Danmarks Tekniske Universitet

(02162) Software Engineering 2

SYSTEMS SPECIFICATION

Group C

Christian Harborg (s214967)
Dana Toma (s232042)
Elias Kallioras (s232069)
Florian Comte (s231816)
Lauge Jensen (s214932)
Léonard Amsler (s231715)
Nikolas Xiros (s231920)

December 21, 2023

1 Introduction (Nikolas Xiros - s231920)

This report documents the requirements of the software system which is being developed in 02162 Software Engineering 2.

Starting with a broader context, roads play a crucial role in economic development and growth, providing significant social benefits.

Standard road measures have been developed to ensure proper road conditions and to optimize maintenance strategies, with a primary focus on:

- Safety
- Comfort
- Durability
- Environmental emissions (including noise and CO_2 emissions)

Several factors can contribute to road damage and influence the aforementioned characteristics of a road. Initiating road maintenance as early as possible is of paramount importance since maintenance costs increase exponentially with time.

The dual objectives of this project are to adeptly showcase road condition data through user-friendly features for enhanced navigation and detailed information along roads. Simultaneously, the aim is to aggregate data from diverse providers, encompassing both Lira and Dynatest, to present a comprehensive dataset.

1.1 Background-Main Terms

In this section, we will provide an introduction to some main terms and an overview of the existing background upon which we developed our application.

Firstly,specifying the type of data that is provided and displayed within the final app:

- Indicators,representing a parameter or variable used to measure specific aspects of road conditions
- Images of different type related to the current position

The data format entails that for each point, there is a corresponding indicator value (or an image from Dynatest, as will be discussed later).

We designate points as "geo-referenced" or "mapped" when they possess specific geographic coordinates, enabling accurate location-based information that can be linked to the internal coordinate system of a digital map.

Another very useful term that we will use extensively,is a trip. A trip must adhere to a specific format and is composed of files containing data related to a chosen road path.

Concerning the pre-existing Lira application, it is crucial to comprehend its functionality, wherein data can be displayed as a colored overlay on the map, offering users a general overview of road conditions,but not organized in roads as we will analyze more in depth later.
Furthermore, the Dynatest database contributes information on road conditions, encompassing details such as potholes, cracks, and surface quality, along with images. Notably, these images are available in the form of trips, a feature that the Lira Map currently lacks but that we aim to incorporate into our final application.

1.2 Motivation

The primary goal of the resulting software system is to improve the user interaction with the existing data, providing a practical and user friendly interface for having an overview of data regarding the conditions of certain roads.
In order to accomplish this, we need to add some key features to the pre-existing project.

- We aim to organize all of our data(both Lira data and data extracted from Dynatest trips) according to the respective roads they pertain to, and additionally, present the data using the corresponding road objects.
- An important addition is to,besides the first main map view already existing in the Lira app,provide a second view in which the user will be able to have an more precise view of the data along a specific road, also providing images of this road when available.
- Provide the user the ability to import data in a specified trip form,and then present all the available data collectively in the app.

All the previously mentioned enhancements collectively contribute to the development of a more practical and user-friendly software system, aimed at maximizing the utility of the existing data.

1.3 Audience

The intended readers of this document are persons working at road maintenance companies. More specifically, it should be read by software engineers to gain a in-depth understanding of the working software and also potential end-users regarding the requirements parts.

1.4 Product Use

The primary use of the final software will be from companies that work on the maintenance of the roads. Using the software they will be able to get a very precise idea of the road conditions along the whole network and then potentially decide what kind of maintenance is needed at each road.

1.4.1 Main concepts-functions

Our project will feature two main pages designed for presenting data, housing the majority of the application's functionalities.

- The first one (seen in 1) will include the Map view from OSM with some layers of the road conditions added as layers. This is very similar to the Lira Map application main page,which is just showing some coloured data on the Map, but with some key differences:
  - All the data will be organized in roads and will be visualized in this format on the map.
  - Incorporate data from Dynatest trips or other provided trips and present them collectively.
  - The ability to get a preview of the data of one trip,when we click on that,without getting in the detailed view described below.
- The second one(as seen in Figure 2), which is also the main and very important addition to the current project, is focused on providing the user with a detailed view of the road data along a selected road. To elaborate further, the included features are:

Figure 1: Main view of the application

- A view of the graphs regarding the indexes that the user will be able to select along the road
- The road images sourced from the Dynatest database, aligned with the graph to assist users in comprehending the data.
- A smaller map component that will be provided for users to navigate through the map and view the selected point on the map from this page too.
- A toolbox that will allow the user to choose the preferred indicators or images.

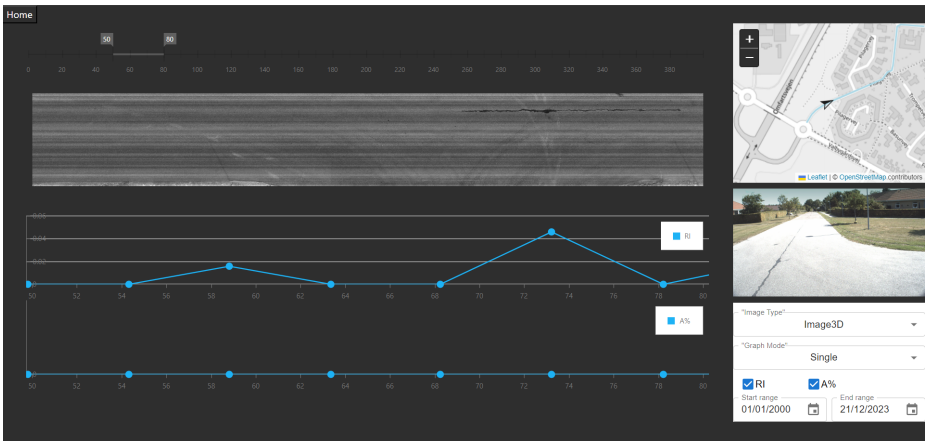

Figure 2: Detailed View of images and indexes graph along selected road

Besides viewing the data with the ways described above, an important function of our resulting application is the user ability to import data in a form of a trip and later view this data on the app.

1.5 Scope

The project primarily focuses on incorporating essential and highly beneficial features, as these are outlined above, into the existing Lira Map application. Simultaneously, it involves integrating new data, including photos, from the provided Dynatest trips. Additionally, it aims to empower end-users by enabling them to import their own trip data.
The collection of data, as well as any potential machine learning procedures associated with it, falls outside the project's scope. The data will be utilized in its original form, sourced from both the Lira and Dynatest databases.

2 System features (Dana TOMA - s232042)

The fundamental purpose of the software, as outlined in Section 1, is the creation of a web application aimed at facilitating the visualization and understanding of road conditions. The software is structured around two distinct pages - the Map page and the Data Page - each designed to address specific user requirements.

The first page, known as the Map page, provides users with a complete view of roads containing data. This page functions as a visual summary, allowing users to choose specific roads, view a synopsis of the available data for a road, search for roads, and import their datasets.

The second page, the Data Page, provides in-depth details about the road, presenting values for various indicators along with images of the road. This page is accessible by selecting individual roads on the initial page. Users can navigate through the road, and access different types of images and graphs based on their preferences.

2.1 Functional Requirements

To ensure the software's functionality aligns with its intended purpose, we established the following functional requirements:

1. The system should allow the user to analyze road indicators by utilizing images and graphs.
2. The system should allow the user to search for roads with available data within the dataset.
3. The system should allow the user to import road data.

2.2 Non-Functional Requirements

To clarify other requirements that are not related to the actual programming of the software, we identified the following requirements:

- Quality: The quality insurance procedures shall follow the list:
  - All development tasks are done in pairs to ensure the detection of common errors.
  - All software components should be tested by at minimum two individuals who were not involved in their development.
  - The development process is divided into smaller releases to guarantee that the software possesses some level of functionality at any given time.
- Development process: The software shall be developed by meeting the following list:
  - Agile development methodology will be followed, with four releases over sixteen weeks.
    - Release 1 by week 41

- Release 2 by week 45
- Release 3 by week 48
- Final Release by week 51

- The team should conduct a retrospective every week to identify areas for improvement.

- Usability: The software aims to provide an effective, efficient, and satisfying experience by following the list:
  - The user interface should ensure an intuitive and user-friendly experience.
  - The system should provide clear and concise error messages to assist users in troubleshooting.
  - New users should be able to understand and use basic features of the application within 15 minutes of exploration.

