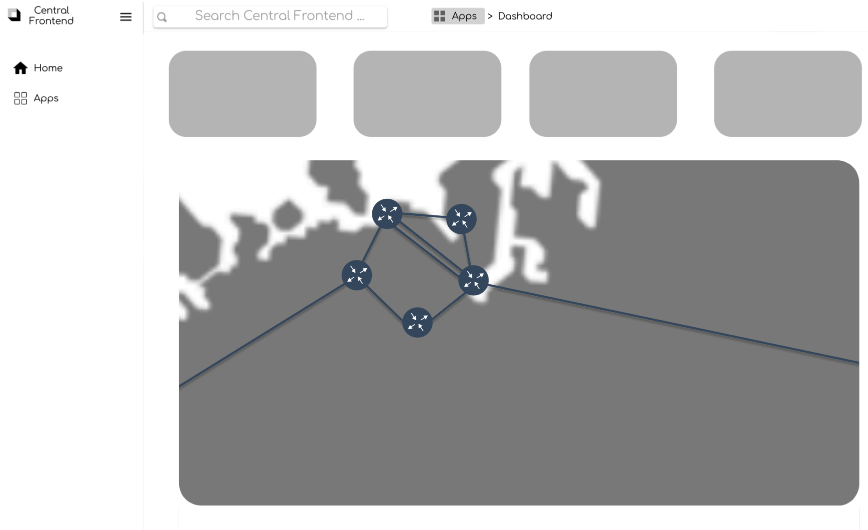


Figure 9: Dashboard
Source: own creation

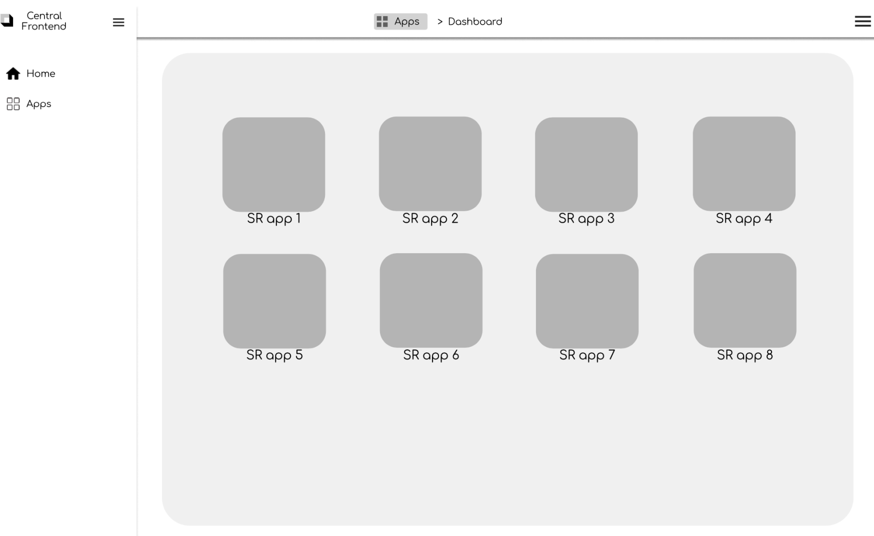


Figure 10: Appscreen
Source: own creation

Figure 11: Open App
Source: own creation

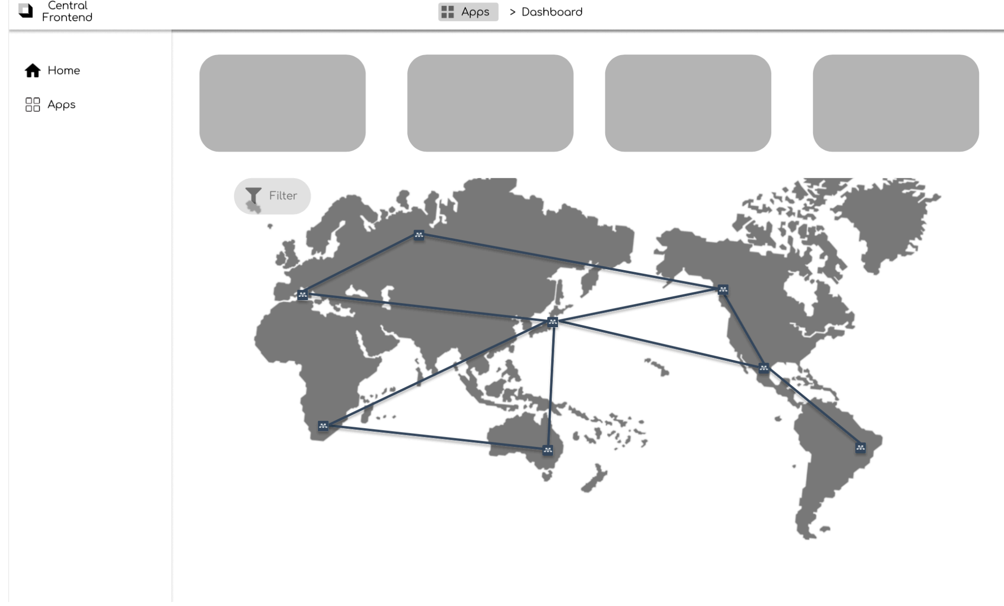


Figure 12: Clustering
Source: own creation

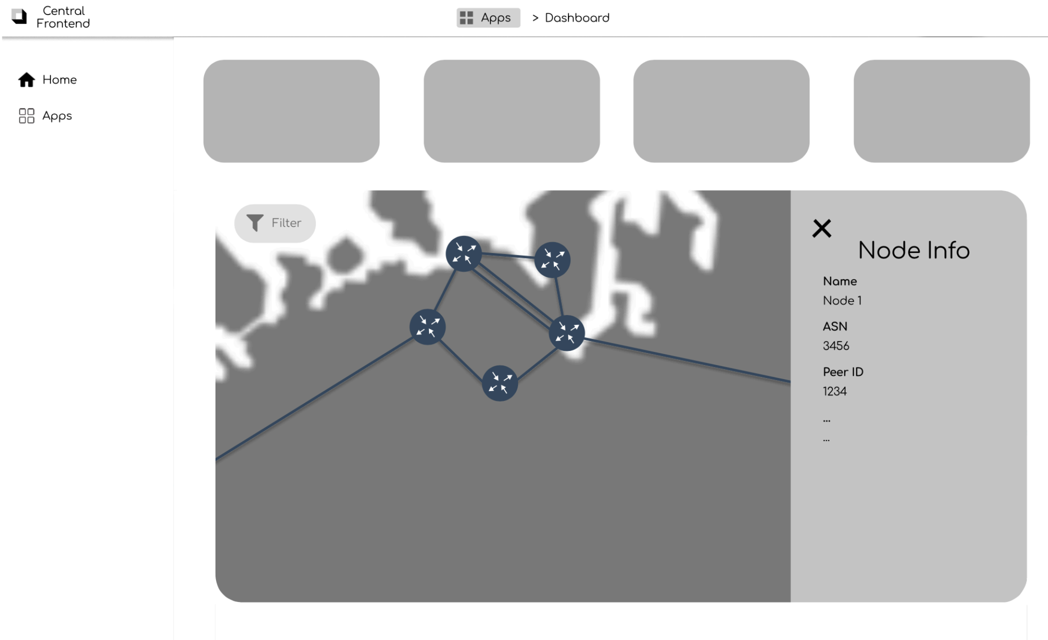


Figure 13: Node Information
Source: own creation

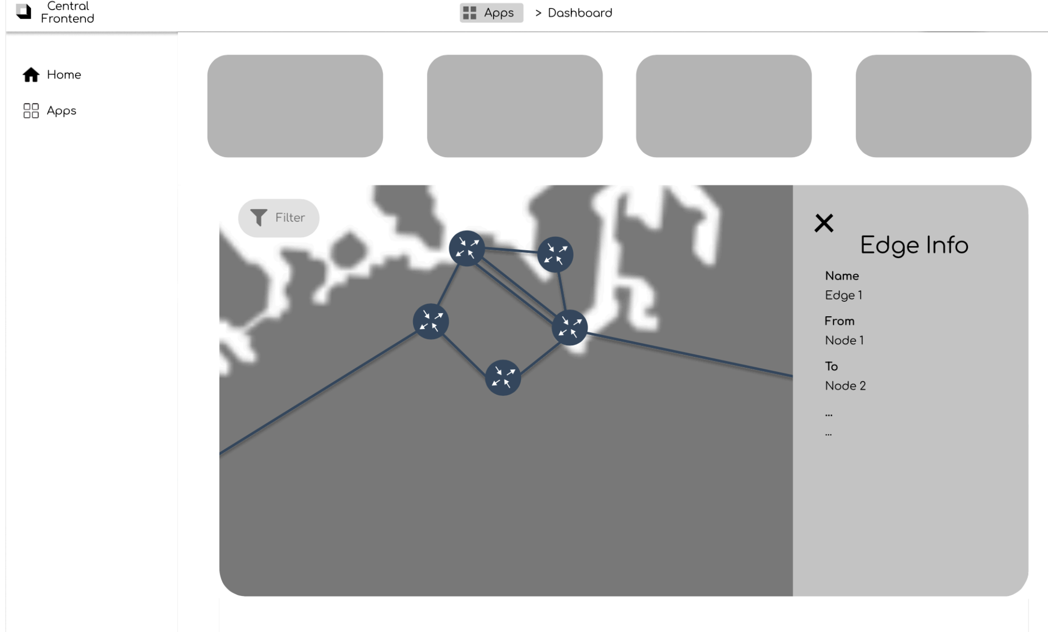


Figure 14: Edge Information
Source: own creation

3.6 Deployment

The Central Frontend, the API and all Micro Frontends will be containerized and deployed to a Kubernetes cluster as visualised in figure 15. To simplify the deployment, the Central Frontend, the API and a mongoDB will be packaged into a helm chart. Micro Frontends will have their own helm charts. All the Central Frontend components will be running in the namespace "central-frontend".

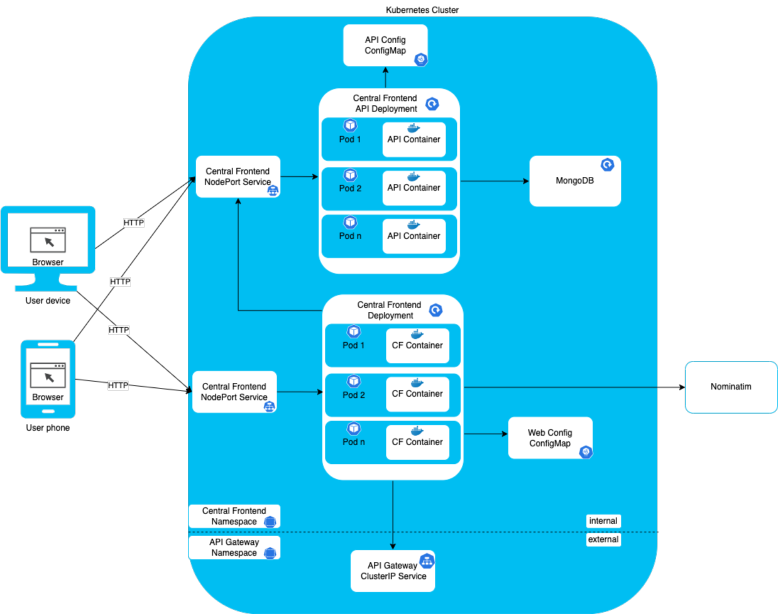


Figure 15: Deployment

3.7 Infrastructure

The Central Frontend has two different environments where it can run:

- Server: sr-000167.network.garden, this environment is a real segment routing network. It contains 10 LSNodes. The only part of it that is mocked, is the ArrangoDB coordinates collection and its values.
- Server: sr-000169.network.garden, this is a mocked environment, which we can use to mock nodes, links, edges and coordinates in high numbers. The Jalapeno API Gateway is deployed and can be used.

4 Design decisions

In this chapter we will elucidate the important design decisions we made a bit more in detail. The advantages and disadvantages of our options were weighed out to find the most suitable solution for this project.

