Bachelor Thesis

Smart Traffic Management System

TI3806 - Bachelor End Project

Team Members
Frank Bredius
Titus Naber
Bailey Tjiong
Leroy Velzel

Committee
K.F. Chan
Dr. A. Katsifodimos
Ir. O.W. Visser

TU Delft

Smart Traffic Management System

Scenwise B.V.

Bachelor Thesis

in partial fulfillment of the requirements for the degree of
Bachelor of Science
in Computer Science

by

Frank Bredius
Titus Naber
Bailey Tjiong
Leroy Velzel

at the Delft University of Technology,
to be defended publicly on 11 February 2020.

SCENWISE

Your dreams of future mobility is our mission today.

TU Delft Delft University of Technology

Keywords: Traffic management system, response plans, congestion, ScenWise

Preface

This thesis elaborates on the process as well as the end-product created during the Bachelor End Project (BEP). The BEP is a 10 week compulsory project to complete the Computer Science and Engineering program of the Delft University of Technology. It was conducted between 11 November 2019 and 11 February 2020.

During the first two weeks the group focused on researching the problem, the possible solutions and traffic management in general. The subsequent 8 weeks were designated to the development of the product. The product was developed for ScenWise, a company that focuses on innovation within traffic management.

In addition to evaluating the product, this thesis gives recommendations for a following group working on extending the product.

Finally, the group would like to thank the client ScenWise, specifically K.F. Chan for providing the project as well as teaching us about the domain knowledge of the client. Furthermore, the group wants to thank Dr. A. Katsifodimos for the coaching throughout the project as well as Ir. O.W. Visser for giving the opportunity to do the BEP in the second quarter of the academic year.

Frank Bredius
Titus Naber
Bailey Tjiong
Leroy Velzel
Delft, February 2020

Summary

This thesis evaluates the created web-application for traffic engineers. In this application, a traffic engineer can prepare the response plans online. Traffic managers can then manage the traffic using the measures prescribed in the response plans. The aim of this thesis is to substantiate the design choices, to elaborate on the system architecture and explain the features of the application. Research was conducted in the field of traffic management, languages, libraries and frameworks used for web-applications, and into the variety of database types available. Consecutively, this resulted in a working application as well as an educational project. Therefore, the conclusion is that both the client and the group members are satisfied with the course of the project. Recommendations are given to extend the application further outside the scope of the project.

Contents

1 INTRODUCTION

Traffic managers manage the traffic flow by controlling various instruments, e.g. the lane control signaling. To make this process as seamless as possible, response plans are created. In these plans, the actions of the traffic operators are predetermined depending on the situation. However, currently some of these actions (e.g. lane closure or diversion routes) are taken manually. Furthermore, the response plans are printed versions of a flow diagram, making it hard to adjust as well as to test the plans.

Scenwise wants to innovate on this aspect of traffic management, by creating a Smart Traffic Management System. This system is a web-application for the online creation and testing of response plans, in contrast to the currently printed versions of response plans. The system is created for the designer of the response plans and is not supposed to control the instruments. This thesis describes the process as well as the end result of the creation of this system.

To design such a system, research has been done. A report of this research can be found in Appendix B. Analysis of the problem is further elaborated in chapter 2. Subsequently, the architecture of the system is explained in chapter 3. The end-product is described in chapter 4. Code quality and the process of the project are discussed in chapter 5 and chapter 6 respectively. Finally, the project will be evaluated and concluded in chapter 7 and chapter 8, followed by recommendations in chapter 9.

2
PROBLEM ANALYSIS

In this chapter the problem will be introduced and analysed, and the requirements of the product will listed according to the MoSCoW method.

2.1. PROBLEM DEFINITION

Currently, traffic operators follow specific response plans that describe which actions should be taken to manage different traffic situations (e.g. congestion, accidents, events). An example of a response plan can be found in Appendix D. Manually following instructions in response plans is tedious and time-consuming, whilst time is a limited resource in traffic management. To improve the workflow, an application that automates the process needs to be developed. This application should help traffic operators to make fast and safe decisions using real-time traffic data to trigger response plans.

2.2. CURRENT PROTOTYPE

The response plans are currently booklets with a combination of text, maps, legens of roadside instruments and flowcharts. These booklets provide the guideline to the traffic manager. In order to automate the process, Scenwise has remodelled the response plans into decision trees. Figure 2.1 shows an example of a simple decision tree.