2.3 Domain Model

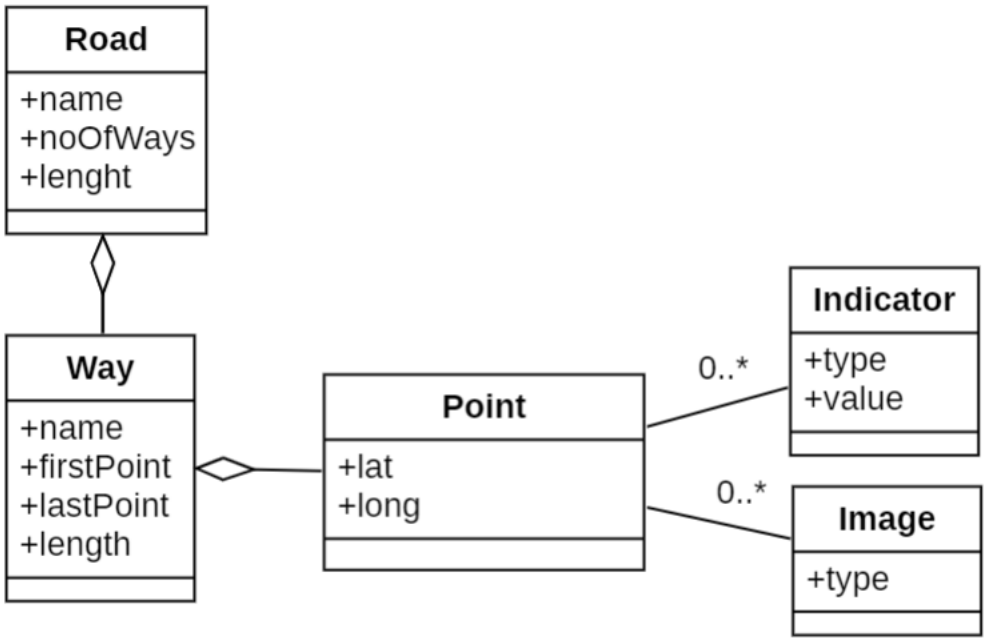

Figure 3: Domain Model

In our domain model 3, we have identified five entities. A "Point" represents a 2D location on a map, characterized by latitude and longitude coordinates. A point may be associated with zero to many indicators and images. An "Indicator" is defined by a type and a corresponding value, while an "Image" is characterized by a specific type.

A "Way" is constructed from an ordered list of points, possessing a name, length, and references to the first and last points in the list. A "Road" is composed of an ordered list of ways that

share the same name. The road entity is defined by a name, the number of ways it comprises, and its total length.

3 Use cases (Christian Harborg - s214967)

Figure 4: Use case diagram

The use cases for this project are illustrated below in the use case diagram. As we can see, there are three main use cases for the road maintainer, which is the import of road data, analysis of road data and the ability to search for a road. Every one of these main use cases has a **includes** relationship which means behaviours that are part of the use case. For instance, if the road maintainer wants to import road data, they have to select a file and upload it to the database after the flask server is done processing the data. Afterword, they select whether to update the database. Therefore, the choice of uploading to the database **extends** the use case since it is conditional whether the road maintainer decides to do so since it might depend on how well the map matching process went.

Another use case is selecting a road and this use case includes the option to then enter a preview of that road section and see a graph displaying road data with indicators for that section. From there, the user can chose whether they want to go to the detailed view where they can see the images of road data (if there are any), graph/graphs with indicator values and navigate to different sections of the road with the slider.

For every use case, we have also made a schema to describe main purpose, the pre- and post conditions.

Table 1: Select a road on the map

Purpose	Get an overview of the road data with available indicator values (KPI, IRI etc.) for the entire road section
Actor	Road maintainer
Preconditions	A map with map-matched road data from Lira and Dynatest with the option to click on the road sections of the map
Postconditions	A graph overview of the data for the section that was clicked on shows up at the bottom of the screen in the preview.

Table 2: Click on a point on the graph in preview

Purpose	To inspect the road in more detail
Actor	Road maintainer
Preconditions	The indicators and data are present on the preview graph and data points are clickable
Postconditions	A mapping of images and data points on the graph which can be analyzed in detail and the images and graphs data is synchronized for the road.

Table 3: Import road condition data

Purpose	To allow the road maintainers to add new data and analyze it in a efficient way
Actor	Road maintainer
Preconditions	Having data from a trip in a specific file format to be uploaded to the database and knowing the password to upload it.
Postconditions	A trip shows up on the main map and the data is able to be analyzed in the same manner as the data that is already existing in the database in the preview and detailed view.

Table 4: Search for a road section on the map

Purpose	To enable users to accurately find parts of the road that they are searching for
Actor	Road maintainer
Preconditions	Road data in postgreSQL database with road names that can be searched
Postconditions	Highlighting the road and navigating to the road section on the main map page

When implementing the different use cases, we had to coordinate the front-end components with the back-end. For instance, when uploading road data, we send a request from the nest-server to the flask server to upload the data and then get await a response which is then sent back to the front end component for sending road data. Most of these use cases involve communication with the nest server and/or flask server in the backend.

4 Graphical User Interface (Lauge Jensen - s214932)

This section contains a description of the graphical user interface (GUI) for our web application. Our web application consists of 2 pages which are accessible to the end user. Images of the 2 pages can be seen below. First we will write about how those 2 pages functions, and afterwards we will mention some of the technologies we used to create the GUI.

Figure 5: Screenshot of Map page

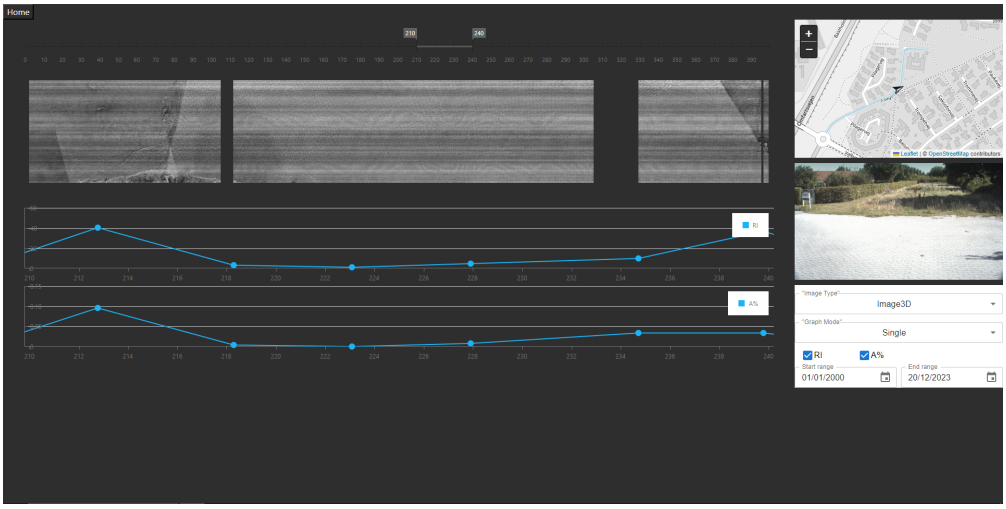

Figure 6: Screenshot of Data page

4.1 Map page

The map pages is the first page, that the user will be directed to when visiting our web application. When the page is initially opened, it will fetch the road data to be visualised from the back end, which will take a couple of seconds, while the site is fetching the data, a spinner is shown in the middle of the page, indicating to the user that the site is in the progress of loading some data. The page primarily consists of 5 components, which will be described in the following 5 subsections.

4.1.1 Map

The map covers the upper 93% of the map page. The map is interactive, such that the user can pan and zoom on the map. On the map, all the road segments which we have data for will be lit up with a light blue color. The map does not create a distinction between LiRA and Dynatest data. When the user then hovers over one of the light blue lines, the line will turn a darker

blue color, and when the user clicks on the line, the preview components will replace the slider component in the bottom of the screen.

4.1.2 Slider

The bottom 7% of the map page is taken up by the Slider component. In the slider component, the user can select what time-range they want the application to fetch data from. On each end of the slider is a black box with a date, which the user can click and drag to another date, after which the site will fetch some new data, which is within the selected time-range. When the page initially loads, the time-range will be set to the date of the earliest labelled data up to the date of the newest labelled data.