4.1 Clustering

It was decided to go for an approach with a geolocation reverse search library that returns a structured address model as a response. The one in question is Nominatim API which is based on OpenStreetMaps data. The distinctive address parts conveniently correlate to the Leaflet zoom levels therefore they can be used interchangably with the Leaflet zoom function to display the desired granularity of nodes or node clusters.

4.1.1 Reasons

With all the geolocation services available the most important features were that it is free to use without a request limitation in place, returns a structured data response and is fast enough to make requests for 1000 nodes in less than 10 seconds. Nominatim is free to use for up to 1 request per second which would have violated two of our three requirements. Fortunately there is a way to bypass this limitation which is to host it privately. This was a valid option and with that all the requirements were met.

4.1.2 Proof of concept / Usability test

As a part of the evaluation for this approach a proof of concept for point clustering was created with the public Nominatim API. In it there is a graph drawn with SigmaJS which displays nodes with different fixed coordinates. For each node the according address is received by calling the Nominatim API `/reverse` endpoint and saving the address into a custom node type. With this information the graph is redrawn with the country part of the address as nodes and all the information is gathered for the implementation to start.

4.1.3 Data structure

The returned address data structure from Nominatim looks like the following listing:

Listing 2: Nominatim address model

```
"address":
{
...
   "village":"Granichen",
   "county":"Bezirk Aarau",
   "state":"Aargau",
   "postcode":"5722",
   "country":"Schweiz",
...
}
```

4.2 User Interface Framework

There are many user interface frameworks on the market and it was decided to use Chakra UI in combination with Choc UI for our purpose. Choc UI features predefined components like a navigation bar, an application drawer, forms and buttons. The details of those will be explained in a later chapter.

4.2.1 Reasons

Of all the user interface frameworks Chakra UI offered the most customisability in terms of already managed styling attributes and modularity at the same time. Choc UI in particular uses Chakra UI styling but takes it a step further and implements lists, pagination and forms which can be easily adapted to fit our user interface with size of elements and data passing between elements.

4.2.2 Proof of concept / Usability test

The evaluation for this framework mostly relied on modularity and a pretty design. With a small example of a landing page with a navigation bar, side bar and content placeholder we could verify that the final design would be possible according to our wireframes. This proof of concept later evolved and morphed into our main project and gave us a baseline for the design of the central frontend for segment routing applications.

4.3 Micro Frontends

It was decided, to implement the Micro Frontend approach for the Central Frontend and all other Segment Routing Application frontends. The current landing page will become a separate Micro Frontend. The landing page of the Central Frontend will be configurable and can be changed to any Micro Frontend. An API will be created which will store all the necessary information to render the landing page, any Micro Frontend and a list of all Segment Routing Application frontends. The Central Frontend will be able to differentiate between a Micro Frontend and a `Standalone Frontend`.

The Micro Frontend will be rendered inside a certain space in the Central Frontend and the `Standalone Frontend` will be opened in a new tab.

Also the API will be implemented in Go using the Gin framework and data will be stored in a MongoDB database.

4.3.1 Reasons

During the proof of concept, there were no problems with implementing the Micro Frontend approach backed with an API.

Micro Frontends seem to work especially well with big or distributed teams, where a lot of people want to make changes to one frontend.

At the INS, there is usually more than one team working on a Segment Routing Application, so a more classical approach would mean that each team either creates their own UI, which is now the

case, or all the teams try to deploy a page in a central UI repository. The second option would lead to very frustrated developers, as it would mean merge conflicts and maybe disagreements of certain parts of the UI.

A centralized and modular platform for all the different Segment Routing teams, seems to be the best.

5 Implementation

Now the architecture, technologies, logical approaches for Micro Frontends and the point clustering algorithm are decided on, the implementation phase can start. This chapter explains the implementation details of all the components.

5.1 Repositories

These are all the GitLab repositories that we created and used in this project. Contained are application repos, generators, docs and proof of concepts:

- root project: Contains all the underlying repos: https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba
- central-frontend: The main repository for the Central Frontend React app. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/central-frontend
- landingpage: Repository for the landingpage SR app. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/landingpage
- api: Repository containing the api of the Central Frontend. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/api
- docs: The repo containing the technical documentation. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/docs
- projectplan: Repository containing the projectplan. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/projectplan
- cra-template-srapp: Micro Frontend template for all SRApp frontends: https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/cra-template-srapp
- helm-chart: Helm Chart that deploys the Central Frontend, API and MongoDB: https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/helm-chart
- sample-srapp: A sample Micro Frontend for a sample SR application: https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/sample-srapp
- sample-srapp-golang: A sample SR application in Go: https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/sample-srapp-golang
- srapp-lib: Repository containing the storybook implementation and components. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/srapp-lib
- gen: The repo containing the small go program used to generate json file which were uploaded to the mocked environment. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/gen
- poc: Repository Group containing all the proof of concepts. https://gitlab.ost.ch/ins-stud/sr-app-central-frontend/ba/poc

5.2 Central Frontend

The Central Frontend web application is the main Container application that renders itself and the other Micro Frontends.

5.2.1 Description

The Central Frontend is a React web application that is the main Container that renders itself and all Micro Frontends.

The landing page of the Central Frontend can be configured to be any Micro Frontend.

Additionally, the Central Frontend allows `SRApps` to be registered and displays them in a list. From the list, a `SRApp` can be launched. If the specific `SRApp` is Micro Frontend, it will be rendered inside the Central Frontend. If it is a `Standalone Frontend`, the `SRApp` will be launched in a new tab.

To render a Micro Frontend, the Central Frontend interacts with the API and gets all the necessary data to render a Micro Frontend. Once it has the data, the Central Frontend gets the `asset-manifest.json` of the Micro Frontend and creates a new `script` tag with the compiled JavaScript code. Finally, the Central Frontend can call the `render` function and dispay the Micro Frontend.

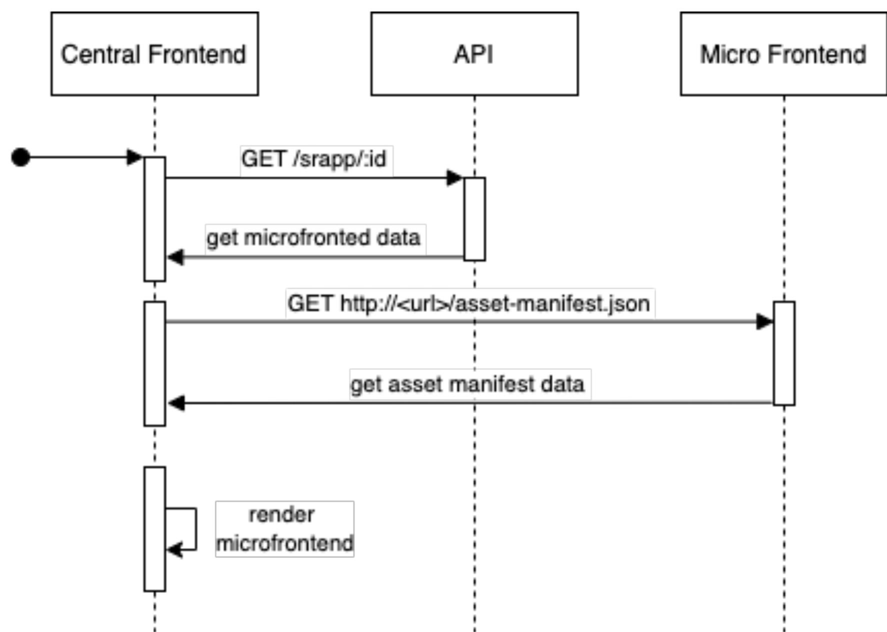


Figure 16: Micro Frontend Sequence Diagram
Source: own creation

5.2.2 Components

5.2.2.1 Chakra UI (Choc UI)

Chakra UI is a simple, modular and accessible component library that gives the building blocks for designing React applications.

Instead of building a standalone UI out of these building blocks, Choc UI is used.

Choc UI s a set of accessible, reusable and prebuilt Chakra UI components. The components have styling applied to them and at the same time it gives the developers a lot of customisation options to design the components to fit the look and feel and usability requirements.

5.2.2.2 Config

The config of the Central Frontend is controlled by the `config.js` located in the public folder of the application and contains environment variables. The config is loaded through the `Config` class.

For now, there are only two environment variables:

- API_URL: The URL of the Central Frontend API
- JAGW_URL: The URL of the Jalapeno API Gateway

5.2.2.3 API Client To interact with the Central Frontend API, an API client abstracts all the HTTP calls to functions.

The API allows for CRUD operations for SRApp, Landingpage and Images.

5.2.2.4 PageController

CentralFrontend

The CentralFrontend component renders the navigation sidebar and the header. Additionally, it renders the Main component.

The navigation sidebar contains the Central Frontend logo, the `Home` buttom, the `Apps` button and the `Landing Page` button.

When the logo is clicked, the Central Frontend will be rerouted to `/`. When `Home` is clicked, the Central Frontend will be rerouted to `/`. When `Apps` is clicked, the Central Frontend will be rerouted to `/apps`. When `Landing Page` is clicked, the Central Frontend will be rerouted to `/landingpage`.

Main

The Main component does the following things:

- Sets up the ChakraProvider
- Sets up the Micro Frontend workspace
- Sets up the Router
- Renders the landing page if configured. If not, it will render a call to action element, that will ask the user to configure a landing page

Route	Component to render	Description
/	RenderLandingPage()	Renders the landing page Micro Frontend or the call to action to configure a landing page
/apps	<Apps />	Renders the <Apps /> component
/app	<App />	Renders the <App />
/app/:id	<App />	Renders the <App /> with the URL parameter of id
/landingpage	<Landingpage />	Renders the <Landingpage /> component
/:id	<Micro />	Renders the <Micro /> component with the URL parameter of id

Table 5: Routes

ChakraProvider

The ChakraProvider sets up Chakra UI in the Central Frontend and its Micro Frontends.

Micro Frontend Workspace

The Workspace component sets up the HTML tag, where the Micro Frontends can be rendered in.

Router

The Router sets up all the routes of the Central Frontend:

RenderLandingPage()

The RenderLandingPage function renders the landing page by calling the `getLandingPage()` function of the API client and getting all the necessary information to render the landing page Micro Frontend.

If the `getLandingPage()` function of the API client does not return a configured landing page, it will render a call to action element, that will ask the user to configure a landing page. If the user clicks the `Configure` button, the app will reroute to the `/landingpage` route.

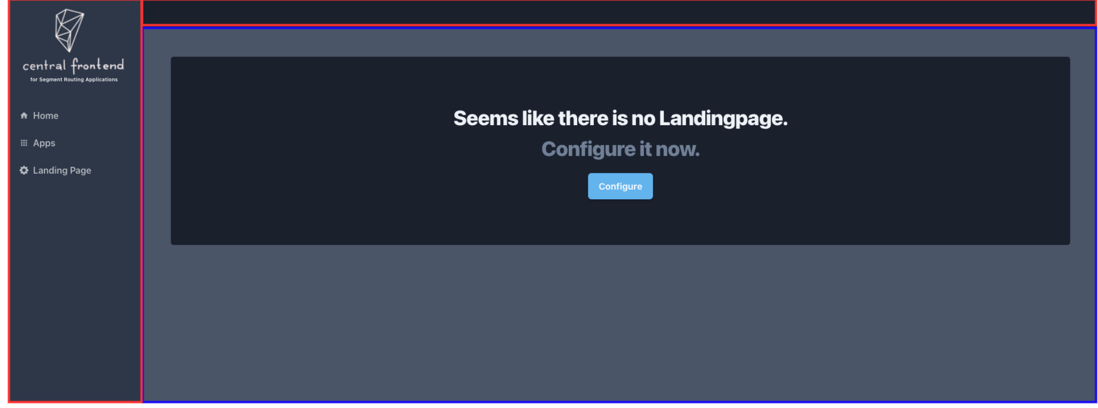


Figure 17: The CentralFrontend component in red and the Main component in blue
Source: own creation

5.2.2.5 MicroFrontend

The MicroFrontend receives all the parameters necessary to render a Micro Frontend. In a first step, the component will get the asset-manifest.json file from the specific Micro Frontend. It is always located at http://<url>/asset-manifest.json, where the URL is the address of the web server, where Micro Frontend is deployed at.

```
{
  "files": {
    "main.css": "/static/css/main.bfda4427.css",
    "main.js": "/static/js/main.5e014873.js",
    "index.html": "/index.html",
    "main.bfda4427.css.map": "/static/css/main.bfda4427.css.map",
    "main.5e014873.js.map": "/static/js/main.5e014873.js.map"
  },
  "entrypoints": [
    "static/css/main.bfda4427.css",
    "static/js/main.5e014873.js"
  ]
}
```

Figure 18: Example asset-manifest.json
Source: own creation

In the main.js file, we can find all the compiled JavaScript code that is required to run our Micro Frontend. The Micro Frontend has a function called render, which the Central Frontend calls. The function returns the Micro Frontend React component, which renders the Micro Frontend into the workspace defined in the Main component.