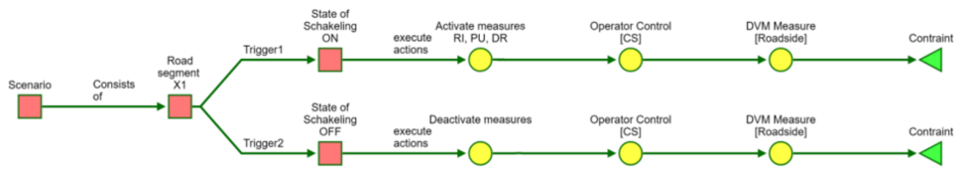


Figure 2.1: The response plan modeled as a decision tree.

In the current prototype the user can construct a decision tree. This tree is stored, in a non-relational database, as one JSON string. This JSON string can become very long and cluttered with more complex decision trees. Furthermore, this makes retrieving and updating a specific node of the decision tree inefficient. Moreover, automatically triggering states based on live data is not possible in the current prototype.

2.3. SCOPE OF THE PROJECT

This project aims to create a web-application for the designers of response plans in order to manage traffic. The designers should be able to construct and test a response plan using historical or real-time traffic data.

2.4. ALTERNATIVES

Before the development phase, alternative systems with similar functionality were found.

2.4.1. ARCADIS

The first system found is Arcadis. A visual representation of the system can be found on YouTube [1]. However, the platform is out-dated. The development stopped because the provided functionality did not met the requirements of the users. This program is one of the few programs which was able to create, read, update and delete response plans.

2.4.2. TRAFFICLINK

This system shows a detailed version of every traffic situation. It is a Dutch company which strives to optimize the throughput of traffic. Examples of their product can be found on their website[2]. TrafficLink was only used by Amsterdam. The system was too expensive and the effort for configuration management was too high.

2.4.3. TECHNOLUTION

MobiMaestro[3] is currently the most advanced program in the field of traffic control. It is created by Technolution, one of the traffic management systems that own different type of road instruments. The product is innovative in the sense that it takes all possible sources of data into account (e.g. data from mobile phones). This solution is closely related to the team's solution, the difference being that the decision tree and the live simulation are not present in Technolution's system.

2.5. INNOVATIVE ASPECTS

As mentioned in section 2.4, there are a few existing alternatives of automated systems for regulating the traffic flow. Some of these alternatives contain response plans, however, none represent those response plans as decision trees. Besides the ability to create and edit response plans as decision trees, the group's application has interaction with the map for (alternative) routes and instruments. Furthermore, the application makes it possible to run the response plan on live data, where it highlights the active parts of the

[1]https://www.youtube.com/watch?v=BKRU-kTYICI
[2]https://www.trafficlink.nl/trafficlink-online-verkeersmanagementoplossing-in-de-cloud/
[3]https://technolution.eu/nl/mobiliteit/mobimaestro/

decision tree based on the live data. Therefore, the designed application is unique in the field.

2.6. REQUIREMENT ANALYSIS

The requirements of the product were composed and analysed during multiple meetings with the client. The MoSCoW method was used to prioritise these features, after finalising the requirements the client agreed upon the functionalities defined below.

2.6.1. MUST HAVES

1. Search and view scenarios.
    - Search scenarios by title.
    - Update the map when clicking on a branch of the scenario, add a small view of the selected scenario.
    - When clicking on a part of the scenario tree the map indicates:
        - The part of the road.
        - Which instruments are used in the plan. The instruments can be Dynamisch Route-informatiepaneel (DRIP), Verkeersregelinstallatie (VRI), Toeritdoseringsinstallatie (TDI), Motorway Traffic Management (MTM).
        - Which alternative route is proposed to the road-users by the instrument.
        - Which arm of the road-crossing gets higher priority and thus longer green-time.
        - How the road users should be informed in an area (polygon) or some routes. The instruments can be DRIP, radio (broadcast) or in-car system (personalise).
2. Display instruments.
    - Update the map when clicking on an instrument.
    - Create-Read-Update-Delete instruments.
    - Loading a list of instruments (list of VRI's, list of DRIP's,etc.).
    - Switch between DRIPs, TDIs, VRIs, MSIs, etc.
3. Create/edit scenarios.
    - Move components of a scenario (with location) in fixed order:
        - Conditions
        - Actions
        - Constraint
    - When clicking on a part of the decision tree, the map indicates:
        - The part of the road or the event location (e.g. Arena stadium).
        - Which DRIPs, VRIs, TDIs and MSIs are activated.
        - Which alternative road is used.
    - Simulate scenarios