4.1.3 Preview

The preview will be shown when the user clicks on of the colored road segments. A screenshot of the preview can be seen in figure 7, in the screenshot the user has clicked on the road segment "Pilagervej", which has then been colored red on the map. The preview shows a graph with the indicator values for the selected road segment. The graph contains all the condition values in a single graph. In the graph, all the data has been normalised to range between 0 and 100. On the map, the user can see a arrow located at the start of the selected road segment. When the user then hovers over a data point in the graph, the position of the arrow will be updated to be positioned corresponding to the hovered data point. If the user then clicks on one of the points in the graph, they will be redirected to the Data page. At the bottom of the preview is a button labeled "Enter detailed view", clicking this button will also redirect the user to the Data page. on the top right corner is a grey cross, clicking this button will exit the preview.

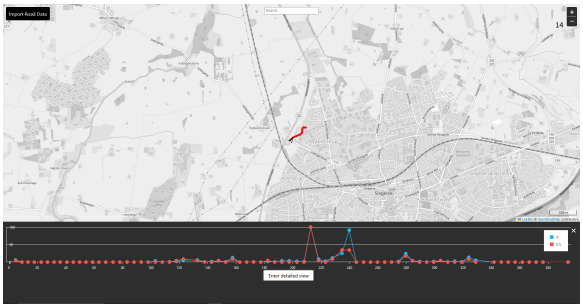

Figure 7: Screenshot of preview component

4.1.4 Road search (Subsection by Christian Harborg - s214967)

In the top-center position of the map, there is a search bar which enables users to search for a road in the road network. Once the user begins to type in the search bar, road names that match the search will appear. When a road is then clicked on, the road will be highlighted in green instead of the usual light blue color and the map re-navigate to the selected road. As seen from figure 8 below the search content is "vestmotorvejen" which is the road that is highlighted in green. This functionality helps road maintainers efficiently filter search for the specific road they want to find.

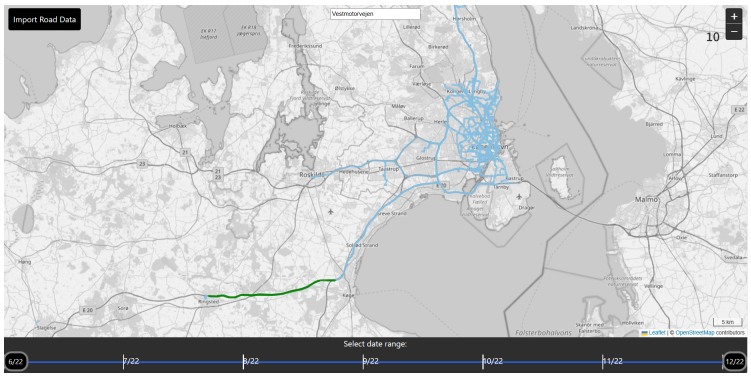

Figure 8: Screenshot of Search Component

4.1.5 Import

At the top left corner of the map is a black button labelled "Import Road Data", when clicking this button, a new window will pop up on top of the map, which can be seen in figure 9. Clicking the button will also apply a blur effect to the map, and make it such that the user cannot pan around the map, or click on a road segment. On this window, the user can import a new road segment contained in a zip file. The process of importing can be found in the handbook.

Figure 9: Screenshot of import component

4.2 Data page

The data page, is the page where the user can get a detailed overview for a selected road segment. The page is accesed from the preview component in the map page. To get back to the map page, the user can click on the home botton in the top left corner. Apart from the home button, the page primarily consists of 6 components, which is visually split into 2 columns with 3 components each.

4.2.1 Selector

The user can use the selector to scroll along the data, by dragging the light grey bar. Under the grey bar is a list of numbers, which tells for many meters from the start of the road segment the data is corresponding to. The furthest number to the right is equal to the max lenght of the road segment. The user can also scroll along the data by using the arrow keys on their keyboard, which will shift the view slightly to the left or right.

4.2.2 Pavement images

Under the selector is a row of pavement images. The images are automatically updated when the user moves the selector. In figure 5, you can see some gaps in the pavements images, this is due to the way we map-matched the data, which is described in more detail in section 5.6. When the page initially loads, there will be a text telling the user that the images are being loaded, or that the road segment does not have any images attached to it. The page will initially load all the different images all at once which can take a bit of time, this ensures that the user experience will be as smooth as possible, as soon as all the images have been loaded.

4.2.3 Graphs

Under the images are the graphs showing the indicator values. When the page initially loads, there will be a graph shown for each indicator value which was recorded for the selected road segment. In figure 5 the data is for the Dynatest survey of "Pilagervej", in that survey there is only recorded 2 indicators, which is why there are 2 graphs. These graphs are also synchronised with the selector. The graph view can be modified by the menus on the right side, which is described in section 4.2.6.

4.2.4 Map

In the top right of the page is a small map, which just shows the selected road segment in a red color, along with a arrow which indicates to the user what point the user is currently showing data for. The user can also go to another point in the data by clicking somewhere on the road segment in the map.

4.2.5 Wide image

Under the map is a image, which is called a Wide Image, which for shows the image located in the "Cam1" folder, which is a zoomed out image, which can give the user to easier be able to tell where they are looks in the real world. This image will only be shown for Dynatest data.

4.2.6 Menu

Under the wide image is a Menu, which allows the user to modify some settings for the data viewing. There are 4 different settings that the user can change:

- **Image Type**: This option allows the user to select what type of images should be used for the zoomed in pavement images. The user is only able to choose between the types which are available for the selected road segment (this could for example be Overlay3D or ImageInt. This option will only affect roads with Dynatest data, since there are no images associated with Lira data. Furthermore, this option will only be visible once the image data has been loaded.
- **Graph Mode**: The user is able too choose between 2 different kinds of graph modes Single or Multiple, Single is the default option, with this option the values for each indicators will be shown in their own seperate graph. If the user instead chooses Multiple, then all the indicator values will be shown in the same graph, where the values are normalized between 0 and 100. This will look like the graph in figure 7.
- **Indicators**: Under the Graph Mode selector, the user can select which indicators they want to show data for. Initially each indicators will be checked on. But the user can just check one or more off, if they don't want to view some specific indicators. Checking off a indicator will result in the graph for that indicator being removed, or it the graph is

in Multiple mode, then the line corresponding to that indicator will be removed from the combined graph.

- **Data range**: In the bottom of the menu, the user can select what date range they want to view data for, which allows the user so see a history of a road segment, in case the road segment has been covered in several surveys. They can do this by selecting a start and end range, which is then used for the data fetching.

4.3 Technologies

Our web page was programmed using the React framework, which we used together with TypeScript, which is a typed language that is an extension of JavaScript. In the project we also used a number of react libraries for the front end GUI, some of them are listed below:

- **React Leaflet:** Used to render an interactive map, which we displayed the LiRA and Dynatest data on.
- **React DevExtreme** Used to render line charts for our data page. Also allowed us to create a slider which lets the user easily scroll along the data.
- **React Material UI:** We used Material UI to help us to easier layout our pages. We mainly used the library on the Data page, where we used the libraries Grid and Stack features to create the layout for the page.
- **React Slider:** We used React Slider to create a slider on the Map page, to enable the user to select what time-range for the data being shown on the map.

5 Software Architecture (Florian COMTE - s231816)

This section delves into the software architecture aspect of our system specification. We will first have an overview of the system, then talk in details about each part of the software.

5.1 Overview of the system (Florian COMTE - s231816)

The architecture of this application is built upon a set of complementary technologies. On the frontend, we employ React, while the backend comprises a NestJS server, a Python server and Valhalla API running on a server. We also have one PostgreSQL database.

Within our system, the React client communicates exclusively with the NestJS server. The NestJS server, in turn, engages solely with the database for read operations. Write operations on database occur exclusively during the import process, a task done by the Flask Server (see section 5.5.1 for more information about importing process). Furthermore, the Valhalla server is accessed only from the Flask server to do the mapping of the data points (see section 5.6 for more information about the map matching).

For a visual representation of the system's key components and their interactions, refer to the component diagram in Figure 10. We will make references to it in the upcoming sections. Further insights into the interactions are detailed in the section 5.4.3.