If the Micro Frontend is still loading, the component renders a circular progress bar. If the Micro Frontend could not be rendered due to an error, the component displays an error message.

The functionality from the file was inspired in this template.

5.2.2.6 Pages

SRApps

The `SRApps` component renders a search bar and a list of `AppCards` representing a `SRApp` in the API. The list of `AppCards` is filled by the API client function `getAllSRApps()`. However, the first element is always the `NewAppCard` component, which is always loaded in and routes the Central Frontend to `/app`.

The search bar will filter the list of `SRApps` and only show the ones that match the searched string either in the name or the description.

Micro

The `Micro` component receives a `id` URL parameter that matches a Micro Frontend entry in the database of the API. The component renders the matching Micro Frontend in the `Workspace` div of the Central Frontend.

AppCard

The `AppCard` component has the title, description and image of a `SRApp`. Additionally, the component renders a `Launch`, `Edit` and `Delete` button for each `SRApp`.

If the `SRApp` is a `Standalone Frontend`, the `Launch` button will open a new tab in the browser and redirect the user to the specific website. On the other hand, if the `SRApp` is a Micro Frontend, it will reroute the Central Frontend to the `/a:id` route, which in turn will render the Micro Frontend.

The `Delete` button makes a `HTTP Delete` call to the API and deletes the `SRApp` and refreshes the `SRApp list`.

The `Edit` button reroutes the Central Frontend to the route `/app/:id` and gives the ID of the `SRApp` as a URL parameter. In turn, the Central Frontend will render the `SRApp` component.

NewAppCard

The `NewAppCard` component is a static `AppCard` that is always rendered at the first position in the list. When clicked, it reroutes the Central Frontend to the `/app` path. In turn, the Central Frontend will render the `SRApp` component.

SRApp

The `SRApp` component allows to create a new `SRApp` or update a existing one.

The component is rendered either at the `/app` path, which is used for creating new `SRApps`. Or at the `/app/:id` path, which is used for updating an existing `SRApp`.

A `SRApp` is either a Micro Frontend or a `Standalone Frontend`.

If the `SRApp` is a `Standalone Frontend`, the following fields can be submitted:

Field	Component to render
Name	Name of the SRApp. Will be displayed in the list.
Host	URL of the SRApp. The http prefix can be ommitted.
Description	Description of the SRApp. Will be displayed in the list.
Picture	Picture of the SRApp. Will be displayed in the list.
Application Kind	Standalone Frontend or Micro Frontend

Table 6: Fields Standalone Frontend

And if the `SRApp` is a Micro Frontend, the following fields can be submitted:

Field	Component to render
Name	Name of the SRApp. Will be displayed in the list.
Host	URL of the SRApp. The http prefix can be ommitted.
Description	Description of the SRApp. Will be displayed in the list.
Picture	Picture of the SRApp. Will be displayed in the list.
Application Kind	Standalone Frontend or Micro Frontend
Micro ID	The ID of the Micro Frontend. Configured in the config-override.js of the Micro Frontend
Base Path	The basepath of the Micro Frontend. Configured in the App.tsx of the Micro Frontend

Table 7: Fields Micro Frontend

The `Picture` field allows the upload of an image for the specific `SRApp`. The user can either drag and drop an image or click on `Upload` which will open a file input.

Landingpage

The `Landingpage` component allows the user to configure a Micro Frontend which will be rendered when the Central Frontend is launched.

The component allows for CRUD operations for the landing page URL.

The API allows only for a single landing page to exist, and the landing page needs to be a Micro Frontend with the `microID` set to `landingpage`.

5.2.3 Docker

The Central Frontend is packaged and deployed into a Docker container that exposes port 3000.

The `Dockerfile` utilises a `multistaged build` which keeps the resulting `Docker Image` compact.

The base Docker image for the Central Frontend is a `nginx` image.

5.2.4 CI CD

The `.gitlab-ci.yml` file contains the CI CD config. On merge requests or on the master branch, the Docker image is being built.

On tagging a commit, the latest Docker image is being tagged with the git tag.

5.3 Central Frontend API

The Central Frontend API is a Go API implemented with the framework Gin.

It allows for CRUD operations for the following objects:

- SRApp
- SRApp pictures
- Landingpage

5.3.1 Persistence

The API stores all data in a MongoDB database. To interact with the database, the official Go Go MongoDB Driver is used.

5.3.2 Model

Both the `Landingpage` and `SRApps` are being represented by the same model:

```
1  type SRApp struct {
2    Id          primitive.ObjectID ‘json:"id,omitempty"‘
3    Name        string         ‘json:"name,omitempty" validate:"required"‘
4    Host        string         ‘json:"host,omitempty" validate:"required"‘
5    Description string         ‘json:"description,omitempty"
       validate:"required"‘
6    IsMicrofrontend bool       ‘json:"ismicrofrontend,omitempty"‘
7
8    // Only Microfrontend
9    MicroID string ‘json:"microid,omitempty"‘
10   BasePath string ‘json:"basepath,omitempty"‘
11 }
```

5.3.3 Routes

The following routes are defined for the API:

```
1   db := client.Database("api")
2    srappCollection := db.Collection("srapp")
3    landingpageCollection := db.Collection("landingpage")
4
5   router.GET("/srapp", controller.GetAllSRApps(srappCollection))
6    router.POST("/srapp", controller.CreateSRApp(srappCollection))
7    router.GET("/srapp/:id", controller.GetSRApp(srappCollection))
8    router.PUT("/srapp/:id", controller.EditSRApp(srappCollection))
9    router.DELETE("/srapp/:id", controller.DeleteSRApp(srappCollection))
10
11   router.GET("/srapp/:id/image", controller.GetImage(db))
12   router.POST("/srapp/:id/image", controller.CreateImage(db))
13   router.PUT("/srapp/:id/image", controller.UpdateImage(db))
14   router.DELETE("/srapp/:id/image", controller.DeleteImage(db))
```

```
15
16     router.GET("/landingpage",
           controller.GetLandingpage(landingpageCollection))
17     router.POST("/landingpage",
           controller.CreateLandingpage(landingpageCollection))
18     router.PUT("/landingpage",
           controller.EditLandingpage(landingpageCollection))
19     router.DELETE("/landingpage",
           controller.DeleteLandingpage(landingpageCollection))
```

5.3.4 SRApp controller

The controller for the `srapp` endpoint allows for CRUD operations for `SRApp` objects and additionally allows for an HTTP GET call to list all existing `SRApps`.

5.3.5 SRApp Image controller

The image controller stores the images of `SRApps` in `GridFS` buckets. `GridFS` allows MongoDB to store a file in chunks instead of one document.

```
1     image, _, err := c.Request.FormFile("image")
2        if err != nil {
3           c.JSON(http.StatusBadRequest, model.Response{
4              Status: http.StatusInternalServerError,
5              Message: fmt.Sprintf("error getting image: %s", err.Error()),
6           })
7           return
8        }
9
10       bucket, err := gridfs.NewBucket(db)
11       if err != nil {
12          c.JSON(http.StatusBadRequest, model.Response{
13             Status: http.StatusInternalServerError,
14             Message: fmt.Sprintf("error creating gridfs bucket: %s",
                   err.Error()),
15          })
16          return
17       }
18
19       if err := bucket.UploadFromStreamWithID(objId, id, image); err != nil {
20          c.JSON(http.StatusBadRequest, model.Response{
21             Status: http.StatusInternalServerError,
22             Message: fmt.Sprintf("error streaming image to bucket: %s",
                   err.Error()),
23          })
24          return
25       }
```

5.3.6 Landingpage controller

The controller for the `landingpage` endpoint allows for CRUD operations for a single `Landingpage` object. There can only be one `Landingpage` object.

5.3.7 Docker

The API is packaged and deployed into a Docker container that exposes port 8080.

The `Dockerfile` utilises a `multistaged build` which keeps the resulting `Docker Image` compact.

The base Docker image for the Central Frontend is a `distroless static` image.

5.3.8 CI CD

The `.gitlab-ci.yml` file contains the CI CD config. On merge requests or on the main branch, the Docker image is being built.

On tagging a commit, the latest Docker image is being tagged with the git tag.

5.4 Helm Chart

To simplify the deployment of the Central Frontend, API and the MongoDB, a Helm chart was created.

The following Central Frontend workloads are deployed:

- Deployment: central-frontend:1.1.0
- Service: Type NodePort
- ConfigMap: config.js with environment variables

The following API workloads are deployed:

- Deployment: api:1.0.0
- Service: Type NodePort
- ConfigMap: environment variables

The following MongoDB workloads are deployed:

- Deployment: mongoDB:4.4.6
- Service: Type ClusterIP
- PersistentVolume: Filesystem
- PersistentVolumeClaim

5.4.1 Limitations

In a production environment, the upstream helm chart for the MongoDB should be considered because there currently are some network timeouts with the MongoDB deployment. Additionally,

the version 5 of MongoDB should be considered for future compatibility. However, version 5 requires an up-to-date CPU microarchitecture.

5.5 srapp-lib

In this thesis centralising the UI components, e.g. Buttons, was started and they were added into dedicated UI library. The specific elements can be viewed in Storybook. The UI library is published here.

The general workflow for the UI library is as follows:

- Write a new UI component in the `src` directory and register it in the `src/index.tsx` file
- Write a story in the `stories` directory
- Merge and publish the library
- Use it in the Central Frontend or a Micro Frontend

For now, only a couple buttons exist in the library. However, the library can be easily extended with elements that are reusable.

5.5.1 Story

A story captures the rendered state of a UI component. Developers car write multiple stories per component that describe all the "interesting" states a component can support.

5.6 Create React App Template

To improve the developer experience of future Segment Routing Application engineers, a create-react-app template was created.

The template sets up all the NPM packages and comes with a pre-written Dockerfile and gilab-ci.yml. Additionally, all the necessary configuration changes are marked with a TODO comment.

With this template, Segment Routing Application developers can create their own Micro Frontend, fix the TODOs and start developing their UI easily and efficiently.

The resulting Micro Frontend repository allows for either standalone development or by inclusion into a Central Frontend instance.

The following paragraph shows the benefits of the template.

5.6.1 config-overrides.js

The Micro Frontend uses react-app-rewired that allows tweaking of the webpack config, without forking the default react scripts.

Webpack is used to bundle the React code into JavaScript code. With this config override, it is possible to control how the Micro Frontend is launched.

The `config.output.library` field should be set to the `micro ID` of the Micro Frontend.

5.6.2 Chart

The Micro Frontend comes with a pre-written Helm chart, which can be used to deploy the Micro Frontend to a Kubernetes cluster.

5.6.3 microfrontend.tsx

In this File, we initialise our Micro Frontend and the register the `render` function, which will be called by the Central Frontend.

The functionality from the file was inspired in this template.

5.6.3.1 API

The template comes with a pre-compiled gRPC web TypeScript client that can talk to any Jalapeno API Gateway.

The following command was used to compile the proto files:

```
1 protoc -I=. **/*.proto \
2 --js_out=import_style=commonjs,binary:./jagw-ts \
3 --grpc-web_out=import_style=typescript,mode=grpcweb:./jagw-ts
```

More information can be found here.