2.6.2. SHOULD HAVES

1. In the search bar:
    - Search scenarios by label.
    - Move scenarios between folders.
    - Create-Read-Update-Delete folders.
    - Sort scenarios.
2. In the instruments bar:
    - Sort instruments.
    - GEO-filter (Only show instruments within the current view of the map).
    - Search instruments by label.
3. In the scenario designer view:
    - Add labels to scenarios; such as the city, road name, event name.
4. Simulate scenarios with real time data.
5. Send messages with an alternative route to users.

2.6.3. COULD HAVES

1. Create and extend the export function for UML.
2. Save a certain simulated timeframe.
3. Add incident detection to application.

2.6.4. WONT HAVES

1. Build the navigation app.
2. Control the roadside equipment.

3 System Design

In this chapter the design of the application will be introduced, explained and motivated. The chapter is divided into two sections, the back-end and the front-end. This is a natural division within the application. Both sections have the same structure. First, the design choices will be listed and motivated. Then, the architecture will be explained and motivated. And finally, some implementation challenges are evaluated.

3.1. Front-end

3.1.1. Design Choices

Frameworks

A large portion of the application is realised by the front-end, thus a main framework had to be chosen to assist in this development. A selection of three widely adopted technologies was made, React.js[1], Vue.js[2] or Angular[3]. These frameworks are the top three entries in Figure B.2. The main considerations were as follows:

- If the framework fits the scale of the project
- Learning curve of framework
- Integration with back-end (GraphQL)
- Integration with important packages for this project
- Previous experience of team members with the framework
- Correspondence of framework with previous projects in the codebase

[1] https://reactjs.org/
[2] https://vuejs.org/
[3] https://angular.io/

Based on these main considerations, a choice for React was made. React and Vue are smaller frameworks than Angular and allow for more flexibility in making own choices. Due to the unique nature of some features in this project it was important to maintain this flexibility. Both React and Vue are said to be easier to learn than Angular. All three options have a good integration with GraphQL. However, React in combination with Apollo is the most common. React does not allow packages with a dependency on jQuery which forces the developer to use the packages specially adapted for React. These packages sometimes do not have the same set of functionalities as the original full versions. Furthermore, two group members already had previous (positive) experience with React and one with Angular. Finally, the original codebase is also written in React, this meant that using React would allow for easy reuse of previously written components.

A second framework is introduced in the form of Apollo[4]. This framework helps with connecting the React classes to the server via the GraphQL architecture. An alternative to Apollo is urql[5], which is a less extensive and heavy version of Apollo. However, it is not widely used and still upcoming. An advantage of using Apollo is that it can also be used as a global state manager. Therefore, Redux, a commonly used global state manager, does not have to be used in the application.

LANGUAGES

Instead of regular JavaScript, TypeScript[6] is used. TypeScript allows for static type-checking. The choice for this language is made to keep the code more structured, less bug sensitive and easier to read.

In the project, the commonly used programming language CSS is replaced by SASS[7]. The language allows for, among other advantages, assigning variables and nested styling. It is easier to write and to read than regualar CSS. Furthermore, the choice was made not to use in-line styling, an option offered by React. Separating the styling from the React classes results in cleaner code and a better overview.

LIBRARIES

React Bootstrap[8] is added to the project in order to use components with standard styling, such as a pop-up or search bar. It is not possible to use regular Bootstrap in combination with React due to Bootstrap's dependency on jQuery.

MapBoxGL[9] is used to perform actions and to draw on the map. MapBox provides a map, styled to preference, and a library to modify and draw on this map. Since this set of features is needed in our project, MapBox was a good fit for the project. A more elaborate consideration of the different Map libraries can be found in B.2.

For constructing the tree, D3.js[10] is added to the project. This is a graphing library with an enormous scope. This large library, instead of a more comprehensible and smaller one, was chosen due to the unique nature of the problem.