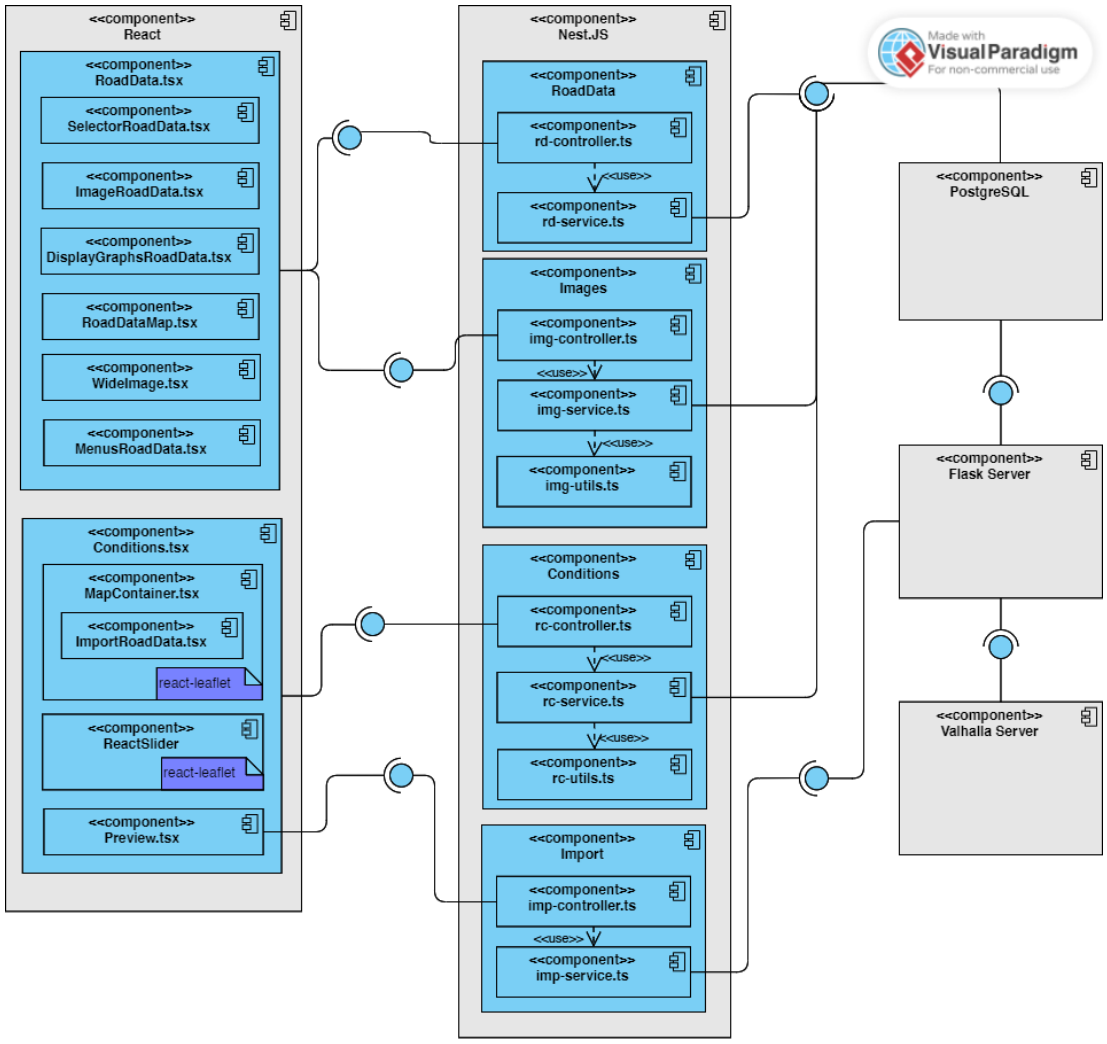

Figure 10: Component diagram of the application architecture

5.2 Database (PostgreSQL) (Florian COMTE - s231816)

Let's begin by discussing the database structure. The database is hosted on the DTU server, accessible through port 5432, and uses PostgreSQL technology. You can find its structure in Figure 11. All measurements related to distance are expressed in meters.

The primary challenge with our data is organizing it according to roads (since it was one of the main user story). In this scenario, we've conceptualized a road as a carefully ordered list of "ways." Each "way" is, in essence, a systematically arranged list of coverage points, with each coverage point representing a geographical location defined by latitude and longitude coordinates.

Utilizing the concept of a "way" allows us to align our data structure with the underlying architecture of OpenStreetMap (OSM), which serves as the foundation for technologies like Valhalla (see section 5.6) and React Leaflet that we integrate into our system.

To establish the order of ways within a road, we leverage the road_index column, assigning 0 to the first way, 1 to the second, and so forth. Similarly, for sequencing coverage within a way, the

distance_way column is employed. Notably, we introduced a distance_road column to simplify frontend display for a given road. This value is calculated during the import process (see section 5.5.1). This eliminates the need for calculating distances from the road's start during render.

Each coverage point is linked to a specific list, encompassing images and coverage values represented by indicator values (e.g., type: KPI, value: 0.5). For instance, a point may possess KPI and IRI values, along with connections to two images. These point/image associations are enhanced with additional details such as date and provider. The length entries in the images table denote the image size in meters, a crucial factor for subsequent scaling in the frontend view.

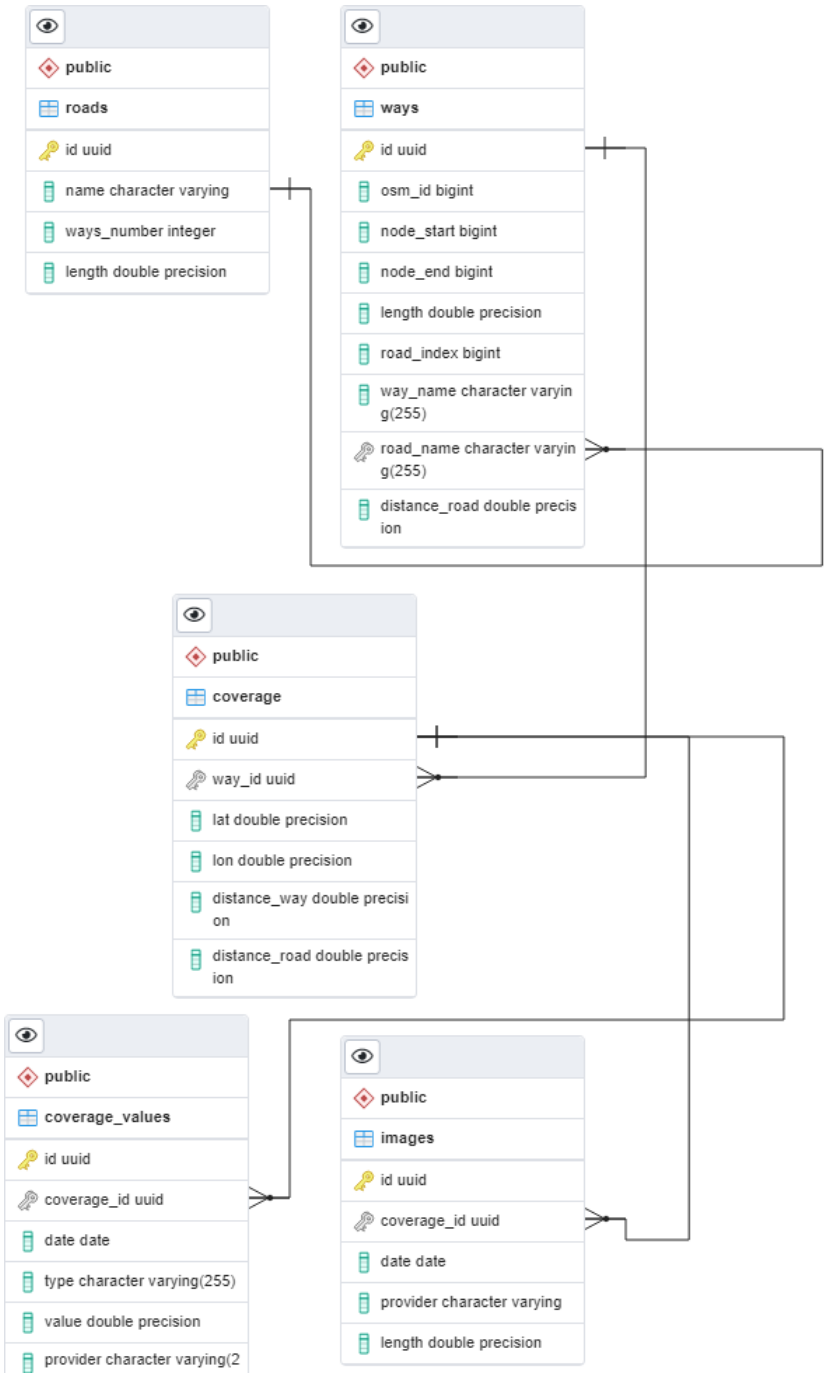

Figure 11: Database structure

5.3 Front-end (React) (Florian COMTE - s231816)

Let's now explore the frontend. It's hosted on the DTU server, accessible through port 3000, and employs React technology (implemented with Typescript). The exchange with backend are done using basic POST/GET request and the precise endpoints will be described in section 5.4.3.