5.6.4 Limitation

Currently, specific CSS of the Micro Frontend need to be added into the CSS of the Central Frontend. The reason for this is, when you build a React application, all the JavaScript code can be found under the `main.js` file in the `asset-manifest.json`, while all the CSS code can be found under the `main.css` file.

As a possible improvement, the functionality of the Central Frontend can be extended to also load the CSS dynamically.

5.7 Landing Page

The landing page was initially extracted from the Central Frontend of the term thesis and improved upon.

5.7.1 Components

5.7.1.1 SigmaMap

This component has been adopted almost identically from our term thesis and remains mostly unchanged. It provides the map container which will then be rendered in our other viewing components.

5.7.1.2 SigmaGraph

The SigmaGraph component was also repurposed but changed a lot in the end. It contains all the logic to convert the data from Jalapeno API Gateway to suitable visualisation data for the graph to display.

Inputs

The component receives the following data:

- nodes (LsNode)
- edges (LsEdge)
- links (LsLink)
- coordinates (LsNodeCoordinates)

Data conversion

After receiving the above data a conversion logic normalises it and creates a new type called `SigmaNode` which contains all the information that is required to display the data in the visualisation framework.

Address request

With the correct format of data the latitude and longitude of these nodes get used to request all of the geolocation data. The response will now get saved into the type in form of an address model aswell.

Zoom levels

Now that all the data is in place all the point clustering levels are calculated and saved into a map with the levels as keys. The levels calculated are aligned with the address model attributes and look like the following:

- country
- state
- county
- city
- suburb

- all nodes

Depending on the zoom level of the map the correct geographical subdivision is triggered, and the correct dataset is loaded into the graph.

Events

There are multiple clickable elements in the graph which trigger different events. Those events can be registered internally and can be used to execute certain tasks when they are demanded.

Firstly there is `enterEdge` & `leaveEdge` which as their name suggest is the event when you hover over an edge and leave it again. This is only used to visually highlight which edge you are currently pointing on.

Then there is `clickEdge` which listens to a click event on the edge you are hovering on. This triggers a new viewing element which then shows you the link information of this edge.

The event `clickNode` is getting processed by executing a similar functionality as the edge click but instead shows cluster or node information.

Finally, there is the `camera.updated` event which is triggered by movement on the graph canvas. That includes zooming, panning and dragging the graph. This event is essential to a lot of functions like synchronising the graph to the background map and deciding if a new zoom level needs to be rendered.

Reducers

With filtering functionality in place there needs to be a way to display a subset of the data and this is realised by using Sigmas inbuilt reducers. They iterate through all the graph data and takes a callback function on whether this element fits the filtering logic and displaying it or disabling it completely.

Rendering

When all the operations above are finished the graph is rendered with the according dataset passed to it. The size of the nodes are normalised relative to the zoom level. The cluster or node image gets rendered suiting the node type.

5.7.1.3 DashboardInfoCard

This component has the purpose to show information that accompanies the graph component. It adds valuable insights on what data is shown in the topology. It displays the number of nodes and edges and what the current status of the network is based on if there was data loaded and the time the data was last loaded.

5.7.1.4 NodeInfo, MultiNodeInfo, EdgeInfo, MultiEdgeInfo

They are all components which appear when clicking on certain view elements and display different attributes belonging to those clicked elements.

5.7.1.5 FilterCriteria

This component on one side displays the filter criteria and on input sends the filter criteria back to the `SigmaGraph` component where it is used to filter the data and display the partial dataset.

5.7.1.6 Home page

The landing page is rendered as part of the whole experience. It consists of a navigation bar, a side drawer and a content box. The navbar/sidebar combination was directly taken out of the Choc UI documentation. It offers flexbox CSS elements that render a nicely designed landing page template that can display custom items, icons and hyperlinks to different parts of the application.

It also comes with a search bar that can take or supply search data and search responses easily. This feature allows us to integrate an app search directly to the home page accessible from anywhere.

5.7.1.7 Landing page

The ability to display content in the dedicated content box div allows for easy access to changing the page contents dynamically. So the landing page also consists of modular viewing components such as `MapCard`, `DashboardInfoCard` and `AppCard`. Every design component can be designed in its own context and this allows for a lot of customisability in a later progress state in the project where we share components in story book for reusability.

5.7.2 Point clustering

Despite having implemented a point clustering algorithm already an overhaul with a new approach was needed in the bachelors thesis. It involves a completely different way to define centroid nodes in the cluster. Instead of calculating them the aforementioned Nominatim API was used to create centroids and map them accordingly with the nodes.

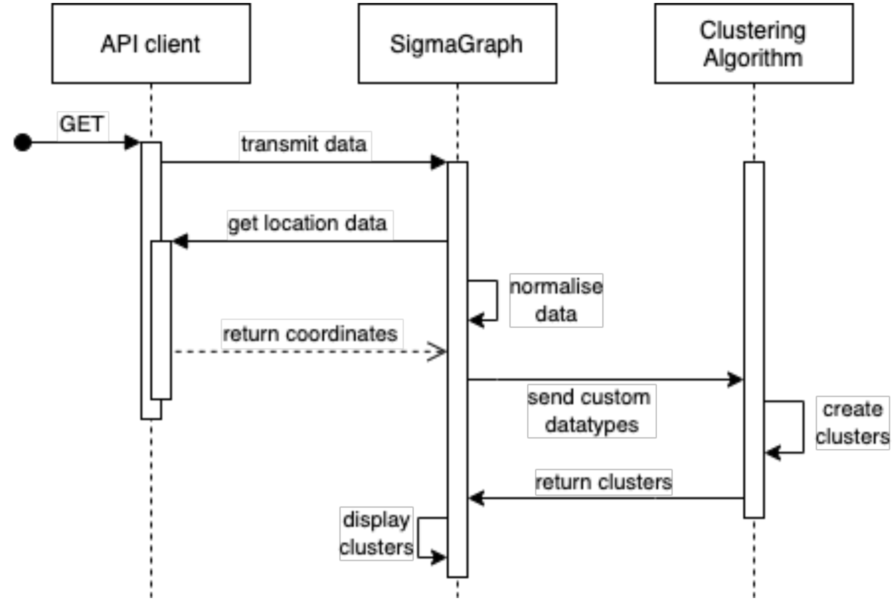


Figure 19: Point Clustering Sequence Diagram
Source: own creation

5.7.2.1 API client

Firstly the API client requests all the nodes and links in the network topology and passes them to the Graph component. It then receives all the coordinates for the nodes to request the exact address model from the Nominatim API and saves it inside the custom node type next to the LsNode received from Jalapeno API Gateway.

5.7.2.2 Custom Node Type

For saving the data the Graph component receives the raw `LsNode` objects and wraps the into the custom `SigmaNode` type for further processing. These `SigmaNode` objects then get populated with the address data received by our Nominatim API instance.

5.7.2.3 Graphology camera ratio

Sigma uses graphology for all the controls used to change the zoom or position in the graph. The graph has a ratio, that is changing by zooming in or out of the printed graph and it changes

linearly. This ratio is used to define which granularity of the point clustering algorithm is triggered and returned. E.g. if the ratio is greater than two we are zoomed out almost all the way on the map and therefore we want to trigger the country level point clustering.

5.7.2.4 Node Point Clustering

The clustering algorithm receives the custom node types as an array and starts iterating over all of them. The provided ratio is used to decide which attribute of the address model is used to define cluster centroids. A cluster map is created which contains a unique key with the correctly assigned nodes as values in an array. With this data structure the graph can print all the keys and still has the according nodes saved in the structure for later use.

5.7.2.5 Edge Clustering

The edges between the nodes now have to be ported over to visualise the clustered connections. This is done by iterating through all connections and saving unique copies of them per cluster. So the algorithm is supplied with the data structure created before and iterates all of the clusters with their contained nodes and checks if the unique connection is already created and if so it doesn't add it a second time.

5.7.3 Filtering

The filtering functionality is split into two categories. There are view filtering options which engage or disengage certain view elements and viewing functionalities like the edges and the clustering mechanism to have a better visual overview in crowded networks. Then there are the data filtering options which enable users to display only a certain subset of the network topology for better data analysing capabilities. This feature is essential for larger networks as it makes it possible to gather the information you need by decreasing the size of the viewed dataset.

5.7.3.1 View Filtering

If the network topology has a lot of links between the nodes the render can quickly become very crowded and confusing to look at. Thats why there are two options added to the view filtering category in our filter area.

The first one being the edge display option which when activated renders all the edges and if deactivated renders none at all leaving only nodes or clusters behind.

The second option is the cluster algorithm viewing. It controls if clusters should be calculated or if all the nodes in the network should be rendered at the same time wether you zoom all the way in or out. This gives the user the choice to overview all nodes in the network with the restriction of longer loading times and possible lag because of insufficient computing power.

5.7.3.2 Data Filtering

Because of the same reasons the view filtering was implemented there are data filtering options in place. Viewing only a subset of the available data makes the topology more useful in general. E.g. you want to see how an ASN is set up, so you filter by ASN. This is done by using Sigmas node reducer functionality. The node reducer can be set on the renderer component and can selectively enable or disable certain nodes in the graph resulting in a `Partial<NodeDisplayData>` object that refreshes the graph with the subset of the data. This feature was implemented fairly easy thanks to the Sigma visualisation framework.

5.7.4 Leaflet / Sigma synchronisation

The Leaflet map layer lives behind the various sigma layers and is therefore completely covered by it. This has the consequence that it cannot directly receive the drag events triggered from dragging around the sigma graph. Also, the geo coordinates are not applicable directly to the sigma graph as it relies on x and y coordinates differing in every generated graph.

So the goal here was to calculate latitude and longitude coordinates to pixel coordinates to display them correctly on the map. This happens by taking the pixel size of the div, displaying the graph and projecting the coordinates to this height and length. This can be done with the project- method that leaflet provides. The other way round this works with the unproject method.

Now when you start moving and zooming the graph those events need to be transferred to the map. This works by getting the top left corner and the bottom right corner of the displayed viewport from the graph and calculating the geolocations of those points. Those will be passed to the leaflet method fitBounds or when zooming flyToBounds which then displays the map correctly according to the following graph.

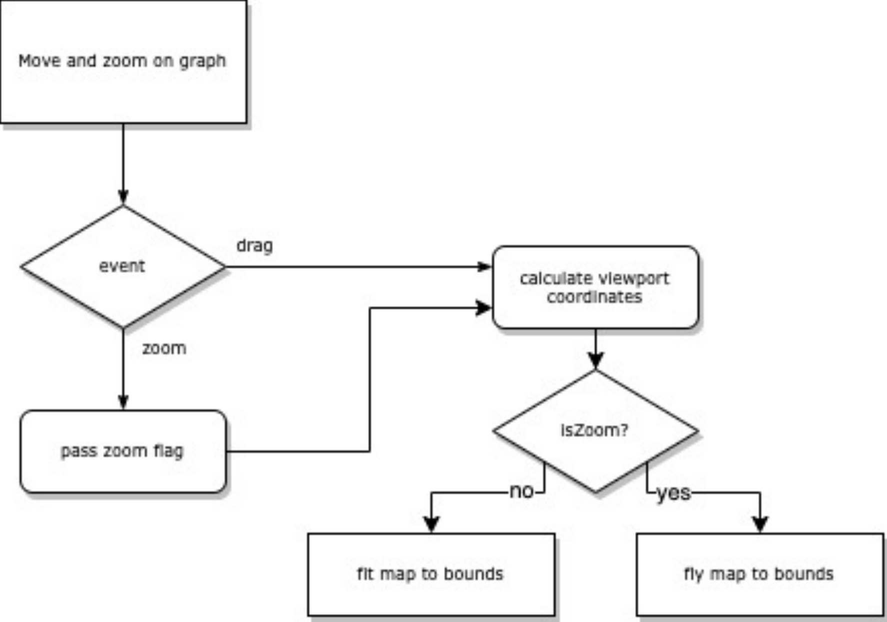


Figure 20: Map projection
Source: own creation

5.8 Sample SR App

To show how a potential Segment Routing Application would be implemented and connected to the Central Frontend, a sample Segment Routing Application was implemented and a sample Micro Frontend was created.