[4] https://www.apollographql.com/
[5] https://formidable.com/open-source/urql/
[6] https://www.typescriptlang.org/
[7] https://sass-lang.com/
[8] https://react-bootstrap.github.io/
[9] https://docs.mapbox.com/mapbox-gl-js/api/
[10] https://d3js.org/

Node Package Manager is used to download extra packages. Babel is used to compile next-gen JavaScript to browser-compatible JavaScript. The full setup is bundled to a static package by webpack[11].

3.1.2. ARCHITECTURE

The application is divided into three different views. The Home view, the Scenario Designer and the Scenario Simulator. These views all have a similar structure when it comes to styling and content. In this section, first, the global structure of the application will be discussed and visualized. Then, a look is taken into the common structure which is shared between the different views. Finally, the connection with the server is discussed.

The UMLs are not exact representations of the application structure, but rather only show the parts relevant to the chosen architecture. Otherwise, the UMLs would become too complex and unreadable.

GLOBAL APPLICATION ARCHITECTURE

A React app is build up by components. However, instead of a more common inheritance model, it has a strong composition model. This means that a class is filled up by the components further down. New XML 'tags' can be created in the form of React classes. These classes can contain their own logic, styling and content. When a view is constructed, such a class can be added similar to other XML tags. For instance, similar to *<img src="/image">* a class can be added like *<Home exampleProp={prop}">*. Thus, when the 'App' component contains the 'Home' class, both XML written in App and Home will be displayed.

In Figure 3.1 can be seen that the application builds up from the 'App' component, as is common for react applications. This component will be the root for all classes. Then, three different views are created, each of them with different features and purposes. The 'App' component will always be rendered and depending on the URL a corresponding view component will be rendered into this 'App' component. Due to every React app being a single-page application, all changes to the URL will be handled inside the application without actually changing its reference location.

The three views share a common structure. Figure 3.2 visualizes this shared structure between the components. Every view is divided into four main parts: a *LeftPane*, a *RightPane*, *LargeWorkspace* and *SmallWorkspace*. In the next section, this design will be discussed in more detail at a lower level. For now, a closer look will be taken into the implementation of these shared components.

Comparable to interfaces in a standard object-oriented environment, high-order components (HOC) are constructed. HOCs extend the normal classes with the set of functionalities shared by the views. These classes typically start with 'as'-Component and are indicated by the '« ClassName »' symbols. The choice for HOCs is made because it prevents a lot of code duplication. However, a drawback of using HOCs is that it requires a complex class structure which makes the application more complex.

At the top of Figure 3.1, a connection to the server is displayed. The Apollo framework allows for every component to query or mutate the server. Additionally, the framework allows for constructing a local store. Every class in React has its own state and through