5.3.1 Source folder organisation

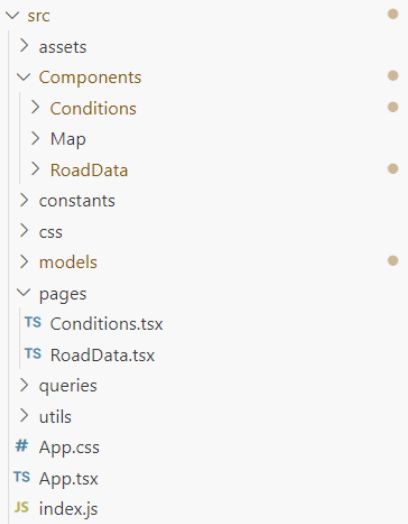

Figure 12: Folder structure of the React APP

The project source is organized into distinct folders as you can see on Figure 12. First, we have our assets, comprising images used as markers (to display the position we are on a road and the direction) on the map. Next, we find all the components essential for building our application. Following that, we have a section dedicated to constants, then CSS files for styling, and definitions in the models folder. Additionally, there are the two pages (which we'll delve into shortly) in the corresponding folder. The structure also encompasses queries responsible for interacting with the backend (see section 5.4.3), along with utility functions in the utils folder. Naturally, the foundational elements of a React application, such as index.json, App.css, and App.tsx, are included.

5.3.2 Routing

Our application features two distinct routes: one for the Conditions page, presenting a global map with comprehensive data, and the other for the RoadData page, designed for road analysis. The Conditions page serves as the default route, accessible when navigating to route which is not the /roaddata one (so the conditions page is the default page). The RoadData route incorporates two parameters: "roadname," denoting the road's name, and "wantedX," representing the distance from the road's start where analysis should commence.

For instance, to analyze Lyngbyvej road starting at 20 meters, the corresponding link would be http://se2-c.compute.dtu.dk:3000/roaddata/Lyngbyvej/20. When you are in the preview on the Conditions page and you click on a point in the graph, it dynamically adjusts the route with appropriate parameters. This functionality is possible because the application has access to the road being previewed and the distance from the start in meters, determined by the point clicked on the graph.

5.3.3 Pages

As depicted in Figure 10, our React application is partitioned into two major components: the RoadData page and the Conditions page.

In both pages, all states are established in the parent components (RoadData.tsx and Conditions.tsx, as illustrated in Figure 10), and each subcomponent gains access to getter/setter functions through their props. However, it's important to note that there is no shared context between these two pages. We display roads on the map using Polyline (React Leaflet Component).

On the RoadData page, we've opted to preload all data, including indicators and images, upon initialization to enhance the overall user experience. This preloading occurs either at the page's loading or when there's a change in the selected date range. The synchronization extends across various components, including the map, graphs, scrollbar, and images. Essentially, the data displayed on the graph corresponds directly to the road image visible above it.

When making adjustments to the map, which involves obtaining updated latitude and longitude coordinates, we traverse the dataset to determine the corresponding segment along the route. This involves identifying the line segment between two data points. Subsequently, we calculate the distance from the starting point, allowing us to update other components (based on distance from start of the road for displaying data).

Similarly, when utilizing the scrollbar and modifying the distance from the starting point, we follow a similar process. In this scenario, we get the coordinates of an interpolated point in the appropriate segment (find based on the new distance from start of the road).

It's worth noting that due to the intricacies of map matching, particularly when utilizing tools like Valhalla (see section 5.6), there might be occasional spacing variations between images or instances of overlapping. This is a result of the imperfect nature of map matching algorithms, and although it may introduce some visual irregularities, it doesn't compromise the overall functionality of the synchronized components.

5.3.4 Models for data

For a detailed view of the data and image models utilized in the application, refer to Figure 13 for data models and Figure 14 for image models.

The RoadDataType serves as the model representing indicator data for a road, including its name, length, and a list of RoadDataTransformedType, which are coverage values (see section 5.2).

Concerning images, the CoverageImagesType encapsulates all images associated with a specific coverage ID, encompassing distance from the road's start, a list of wide images and zoom images (see section 4.2.6. Each image is represented by the ImageType, which includes basic information about the image and, naturally, the image itself saved as a base64 string.

```
export interface RoadDataType {
    road_name: string;
    road_length: number;
    data: RoadDataTransformedType[];
}

export interface RoadDataTransformedType {
    value: number;
    distance: number;
    type: string;
    lat: number;
    lon: number;
    date: Date;
    coverage_id: string;
}
```

Figure 13: Models for indicators data

```typescript
export interface ImageType {
    id: string;
    date: Date;
    provider: string;
    length: number;
    type: string;
    img: string;
}

export interface CoverageImagesType {
    distance_road: number;
    wide_images: ImageType[];
    zoom_images: ImageType[];
}
```

Figure 14: Models for images data

5.4 Backend (NestJS server) (Florian COMTE - s231816)

Let's now explore the backend. First we have a NestJS server which is hosted on the DTU server, accessible through port 3002.

5.4.1 Organization of the server

This server is segmented into four distinct parts, each serving a specific function:

1. Imports: Facilitates message transmission between React and Flask. Flask manages the import-related operations (refer to section 5.5 for details).

2. Images: Empowers users to access all images associated with a given road. Refer to section 5.4.2 for insights into the structure of images.

3. Conditions: Gathers data pertinent to all roads, exclusively utilized in the Conditions page.

4. RoadData: Retrieves data specific to a particular road, exclusively applied in the RoadData page.

Each segment follows a consistent organizational pattern, featuring a controller that defines all relevant endpoints, a service responsible for executing logical operations, and occasionally a utils section that provides utility functions, contributing to a more structured code. The structure is visually represented in Figure 15.

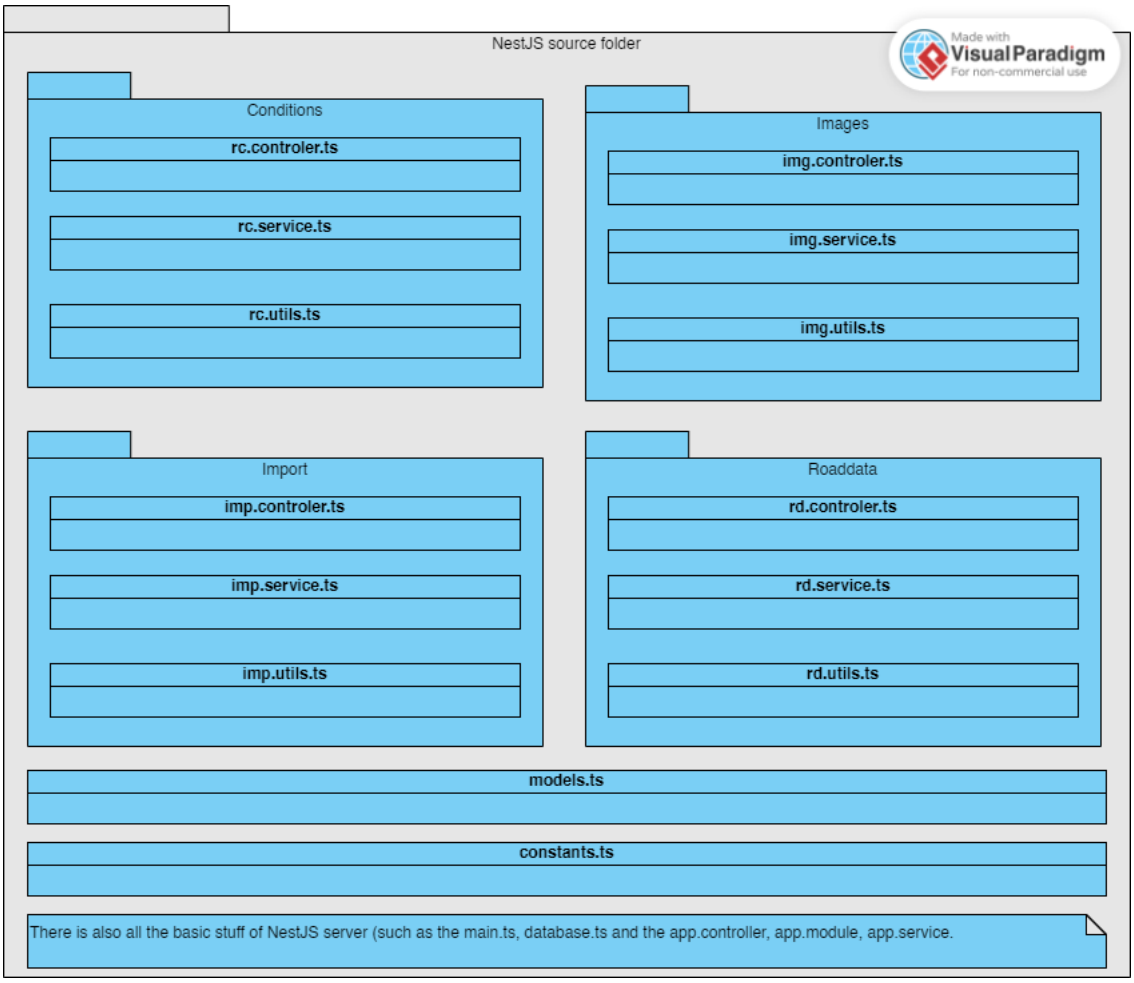

Figure 15: Class diagram of the general structure of the NestJS server

It's worth noting that the models employed here align with those used in the frontend, as detailed in section 5.3.4.