5.8.1 Golang Sample SRApp

There was an existing sample Segment Routing Application found here[5]. The Segment Routing Application just made an API call to Jalapeno API Gateway and printed the LsNodes to standard out. The modified sample app instead allows for an HTTP GET call and lists all the LsNodes in the response JSON.

5.8.2 Micro Frontend Sample SRApp

The Micro Frontend for the sample Segment Routing Application was templated using the Segment Routing Application create-react-app template. The frontend itself has two buttons, both of which are part of the `srapp-lib` UI libary. One button makes an HTTP GET call to the sample Segment Routing Application and lists the LsNode names. The other one deletes the list.

[5]https://github.com/jalapeno-api-gateway/sample-sr-apps

5.9 Review

There were a lot of challenges in this project. In the following sections we will elaborate on them a bit more and get to know which ones had the biggest impact on the success of this project.

5.9.1 Central Frontend

Generally, the implementation of the Central Frontend was smooth and no problems arose.

5.9.1.1 Micro Frontends

The rendering of Micro Frontends has worked flawlessly and already showed its value. One part of the team was working on the Central Frontend and the other part was working on the landing page without any disturbance from the other side or any merge conflicts.

5.9.1.2 Chakra UI

Working with Chakra UI/ChocUI has been smooth for the most part. The only challenge was pinning versions of the different Chakra components to older versions, as the newest version works only with React version 18.

5.9.2 API

Generally, the implementation of the API was smooth and no problems arose.

5.9.2.1 MongoDB Golang Driver

Working with the official MongoDB library has been very rewarding and was definitely the right choice. The library follows Go best practices and seems performant.

5.9.2.2 Images and GridFS

The MongoDB Golang Driver supports GridFS out of the box and allows the upload of a file by passing a stream, which is exactly what the MultiPart form of the Gin provides. So the upload of an image is very straightforward.

5.9.3 Micro Frontends

Implementing the Micro Frontends has been also a satisfactory experience. The distributed frontend approach has shown already its value. By splitting up the code base, multiple people can work on a single frontend without caring about merge conflicts. Additionally, if an issue arises in a Micro Frontend, the Central Frontend still works.

5.9.3.1 UI library

While the UI library and the Storybook has been set up, it currently has only a couple buttons as components. More components should be added.

5.9.3.2 create-react-app template

The template for create-react-app is working effectively and sets up the Micro Frontend correctly. This improves the developer experience substantially.

5.9.3.3 CORS

One negative aspect of a distributed frontend is that by default you will run into CORS errors, since the origin of the request is different.

Therefore, we had to add the `Access-Control-Allow-Origin` header to allow all origins.

5.9.3.4 CSS and config.js

While the JavaScript code of the Micro Frontend is being dynamically loaded into the Central Frontend, the CSS needs to be added explicitly in the `App.tsx` of the Central Frontend. However, this can be solved with future effort outside of this thesis.

The same goes for the config.js which hold the environment variables. The environment variables need to be added to the Central Frontends config.js.

5.9.4 Landing Page

5.9.4.1 Clustering

One of the bigger challenges was to work out a good way to create multiple levels of point clustering on our nodes. This consumed plenty of time in the elaboration phase to evaluate the best fit for it. The outcome was an API which returns structured addresses was just a perfect fit. For this Nominatim reverse geocode API was chosen. That meant it needed to be self-hosted because of API rate limits. Following problems arised:

- The installation and importing of the planet OSM data proved to be very resource heavy, and it took some time to correctly set up the reduced Europe data.
- The clustering algorithm still has some minor bugs when a so-called rebound happens while zooming into or out of the graph. This happens as you zoom and just barely get into the threshold of a different cluster level, and it quickly changes back and forth between two levels.

Those issues were mostly solved or bypassed and are no longer restrictive to the use of our application.

5.9.4.2 Filtering

The filtering component was a new challenge which we implemented in the scope of this bachelors thesis. It was rather straight forward to implement the component itself, but the following problems arose:

- It was not clear from the beginning if the data itself should be filtered or if the graph can be edited while already being rendered. After trying different approaches we settled with the internal functionality of Sigma which offers on runtime editing of rendered data.
- It was the plan to dynamically add filter attributes. It was decided against, because it would be difficult and out of scope for the late state of the implementation phase.

Especially the latter had the disadvantage that less filtering options are available to the user, but this functionality can be added later on.

5.9.4.3 Node/Edge Information

The information modals were easy to implement. They just take the component they display information on and show important key attributes. There has been a small issue, where the data for the from and to node for a link was not available and had to be gathered first. This is not a problem anymore.

5.10 Improvements

5.10.1 Central Frontend

5.10.1.1 CSS

The CSS of the Micro Frontends could also be loaded into the Central Frontend. Currently, only the JavaScript code is being loaded in and thus, Segment Routing Application developers need to add not existing CSS to the Central Frontend first.

5.10.2 Micro Frontends

5.10.2.1 CORS

Currently, the `Access-Control-Allow-Origin` header is set to $*$.

The header could be restricted maybe to a specific domain or maybe to a specific HTTP method.

5.10.3 Landing Page

5.10.3.1 Clustered Edges onClick event

Currently, when a clustered edge is clicked, an Infobar is shown for the first edge in the clustered edges because we do not differentiate between unclustered and clustered edges in the onClick handler. This should be improved, by either displaying all edged in the clustered edge or not display anything, unless an unclustered edge is clicked.

5.10.3.2 Maximum/Minimum Zoom

As the main layer we navigate on is the Sigma graph. The leaflet map has no power on how much you are allowed to zoom and the Sigma min/max zoom parameters are relative to the initial graph size. This makes it hard to set a limit for the zoom as it differs from graph to graph. This does not severely impact the usability other than a visual inelegance in the background map layer.

5.10.3.3 Empty Address Model Values

Based on the different conventions countries make up their addresses with it can happen that certain address attributes are not available in the Nominatim API response. At the moment we handle them as `unidentified` entities. This brings up a new problem that a lot of edges are pointing to the same unidentified cluster centroid that is in an incorrect position. This can be handled by either removing those entities entirely from the map or at least find out the correct coordinates for them.

5.10.3.4 Zoom Level Rebound

Sometimes when just done exactly you can land in between two zoom levels, and it starts bugging out. This error is happening because of the way the synchronisation between leaflet and Sigma is done. The behaviour will stay like this unless the synchronisation is fundamentally changed.

5.10.3.5 Centroid Placement

The centroids for clusters are currently placed at the coordinates of the first node in this cluster. Ideally those would be placed in the geographical center of the cluster region. This cannot be done without forward searching those coordinates by the name of the cluster. Unfortunately this is not possible with the reverse geocode API that is in place.

5.10.4 Deployment

5.10.4.1 Central Frontend

Currently, the Central Frontend does not respond immediately on the deployed Kubernetes clusters. Sometimes a refresh is necessary. The unresponsiveness only happens on the deployed environment. Locally it works fine. This can and should be further analysed and maybe LivenessProbe should be added to the container. This could be due to the Nginx configuration or due to the Kubernetes cluster.

5.10.4.2 MongoDB

Currently, the MongoDB is deployed through a self created Helm chart and suffers from random network timeouts. The upstream Helm chart should be considered. Additionally, if the underlying CPU is updated, MongoDB version 5 can also be considered.

5.10.4.3 Bypassing issues

To bypass the deployment issues, the Central Frontend can be started locally. To start the Central Frontend locally, the guide in the README.md of the Central Frontend can be followed.

6 Conclusions

6.1 Results

This bachelor thesis has resulted in a successfully developed product which can be used in a variety of applications. This can be verified by looking at the defined requirements that have been set in the beginning.

The application has been deployed to both environments:

- `http://sr-000167.network.garden:30000/`
- `http://sr-000169.network.garden:30000/`

In addition, the following items have been deployed:

- MongoDB on both environments
- API `http://sr-000167.network.garden:30001/`
- API `http://sr-000169.network.garden:30001/`
- Landing Page `http://sr-000167.network.garden:30002/`
- Landing Page `http://sr-000169.network.garden:30002/`
- Sample SR App Micro Frontend `http://sr-000167.network.garden:30003/`
- Storybook `http://sr-000167.network.garden:30004/`
- Storybook `http://sr-000169.network.garden:30004/`
- Sample SR App `http://sr-000167.network.garden:30005/`

6.1.1 Remark regarding Deployment

There are some issue with the deployment of the Central Frontend.

To bypass the deployment issues, the Central Frontend can be started locally. To start the Central Frontend locally, the guide in the README.md of the Central Frontend can be followed.

6.1.1.1 MongoDB

Both deployed environments suffer currently from MongoDB connection timeouts. This is due to the Central Frontend helm chart deploying the MongoDB instance in a custom Deployment manifest.

The connection timeouts of the MongoDB cause the API to sometimes return a 500 error. This sometimes causes unexpected behaviour (e.g. No Landing Page registered, Pictures not showing).

This should be fixed by either deploying the MongoDB with the official Helm chart or by hosting the instance externally (e.g. MongoDB cloud).

6.1.1.2 Nginx

The Central Frontend sometimes does not load immediately when opened in a browser and a refresh is needed. This could be either due to the Nginx configuration or an issue with the

Kubernetes cluster. This needs to be further analysed.

6.1.2 Use Cases

All the required use cases have been implemented, and some of optional ones as well. The left out optional use cases had to skipped because of timely reasons. Priorities were set on the important, required use cases primarily.

#	**Use Case**	**completed**
UC01	View Landing Page	✓
UC01-1	Landing Page - Map	✓
UC1-2	Landing Page - Information of Links and Nodes	✓
UC1-2-1	Additional added value statistics	✓
UC1-4	Landing Page - Filtering	✓
UC1-4-1	Filtering - View Filtering	✓
UC5	Register and show/link other SR apps	✓
UC5-1	CRUD SR apps dynamically	✓
UC6	Port over the Service Programming application	optional
UC7	Manage favourite SR apps	optional
UC8	Identity and Access Management	optional
UC9	Scaffold new Microfrontend	✓
UC10	UI Library	✓

Table 8: Use case completion

6.1.3 NFRs (Non functional requirements)

For the non-functional parts of our application all the requirements were completed. The algorithm scale has subproblems like the edges that scale with $O(n^2)$ but others that scale with $O(n)$ and it is difficult to generalise this requirement into one scale. Nevertheless, it is checked because the speedtest shows that it is performing well, and therefore it is a pass.

#	Requirement	completed
NFR1	Clustering algorithm scales with O(n * log n) or better.	(✓)
NFR2	Documentation and reusable building blocks for developing compatible SR apps exist.	✓
NFR3	The application needs to be developed cloud-natively. Meaning it has to run in a Docker.	✓
NFR4	The application needs to easily scale horizontally and be able to be deployed onto any Kubernetes cluster.	✓
NFR5	The renderer should display at least 1000 nodes and 10000 links. This can be achieved by using a webGL framework for client side graphics assisted calculation.	✓
NFR6	It needs no more than 3 clicks to get to the desired SR app.	✓

Table 9: Non Functional Requirements

6.1.4 Speedtest

The goal was to achieve loading times lower than 10s for a thousand nodes. The performance vastly changes with the device you are using. In the table there are the results of our speed tests:

#	No. Nodes/Edges	MacBook integrated graphics	Computer with dedicated GPU
1	200 Nodes / 200 Edges	2s	1.9s
2	1000 Nodes / 10'000 Edges	6s	5.3s

Table 10: Speed Test

The above tests show that the goal was reached with no problem. Even at a thousand nodes on a laptop without dedicated graphics the nodes load in 6s. These results differ from the term thesis as there now is a bottleneck of the Nominatim API hosted on INS infrastructure. The loading times are due to either the resources or configuration issues.

6.1.5 System Test

The system test covers all of the functionality in detail and was performed after the implementation phase.