5.4.2 Images storage structure

All our images are stored in the server-nest within a designated "data" folder. An illustrative example can be found in Figure 16.

The organizational structure involves a directory for each coverage, further divided into two subdirectories—one for wide images and another for zoom images. The images are labeled with their corresponding coverage IDs, as depicted in Figure 11. Notably, for zoom images, we explicitly specify their image type (more details about type in section 4.2.6). Upon importation by Flask (see section 5.5.1), all images undergo preprocessing and are saved in the webp format to optimize performance.

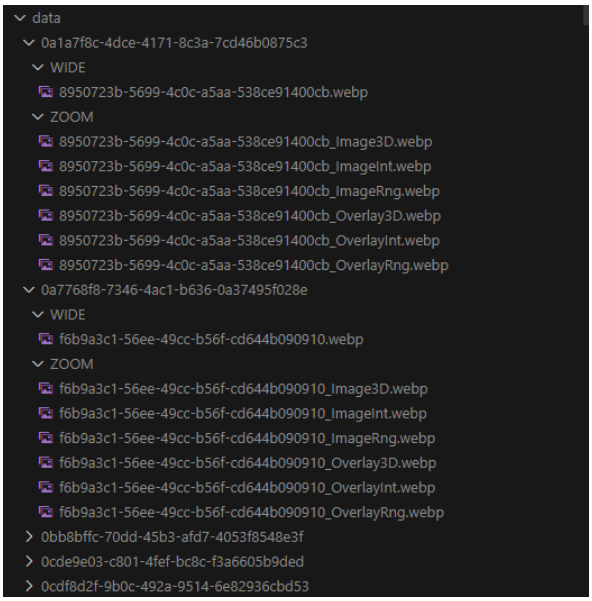

Figure 16: Images storage structure

5.4.3 Accessible endpoints

Outlined below are the available endpoints of our NestJS server:

1. `GET /roads:`
   - Retrieves all roads within a specified date range
   - Requires: Start and end dates.
   - Returns: A list of RoadDataType (refer to section 5.3.4 for details).
2. `GET /roads/dates:`
   - Fetches the date range for all available data
   - Returns: The start and end dates for the road data.
3. `GET /roaddata:`
   - Obtains all indicator data for a specified road.
   - Requires: Road name, start date, and end date.
   - Returns: RoadDataType (refer to section 5.3.4 for details).
4. `GET /images:`
   - Retrieves all images for a particular road.
   - Requires: Road name, start and end dates, image types (see section 4.2.6).
5. `GET /import/checkProcessStatus:`
   - Get the preprocessing status (preprocess data) of a certain import.
   - Requires: ID of the import.
   - Returns: Status of the processing and error message if there is some error

6. `GET /import/checkUpdateStatus:`
    - Get the updating status (update in database) of a certain import.
    - Requires: ID of the import.
    - Returns: Status of the updating and error message if there is some error
7. `GET /import/confirmUpdate:`
    - Send the user choice for updating database or not (see section 5.5 for more informations).
    - Requires: ID of the import, answer of the user.
8. `POST /import/road:`
    - Send a file to be imported
    - Requires: the ZIP file to be imported
    - Returns: ID of the import

For all the import endpoints, see section 5.5 for more details and why we need that.

5.5 Backend (Flask Server) (Florian COMTE - s231816)

The flask server (Python) is used only to do the importing part of the software. It receives data from the backend and interact with the database (write) and the Valhalla server (see Figure 10).

5.5.1 Import process

You can have a look to the complete importing data process on the Figure 17.

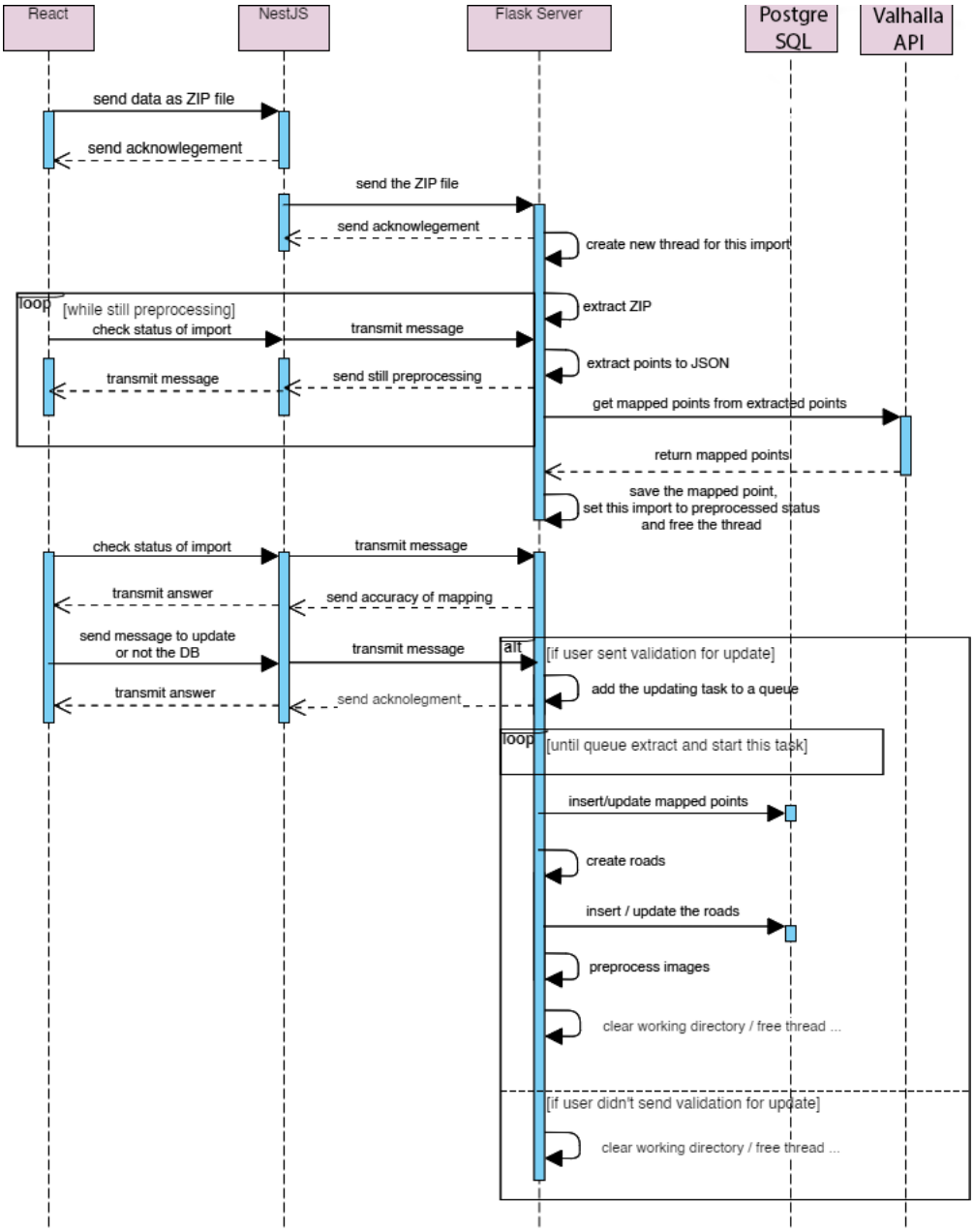

Figure 17: Sequential diagram for the importing process

Note that a password is needed to import data (and asked when uploading the ZIP file and can be defined in the .env of the Flask Server).

Every preprocessing task is executed in parallel, ensuring efficient processing. However, updates to the database follow a serialized approach using a queue, allowing only one update at a time.

Upon receiving mapped points from Valhalla, the system generates accuracy results. These results include the count of successfully matched points (indicating Valhalla's ability to locate a point near the road), the number of unsuccessfully matched points, and the identifiers of the impacted ways. Subsequently, these results are transmitted to the user, who has the authority

to decide whether to proceed with updating the database.

Additionally, the React app initiates a status check every 10 seconds, querying the Flask server for updated information. This periodic communication ensures real-time synchronization between the frontend and backend components of the system.

5.5.2 Road creation

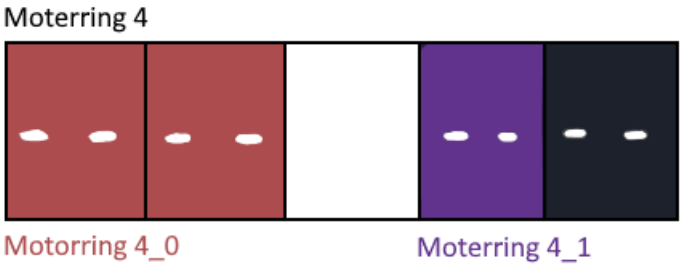

Figure 18: Road creation illustration

We created an algorithm aimed at generating roads based solely on ways. The approach involves utilizing the starting and ending node IDs associated with each way. The concept revolves around identifying common start and ending nodes, allowing us to construct roads accordingly.

However, a limitation arises in this model: for a given road, data may be available only for specific ways. To illustrate, consider Figure 18, where data exists for the first two ways and the fourth way of the Motorring 4 road. In response, our algorithm generates two distinct roads: one comprising the first two ways and another featuring the fourth way. These roads are labeled as Motorring 4_0 and Motorring 4_1, respectively.

5.6 Backend (Valhalla API) (Florian COMTE - s231816)

Now, let's delve into the Valhalla API segment. This API is integrated as a server and runs on the port 8002 of the DTU server.

The API plays a crucial role in our system, serving as a microservice dedicated to map matching. It facilitates the alignment and correlation of geographical points, enhancing the accuracy and precision of location-based functionalities within our application.

As we saw before, when the Flask server triggers the need for mapping matched points, it does the map matching and then return the mapped points. The input format consists of a list of latitude/longitude points, and the output includes information for each point. This information includes details such as whether the point has been accurately map-matched, interpolated longitude/latitude, the corresponding road segment, and the distance along this segment.

If you need more information about the process, you can visit the official documentation about the map matching using Valhalla.

6 Deployment of the Application (Florian COMTE - s231816)

In conclusion of this system specification, we will provide you with all the necessary information for deploying our software on an Ubuntu machine.

This set of instructions guides you through deploying the software by utilizing our Valhalla server and PostgreSQL database. If you prefer to set up your own Valhalla and PostgreSQL, begin by following these tutorials:

1. Valhalla Installation: `https://gis-ops.com/valhalla-part-1-how-to-install-on-ubuntu/`
2. PostgreSQL Installation: `https://www.ionos.com/digitalguide/server/configuration/install-postgresql-on-ubuntu-2004/`

Afterward, proceed to create a database in your PostgreSQL instance using the provided script:

```
BEGIN;

CREATE TABLE IF NOT EXISTS public.coverage
(
    id uuid NOT NULL DEFAULT uuid_generate_v4(),
    way_id uuid,
    lat double precision,
    lon double precision,
    distance_way double precision,
    distance_road double precision,
    CONSTRAINT coverage_pkey PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public.coverage_values
(
    id uuid NOT NULL DEFAULT uuid_generate_v4(),
    coverage_id uuid,
    date date,
    type character varying(255) COLLATE pg_catalog."default",
    value double precision,
    provider character varying(255) COLLATE pg_catalog."default",
    CONSTRAINT coverage_values_pkey PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public.images
(
    id uuid NOT NULL,
    coverage_id uuid,
    date date,
    provider character varying COLLATE pg_catalog."default",
    length double precision,
    CONSTRAINT images_pkey PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public.roads
(
    id uuid NOT NULL DEFAULT uuid_generate_v4(),
    name character varying COLLATE pg_catalog."default",
    ways_number integer,
    length double precision,
    CONSTRAINT roads_pkey PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public.ways
(
    id uuid NOT NULL DEFAULT uuid_generate_v4(),
    osm_id bigint,
    node_start bigint,
    node_end bigint,
    length double precision,
    road_index bigint,
    way_name character varying(255) COLLATE pg_catalog."default",
    road_name character varying(255) COLLATE pg_catalog."default",
```

```
54      distance_road double precision,
55      CONSTRAINT ways_pkey PRIMARY KEY (id)
56  );
57  END;
```

Now that you have your own Valhalla and PostgreSQL (or use ours), you can continue the tutorial.

6.1 SSH Connection

(replace `<USER>` with your user name and use your URL if you have your own machine)

```
1  $ ssh <USER>@se2-c.compute.dtu.dk
```

6.2 Install Global Libraries

```
1  user@se2-c.compute.dtu.dk:¬$ sudo apt update
2  user@se2-c.compute.dtu.dk:¬$ sudo apt install nodejs
3  user@se2-c.compute.dtu.dk:¬$ sudo apt install npm
4
5  user@se2-c.compute.dtu.dk:¬$ sudo npm install -g serve pm2
6  user@se2-c.compute.dtu.dk:¬$ sudo apt-get install python3.6
```

6.3 Clone the repository

```
1  user@se2-c.compute.dtu.dk:¬$ git clone <GITLAB LINK>
2  user@se2-c.compute.dtu.dk:¬$ cd software-engineering-group-c
```

6.4 Setup the React Client

```
1  user@se2-c.compute.dtu.dk:¬/software-engineering-group-c$ cd client
2  user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/client$ nano .env
```

And then put this in the .env:

```
1  REACT_APP_BACKEND_URL_DEV = ""
2  REACT_APP_BACKEND_URL_PROD = ""
```

If you use your own machine, you should put your URL instead of ours.

6.5 Start the React Client

```
1  user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/client$ npm install
2  user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/client$ npm run build
3  user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/client$ pm2 serve ...
       build 3000 --spa --name react
```

6.6 Setup the NestJS Server

```
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c$ cd server-nest
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/server-nest$ nano .env
```

And then put this in the .env:

```
DB_HOST = "130.225.69.215"
DB_PORT = 5432
DB_NAME = "se2-c-database"
DB_USER = "postgres"
DB_PASS = "user"
```

If you are using your own database, you should put your credentials here.

6.7 Add Images in NestJS Server

Download data images (images of the current data we have in the database) using this link and put the `/data` folder at the root of `/server-nest` directory.

6.8 Start the NestJS Server

```
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/server-nest$ npm install
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/server-nest$ npm run ...
    build
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/server-nest$ pm2 ...
    start dist/main.js --name nestjs
```

6.9 Setup the Flask Server

First, we will install dependencies we need:

```
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c$ cd flask-server
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/flask-server$ sudo ...
    pip install python-dotenv Pillow gunicorn flask flask_cors pandas numpy ...
    psycopg2
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/flask-server$ sudo ...
    apt-get install python3-psycopg2
```

Now we need to add the `.env`:

```
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c$ cd flask-server
user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/flask-server$ nano .env
```

And then put this in the `.env`:

```
IMPORT_PASSWORD = ""
DB_HOST = ""
DB_PORT = ""
DB_NAME = ""
DB_USER = ""
DB_PASS = ""
```

```
7
8 VALHALA_URL = ""
```

If you are using your own database, you should put your credentials here. If you are using your own Valhalla server, you should put your URL here followed by `/trace_attributes`.

6.10 Start the Flask Server

```
1 user@se2-c.compute.dtu.dk:¬/software-engineering-group-c/flask-server$ pm2 ...
      start "gunicorn -b localhost:3003 main:app" --name flask
```

Then you should be able to use our app.

7 Handbook (Léonard AMSLER - s231715

Our application's Handbook is designed to provide users with a comprehensive guide to effectively use the application. It covers the application's features, functionalities, and provides users with best practices and troubleshooting guidance. This handbook is intended for users, including road maintainers, data collectors, and any personnel interacting with the application.