6.1.5.1 Open Central Frontend

Description	Open the running Central Frontend
Expected Result	Central Frontend launched with Navbar and Call to Action to configure Landing Page
Pass	✓
Further remarks	-

Table 11: Test Case

6.1.6 Click Central Frontend logo

Description	Click Central Frontend logo
Expected Result	Central Frontend routes to /
Pass	✓
Further remarks	-

Table 12: Test Case

6.1.6.1 Click Home in Navbar

Description	Click Home in Navbar
Expected Result	Central Frontend routes to /
Pass	✓
Further remarks	-

Table 13: Test Case

6.1.6.2 Click Apps in Navbar

Description	Click Apps in Navbar
Expected Result	Central Frontend routes to /apps
Pass	✓
Further remarks	-

Table 14: Test Case

6.1.6.3 Click Landing Page in Navbar

Description	Click Landing Page in Navbar
Expected Result	Central Frontend routes to /landingpage
Pass	✓
Further remarks	-

Table 15: Test Case

6.1.6.4 Click Configure on not registered Landing Page

Description	Click Configure on not registered Landing Page
Expected Result	Central Frontend routes to /landingpage
Pass	✓
Further remarks	-

Table 16: Test Case

6.1.6.5 Register Landing Page

Description	Register a landing page in the Landing Page component
Expected Result	Landing Page registered
Pass	✓
Further remarks	Landing Page URL: sr-000169.network.garden:30002

Table 17: Test Case

6.1.6.6 Landing Page zoom out

Description	Zoom out in the Landing Page and pan on the map
Expected Result	Graph is displayed correctly and is clustered
Pass	✓
Further remarks	-

Table 18: Test Case

6.1.6.7 Landing Page zoom in

Description	Zoom in the Landing Page and pan on the map
Expected Result	Graph is displayed correctly and is clustered
Pass	✓
Further remarks	-

Table 19: Test Case

6.1.6.8 Landing Page click node

Description	Click a node in the graph
Expected Result	Clustered node: list all nodes in cluster. Single node: show information
Pass	✓
Further remarks	-

Table 20: Test Case

6.1.6.9 Landing Page click edge

Description	Click an edge in the graph
Expected Result	Clustered edge: nothing happens. Single edge: show information
Pass	✓
Further remarks	-

Table 21: Test Case

6.1.6.10 Landing Page click multi edge

Description	Click a multi edge in the graph
Expected Result	List all links in multi edge
Pass	✓
Further remarks	-

Table 22: Test Case

6.1.6.11 Landing Page disable clustering

Description	Disable clustering for the graph
Expected Result	No matter zoom level, the graph is unclustered
Pass	✓
Further remarks	-

Table 23: Test Case

6.1.6.12 Landing Page re enable clustering

Description	Re enable clustering for the graph
Expected Result	The graph is clustered correctly
Pass	✓
Further remarks	-

Table 24: Test Case

6.1.6.13 Landing Page disable edges

Description	Disable the edges for the graph
Expected Result	The graph does not display edges
Pass	✓
Further remarks	-

Table 25: Test Case

6.1.6.14 Landing Page re enable edges

Description	Reenable the edges for the graph
Expected Result	The graph does have display edges
Pass	✓
Further remarks	-

Table 26: Test Case

6.1.6.15 Landing Page filter node

Description	Filter for a node
Expected Result	Only the node is displayed
Pass	✓
Further remarks	Node: 110

Table 27: Test Case

6.1.6.16 Landing Page filter area

Description	Filter for an area
Expected Result	Only the nodes in the area are displayed
Pass	✓
Further remarks	Area: 1

Table 28: Test Case

6.1.6.17 Landing Page filter asn

Description	Filter for an ASN
Expected Result	Only the nodes in the asn are displayed
Pass	✓
Further remarks	Area: 103

Table 29: Test Case

6.1.6.18 Update Landing Page

Description	Update a landing page in the Landing Page component
Expected Result	Landing Page updated
Pass	✓
Further remarks	Landing Page URL: localhost:30002

Table 30: Test Case

6.1.6.19 Delete Landing Page

Description	Delete a landing page in the Landing Page component
Expected Result	Landing Page deleted
Pass	✓
Further remarks	-

Table 31: Test Case

6.1.6.20 Register SR App

Description	Register a SR App
Expected Result	SR App created
Pass	✓
Further remarks	One standalone sr app: google.com and one micro: sr-000169.network.garden:30002

Table 32: Test Case

6.1.6.21 Launch SR App

pbox

Description	Launch a SR App
Expected Result	SR App launched
Pass	✓
Further remarks	One standalone sr app: google.com opened in tab, and one micro: sr-000169.network.garden:30002 launched in Central Frontend

Table 33: Test Case

6.1.6.22 Update SR App

Description	Register a SR App
Expected Result	SR App updated and can be launched
Pass	✓
Further remarks	One standalone sr app: youtube.com and one micro: sr-000169.network.garden:30003

Table 34: Test Case

6.1.6.23 Delete SR App

Description	Deleta a SR App
Expected Result	SR App deleted
Pass	✓
Further remarks	-

Table 35: Test Case

6.1.7 Screenshots

The screenshots are showing all the pages of the finished application in detail, as readers might not have access to the page that is hosted on INS infrastructure.

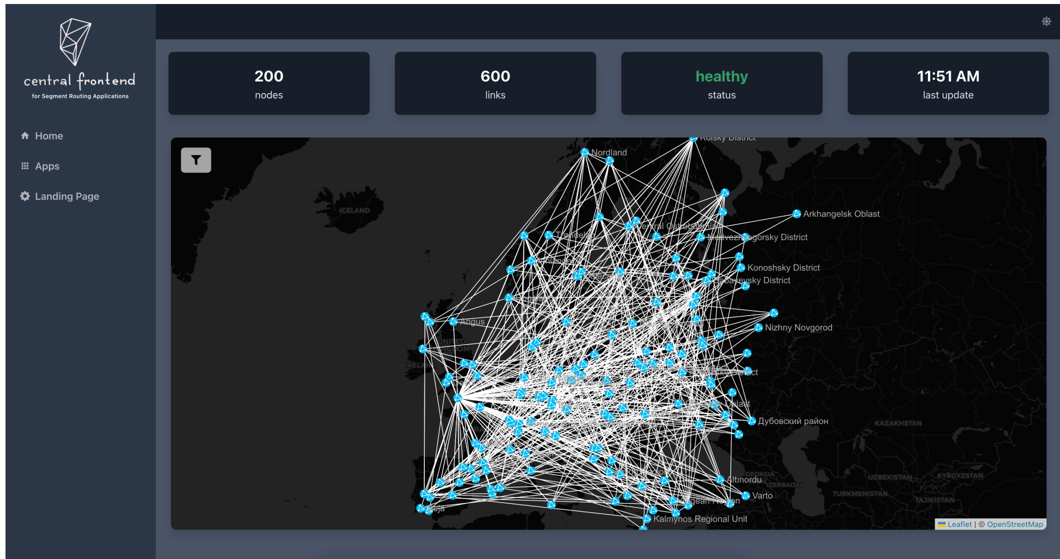


Figure 21: Screenshot 1
Source: own creation

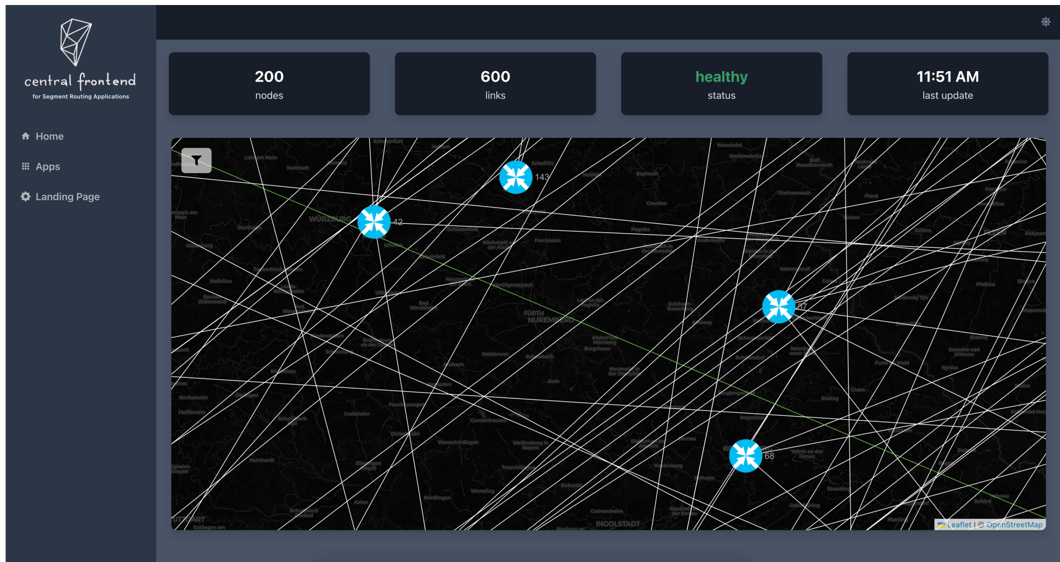


Figure 22: Screenshot 2
Source: own creation

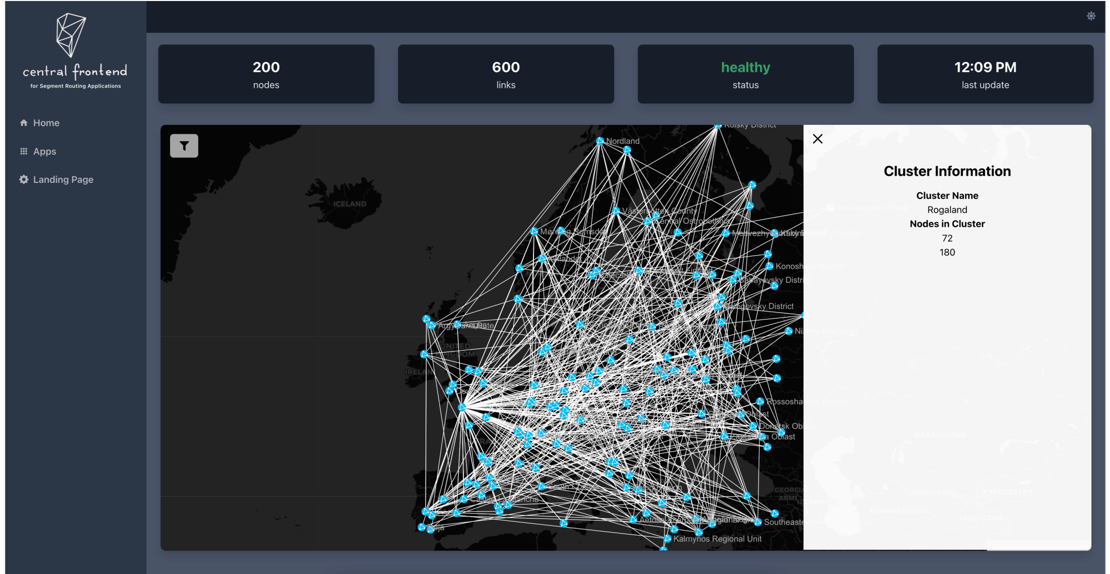


Figure 23: Screenshot 2.1
Source: own creation

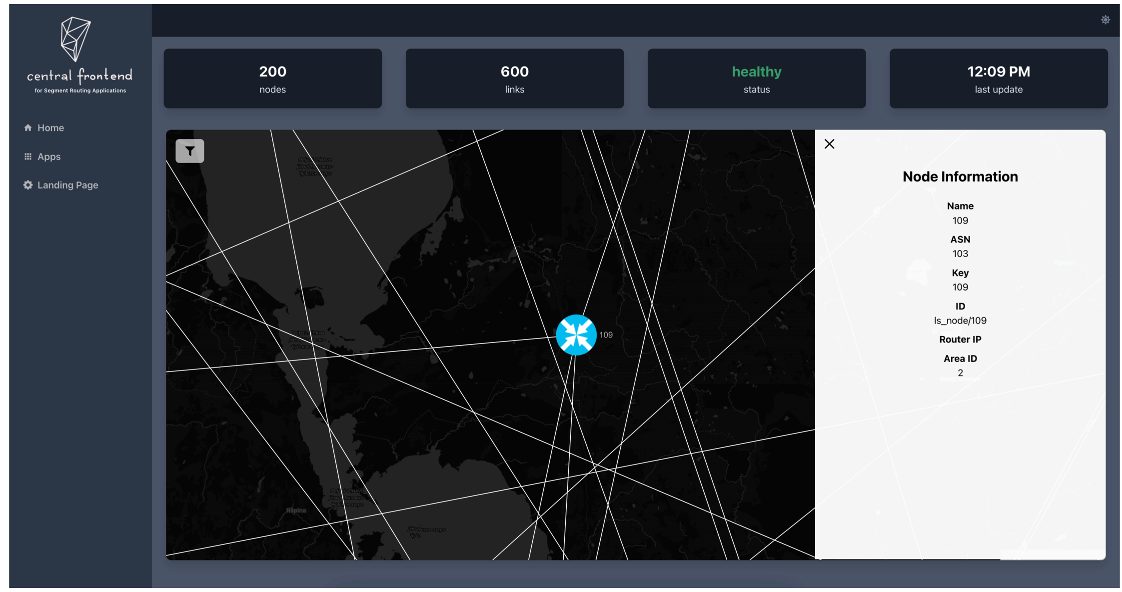


Figure 24: Screenshot 2.2
Source: own creation

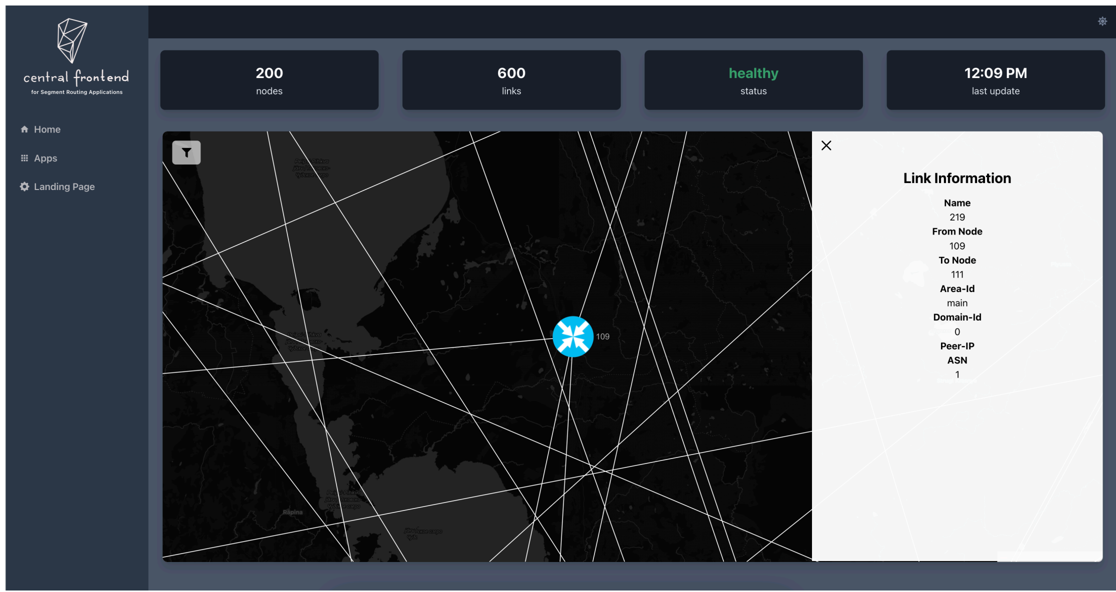


Figure 25: Screenshot 2.3
Source: own creation

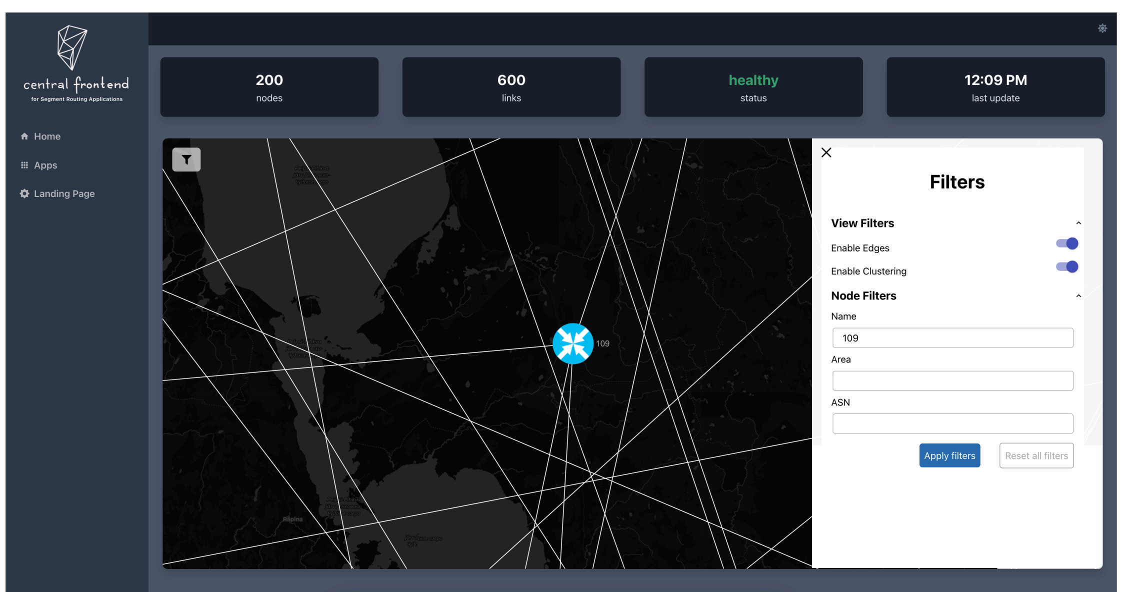


Figure 26: Screenshot 2.4
Source: own creation

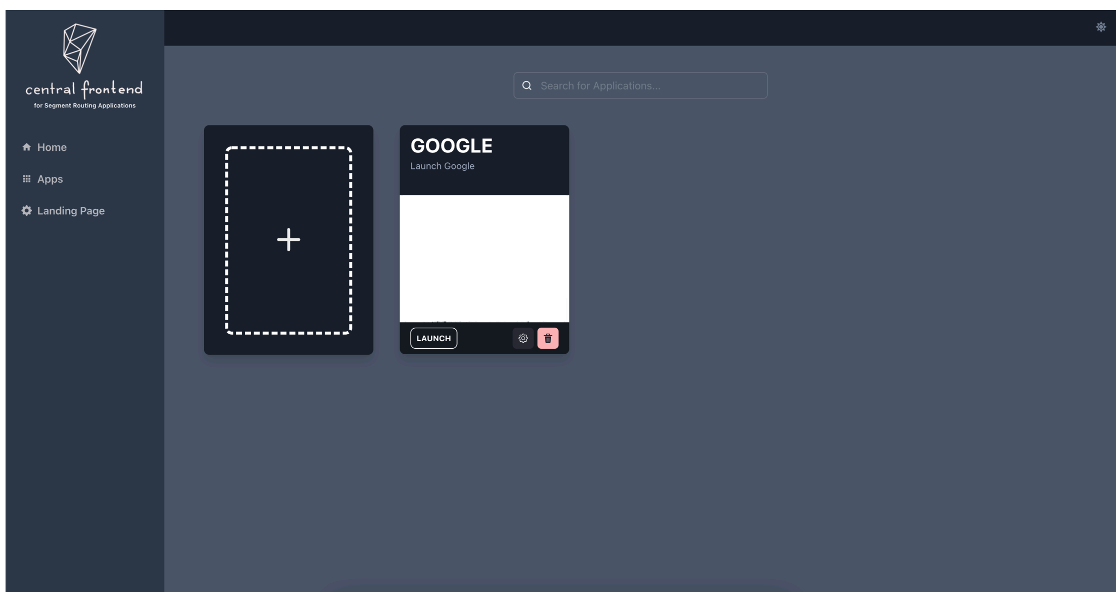


Figure 27: Screenshot 3
Source: own creation

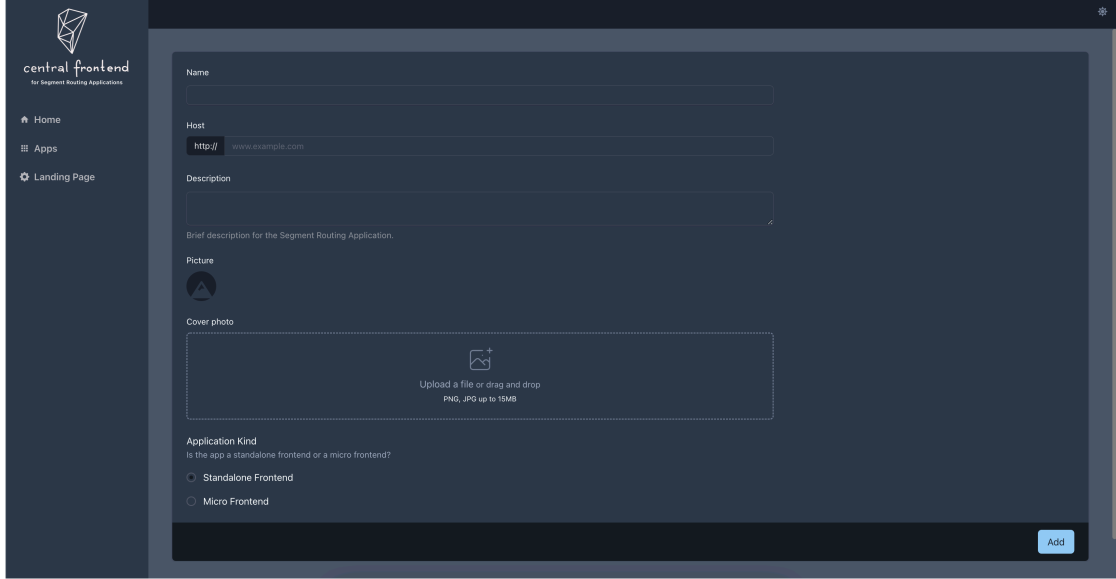


Figure 28: Screenshot 4
Source: own creation

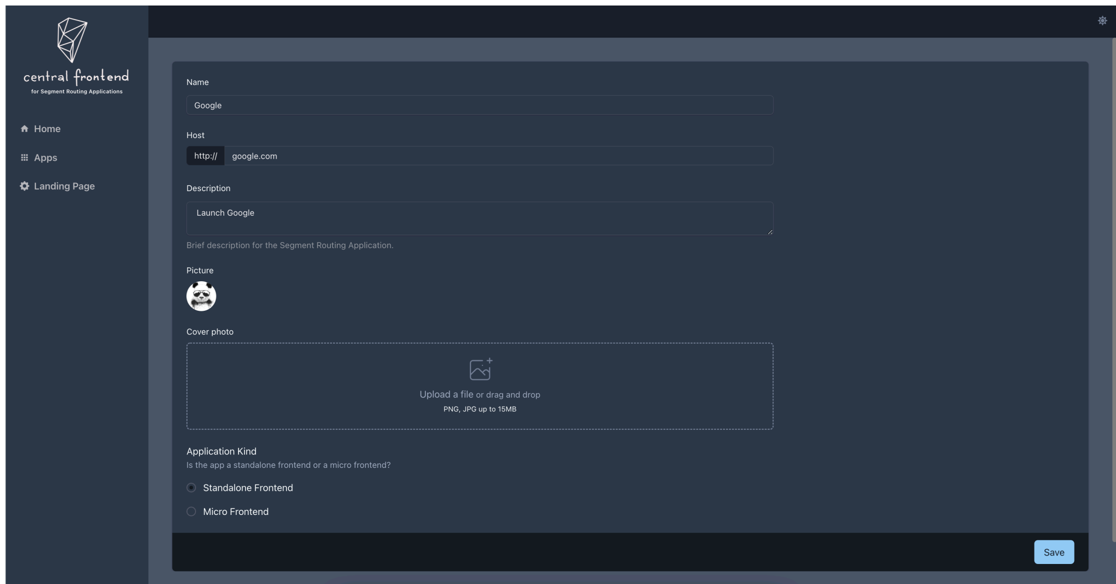


Figure 29: Screenshot 5
Source: own creation

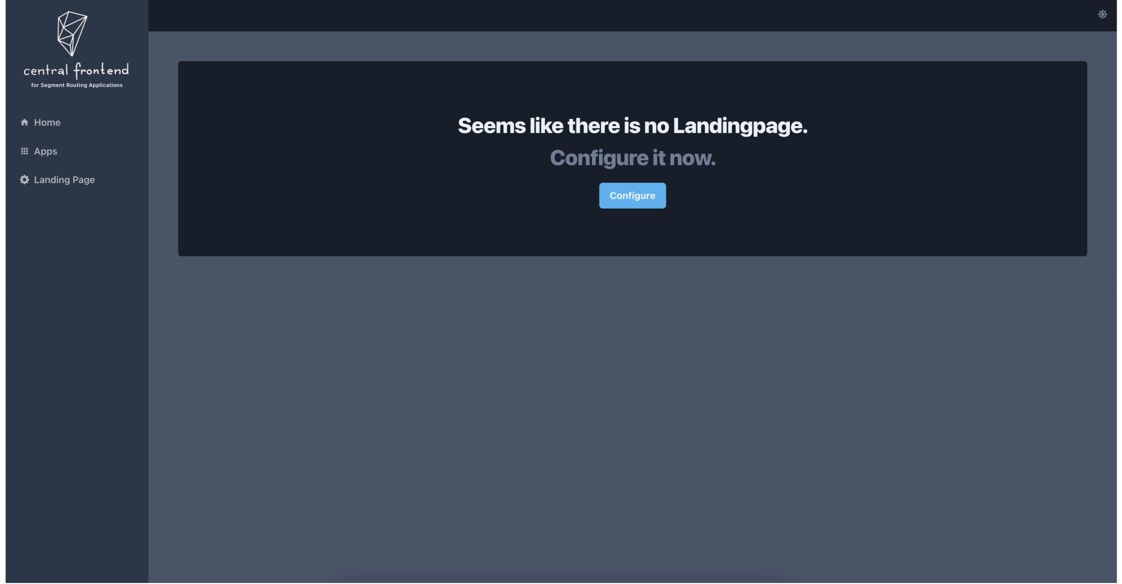


Figure 30: Screenshot 6
Source: own creation

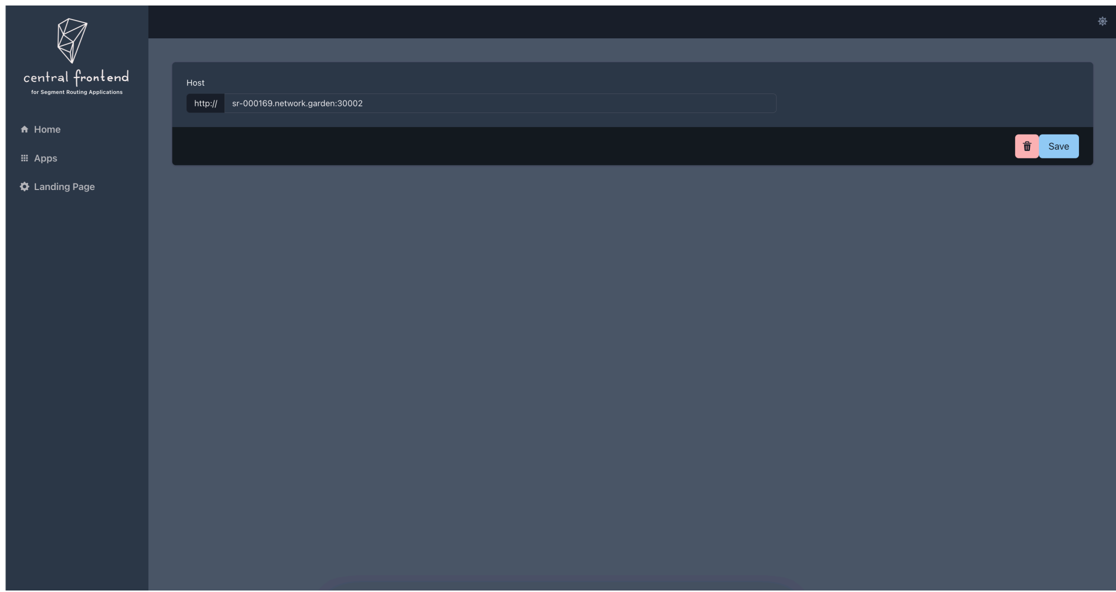


Figure 31: Screenshot 7
Source: own creation

6.1.8 Outlook

6.1.8.1 Central Frontend

The next steps for the Central Frontend should involve open-sourcing it along with the Micro Frontend template. The Central Frontend has the possibility of becoming a Segment Routing community standard. The process of rolling it out to the community, should be planned and some additional work might be required.

6.1.8.2 Landingpage

Leaflet to Sigma Synchronisation

Given the current implementation the panning movement that gets sent to the map includes a zoom movement which can sometimes cause a jitter that moves between zoom levels in the Sigma graph. This logic has to be improved or completely overthought to mitigate this behaviour. The advantages of this approach are a very fast synchronisation time with almost no lag and an easy-to-understand mechanism.

Clustering

The clustering algorithm has proven that it is very suitable to use a reverse geocoding API for geographical point clustering as it returns an address model which is ideal to enrich nodes with geographical data. However, there are a few caveats in terms of completeness of the aforementioned data model. There are different approaches on building addresses all over the world which results in missing data. For the future use of this application one needs to think about the cases where data is missing for the cluster to be built. This was clearly out of scope of this project and our findings can be looked at as an elaborate proof of concept. It was built in a way that every component of the clustering algorithm is interchangeable.

Filtering

The filtering component is working as such and has no problems. However, it can be extended to allow for more flexibility on which attributes can be used for filters. The attributes we chose were the most impactful of them all and therefore were added in the scope of this project. Also, here extensibility is not a problem and functionality can easily be added.

7 List of Figures

1 Use Case Diagram . . . 12
2 Micro Frontend . . . 22
3 GET Request on the API . . . 23
4 Storybook Overview 1 . . . 26
5 Storybook Overview 2 . . . 27
6 Choc UI - Sidebar . . . 28
7 Material UI - Dashboard template . . . 29
8 System Overview . . . 30
9 Dashboard . . . 34
10 Appscreen . . . 34
11 Open App . . . 35
12 Clustering . . . 35
13 Node Information . . . 36
14 Edge Information . . . 36
15 Deployment . . . 37
16 Micro Frontend Sequence Diagram . . . 42
17 The CentralFrontend component in red and the Main component in blue . . . 44
18 Example asset-manifest.json . . . 45
19 Point Clustering Sequence Diagram . . . 56
20 Map projection . . . 59
21 Screenshot 1 . . . 76
22 Screenshot 2 . . . 76
23 Screenshot 2.1 . . . 77
24 Screenshot 2.2 . . . 77
25 Screenshot 2.3 . . . 78
26 Screenshot 2.4 . . . 78
27 Screenshot 3 . . . 79
28 Screenshot 4 . . . 79
29 Screenshot 5 . . . 80
30 Screenshot 6 . . . 80
31 Screenshot 7 . . . 81

8 List of Tables

1 Functional Use Cases . . . 14
2 Non Functional Requirements . . . 14
3 Nominatim Import Styles . . . 17
4 Scalability Test . . . 19
5 Routes . . . 44
6 Fields Standalone Frontend . . . 46
7 Fields Micro Frontend . . . 47
8 Use case completion . . . 67
9 Non Functional Requirements . . . 68
10 Speed Test . . . 68