Here is a comprehensive flow diagram about our application:

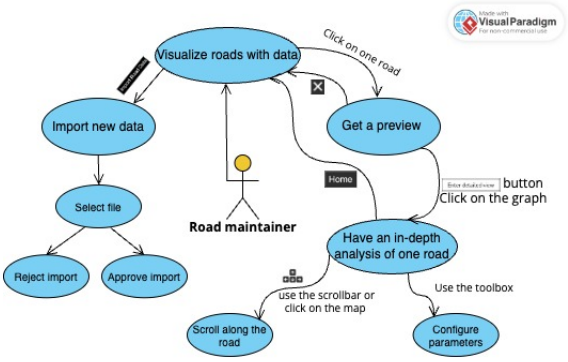

Figure 19: Flow diagram of the application

We will now describe every functionality of the application and how to use it correctly.

7.1 Getting started

To get to the web application, enter the following link into your web browser: http://se2-c.compute.dtu.dk:3000/ . You will be redirected to the main page of the application.

7.2 Main Page

Welcome to the main page of our innovative road monitoring application! This central hub provides users with a comprehensive overview of road conditions, facilitating efficient navigation and exploration of key features. Whether you are a road maintainer, data collector, or any personnel interacting with the application, the main page serves as your gateway to a wealth of information and functionality.

We will then describe a multitude of functionalities from that page.

7.2.1 Selecting the Date Range

To choose a specific date range, utilize the slider at the page's bottom. It features two draggable points, one on the right and one on the left indicating the start and end dates. Adjust them by dragging with the mouse. Upon release, the application displays only the data collected during that period.

Figure 20: Screenshot of the slider

7.2.2 Importing a New Road Segment

To import a new road segment, click on the "Import Road Data" button at the top left of the main page. A new menu appears in the page center. Click the "Browse" button to open a file explorer, choose the desired zip file, and enter the correct password to enable the send button. Clicking "Send" starts the processing. Once finished, a popup will appear displaying details about the road, such as the number of matched points, unmatched points, and affected ways. Confirm or cancel the upload based on the displayed data. We recommend that at least 80% of the points should be matched to accept the import. If confirmed, the importation will take up to several minutes to be effective in the application.

7.2.3 Previewing a Road Segment

Obtain a preview of a road segment by clicking on its blue line. Once clicked, the map re-centers around the selected road segment and a graph section is shown with all the indicators for that road. When hovering the graph, the cursor will be shown at the corresponding position on the map. You can return to the main page by clicking on the cross in the top left corner of the preview.

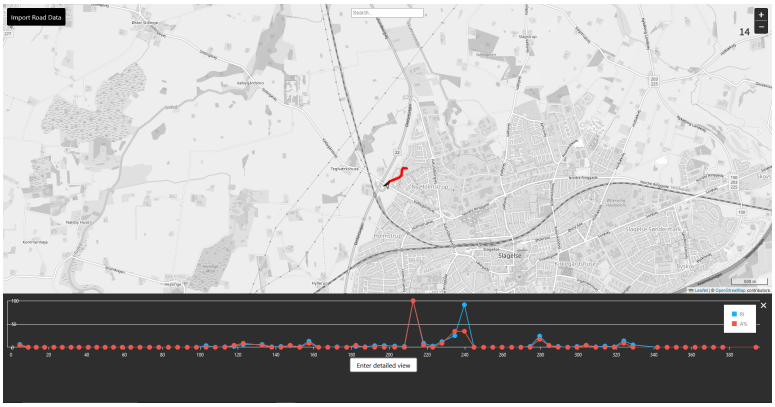

Figure 21: Screenshot of a preview

7.2.4 Get to a page with more details about one road

To access detailed data, first open the preview view by clicking on a specific road segment (see preceding section) and then either click on a point on the graph or on the "Enter Detailed View" button located under the graph.

Clicking on a specific point of the graph will make you go directly to that point on the other page.

7.3 Data Page

Welcome to the data page, where you can dive deeper into the intricacies of road segments and explore detailed data sets. This section is tailored to provide users with an in-depth understanding of road conditions, allowing for comprehensive analysis and informed decision-making.

7.3.1 Navigating the Data Page

As you transition from the main page to the data page, you'll unlock additional layers of information. Here, we guide you through the various functionalities that empower you to explore and analyze road data with precision.

Scrolling Through Data

1. **Slider Navigation:** Utilize the slider at the top of the page by dragging it with the mouse or clicking along the slider to scroll through the data horizontally.
2. **Arrow Keys:** Use the right or left arrow keys to make slight adjustments to the data display.
3. **Map Interaction:** Click on a road segment on the map in the top right corner to jump to a specific point in the data.

Graph Exploration

1. **Graph Modes:** On the right side of the screen, find the "Graph Mode" drop-down menu. Choose "Multiple" to combine all indicators into a single graph, with values scaled between 0 and 100.
2. **Indicator Selection:** Customize the graph by selecting which indicators to display. Use the check-boxes in the menu to toggle individual indicators on or off.

Returning to the Main Page

- Click on the "Home" button at the top left of the data page to return to the main page.

7.3.2 Scenarios and Best Practices

As you explore the data page, keep in mind the following scenarios and best practices to maximize your experience:

- **Efficient Data Scrolling:** Choose the navigation method that suits your preference, whether it's using the slider, arrow keys, or map interaction.
- **Graph Customization:** Tailor the graph display to your specific needs by adjusting graph modes and selecting relevant indicators.
- **Smooth Transition:** Seamlessly switch between the main page and data page to explore road segments comprehensively.

The data page is your gateway to detailed insights and comprehensive analysis. Harness its capabilities to make informed decisions and enhance your road management strategies.

8 Appendix

8.1 Glossary

Back-end - The server-side component of the application responsible for handling requests, processing data, and interacting with databases.

Front-end - The client-side component of the application that provides the user interface and interacts with the back-end.

API - Application Programming Interface: A set of rules and protocols that allow different software applications to communicate with each other.

Endpoint - A specific URL or route in a web API that can be used to access particular functions or data.

Machine Learning - A subset of artificial intelligence that focuses on algorithms and statistical models to enable computers to learn and make predictions or decisions based on data.

JavaScript - A programming language commonly used for web development, allowing developers to create dynamic and interactive elements on web pages.

React - A JavaScript library for building user interfaces, used to create the graphical user interface (GUI) for the web application.

TypeScript - A typed programming language that enhances code quality and type safety.

React Leaflet - A library for integrating interactive maps into web applications built using the React framework, allowing developers to display and work with map data.

DevExtreme Charts - A library for creating interactive and customizable charts in web applications.

OpenStreetMap (OSM) - Open Street Map: A geographic database that provides information about roads, including GPS coordinates, types of roads, and road names.

Web Application - A software application that is accessed and used via a web browser, typically providing interactive user interfaces and functionality over the internet.

GPS - Global Positioning System: A satellite-based navigation system that provides location and time information in all weather conditions.

LiRA Map - The existing project that serves as the foundational source for the current project. It includes road layouts, coordinates, and condition data of different types.

Dynatest Database - A database that provides data on road conditions, including potholes, cracks, surface quality, and images.

Georeferenced - Data that is associated with specific geographic coordinates, allowing precise location-based information.

GeoJSON - A format for encoding geographic data structures in JSON (JavaScript Object Notation).

Polyline - A connected sequence of line segments formed by joining coordinates on a map.

FeatureCollection - A collection of GeoJSON features, typically used to represent geographical features on a map. **Road Condition** - The state or quality of a road, often assessed using various indicators and measurements, such as KPI (Key Performance Indicator) and IRI (International Roughness Index), to evaluate safety and comfort.

Indicator - A parameter or variable used to measure specific aspects of road conditions, such as KPI (Key Performance Indicator) and IRI (International Roughness Index).

KPI - Key Performance Indicator: A measurable value that indicates the performance or effectiveness of a specific process or aspect.

IRI - International Roughness Index: A standardized measurement of road surface roughness used to assess road conditions.

GUI - Graphical User Interface: The visual elements and controls that users interact with in a software application.

Road Network - A system of interconnected roads that form a transportation infrastructure.

Road Segment - A specific section of a road, often between two distinct points.

9 Contributions

9.1 Roles and Leadership

In the context of this project, Florian COMTE assumed the role of Leader, leveraging his experience with prior experience in React. Meanwhile, Léonard AMSLER took on the Deputy role, capitalizing on his expertise in GIT and organizational skills within a team.

9.2 Deployment of the software on the VM

Dana Toma took charge of deploying the software on the virtual machine (VM) of the university.