11 Test Case . . . 69

12 Test Case 69
13 Test Case 69
14 Test Case 69
15 Test Case 70
16 Test Case 70
17 Test Case 70
18 Test Case 70
19 Test Case 71
20 Test Case 71
21 Test Case 71
22 Test Case 71
23 Test Case 72
24 Test Case 72
25 Test Case 72
26 Test Case 72
27 Test Case 73
28 Test Case 73
29 Test Case 73
30 Test Case 73
31 Test Case 74
32 Test Case 74
33 Test Case 74
34 Test Case 74
35 Test Case 75

Glossary

Central Frontend

Central Frontend, the central UI for all segment routing applications 10–14, 20, 21, 23, 24, 30–33, 37, 39, 42–47, 50–53, 60–62, 64, 65, 82, 86

Container

The main web application that renders individual Micro Frontends 21, 23, 42, 86

gin

gin is a high performance web framework written in Go (Golang). 22, 39, 48, 61, 86

Go

Go or Golang is a statically typed, compiled programming language. 22, 31, 39, 48, 61, 86

Go MongoDB Driver

Go Mongodbdriver used for interacting with a MongoDB database in Go. 22, 48, 86

INS

Institute for Network Solutions at OST 10, 86

Jalapeno

A cloud-native infrastructure platform to enable development of network services 11, 86

Jalapeno API Gateway

Single point of access for SR-Apps into the Jalapeno network 43, 52, 60, 86

JavaScript

JavaScript is a dynamically-typed scripting or programming language that allows you to implement complex features on web pages 24, 42, 45, 51, 52, 62, 64, 86

Leaflet

Leaflet is an open source JavaScript library used to build web mapping applications 10, 30, 86

LsEdge

Is a type which represents an Edge in a Segment Routing network 10, 86

LsLink

Is a type which represents a Link in a Segment Routing network 86

LsNode

Is a type which represents a Node in a Segment Routing network 10, 60, 86

Micro Frontend

A Micro Frontend is part of a web application that is run and hosted on a separate web server independently of all other components. The Micro Frontend is rendered by the main Container application. 20, 21, 23, 24, 26, 27, 30, 31, 39, 42–47, 51, 52, 60–62, 82, 86

Micro Frontends

Multiple Micro Frontends 21, 24, 30, 31, 37, 39, 41, 42, 44, 61, 64, 86

micro-frontend

Micro-frontend architecture is a type of architecture where a web application is considered as a composition of features that are owned by separate individual teams. In the micro-frontend approach, you split up your web application into Micro Frontends, that are then rendered by a Container application. 86

MongoDB

MongoDB is a document NoSQL database with the scalability and flexibility that you want with the querying and indexing that you need 22, 31, 39, 48–51, 61, 65, 86

Nominatim

Nominatim (from the Latin, 'by name') is a tool to search OSM data by name and address (geocoding) and to generate synthetic addresses of OSM points (reverse geocoding) 5, 13, 15, 16, 31, 86

Open Street Map

Collaborative project to create a free, editable, geographic database of the world 13, 86

React

React is a free and open-source front-end JavaScript library for building user interfaces based on UI components. 24, 25, 30–32, 42, 43, 45, 51, 52, 61, 86

Scrum+

Scrum and Rational Unified Process in one framework 86

Segment Routing

Segment routing (SR) is a source-based routing technique that simplifies traffic engineering and management across network domains 10, 11, 31, 40, 82, 86

Segment Routing Application

Segment Routing Application is an application that uses Jalapeno data to implement topology or traffic engineering use cases 10, 11, 14, 22, 24–27, 31, 39, 51, 60, 64, 86

Segment Routing Applications

Multiple SR apps 10, 11, 14, 20, 21, 24, 86

SigmaJS

Open source graph visualisation framework 11, 86

SR application

Is an application that communicates with a Segment Routing network 86

Storybook

Storybook is an open source tool for building UI components and pages in isolation. Together with TSDX, a UI library can be built in isolation. 21, 25–27, 86

TSDX

TSDX is a zero-config CLI that helps you develop, test, and publish modern TypeScript packages 25, 27, 31, 86

TypeScript

TypeScript is a programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. It is designed for the development of large applications and transpiles to JavaScript. 24, 25, 30, 31, 52, 86