[11]https://webpack.js.org/

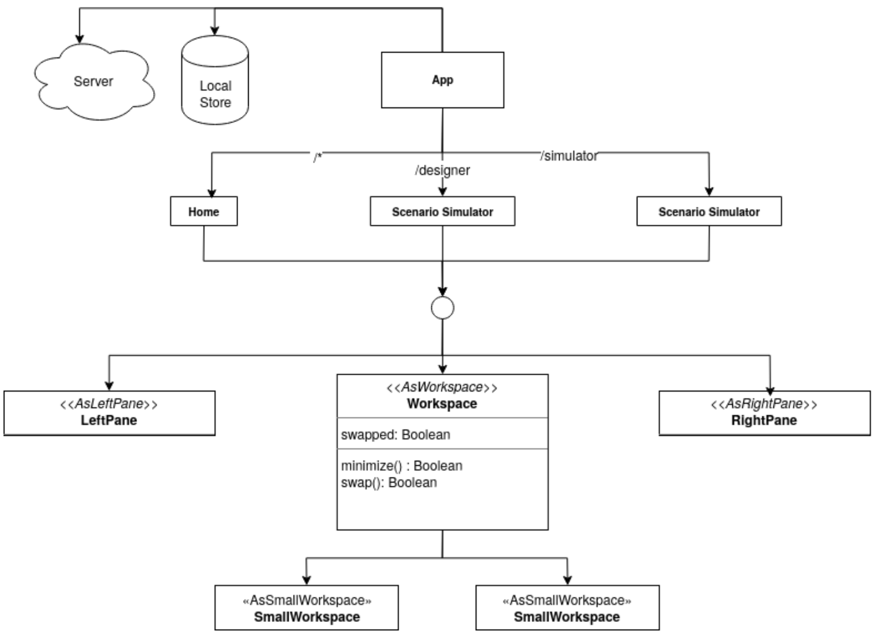
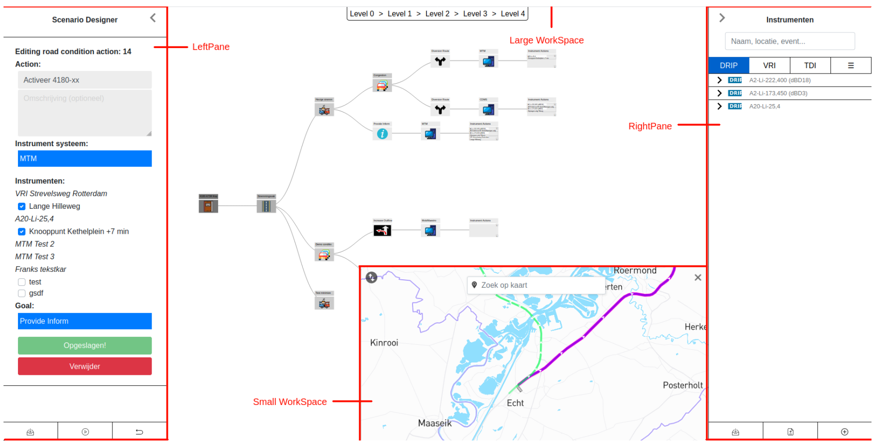


Figure 3.1: Global structure of the front-end



Scenario Designer

Editing road condition action: 14

Action:

Activeer 4180-xx

Omschrijving (optioneel)

Instrument systeem:

MTM

Instrumenten:

VRI Strevelsweg Rotterdam

- [✓] Lange Hilleweg

A20-Li-25,4

- [✓] Knooppunt Kethelplein +7 min

MTM Test 2

MTM Test 3

Franks tekstkar

- [ ] test
- [ ] gsdf

Goal:

Provide Inform

Opgeslagen!

Verwijder

Figure 3.2: Global architecture of the front-end

the use of props these classes can share parts of their state with other classes. However, when dealing with larger structures it can quickly become too complex to share some states through props. Therefore, a local store is introduced. This store can be seen as a global state for all components. Using Apollo, this local store can be accessed and mutated in the same way as the remote server.

SHARED COMPONENTS OF THE VIEW ARCHITECTURE

In this section the structure shared by the three components will be discussed. In contrast to the previous segment, this section will go over the inner workings of some of the components. Starting with the view component, this component has relatively little logic or features. It is mainly used for setting up the structure and providing its children with their corresponding data. This component keeps track of and controls whether the *LeftPane* or *RightPane* are open or closed.

The *asLeftPane* and *asRightPane* interfaces construct the styling of the panes based on whether they are active or not. Because the contents of these panes vary strongly per view, these interface do not contain any logic besides the logic needed to add the right styling to the component.

The *asWorkspace* interface is a central part of the whole application. The *Workspace* class hardly differs between views and therefore a lot of the code can be included in its interface. The interface has three main functionalities.

Firstly, to provide data to its children. The *Tree* and *Map* component both require a substantial amount of data to be passed down as props. This data needs to be fetched from both the server and the store. A large query is defined to fetch all this data in one call from both stores. Examples of the data needed are the current focus point for the map (from the local store) and the actual scenario which is displayed in the tree (from the server).

Secondly, processing this data. This data needs to be processed for the tree to be able to display a scenario from the server. For instance, different types of nodes need to be combined into a single ‘children’ array. Due to the complexity of these functions they have been moved to a separate file to keep the class clean and straightforward.

Finally, the children need to be rendered. Based on whether the ‘Tree’ and ‘Map’ are swapped, one is loaded into the large workspace and the other in the small workspace. The small and large workspace do not contain much logic themselves as they mainly change the styling of their child. The small workspace interface adds the possibility to be minimized. However, this logic is mainly contained by its parent.

It is recommended to refactor this file by dividing these three tasks over three different components when it grows any larger in the future. However, at the current scale a refactor is not needed and the class is still clear.

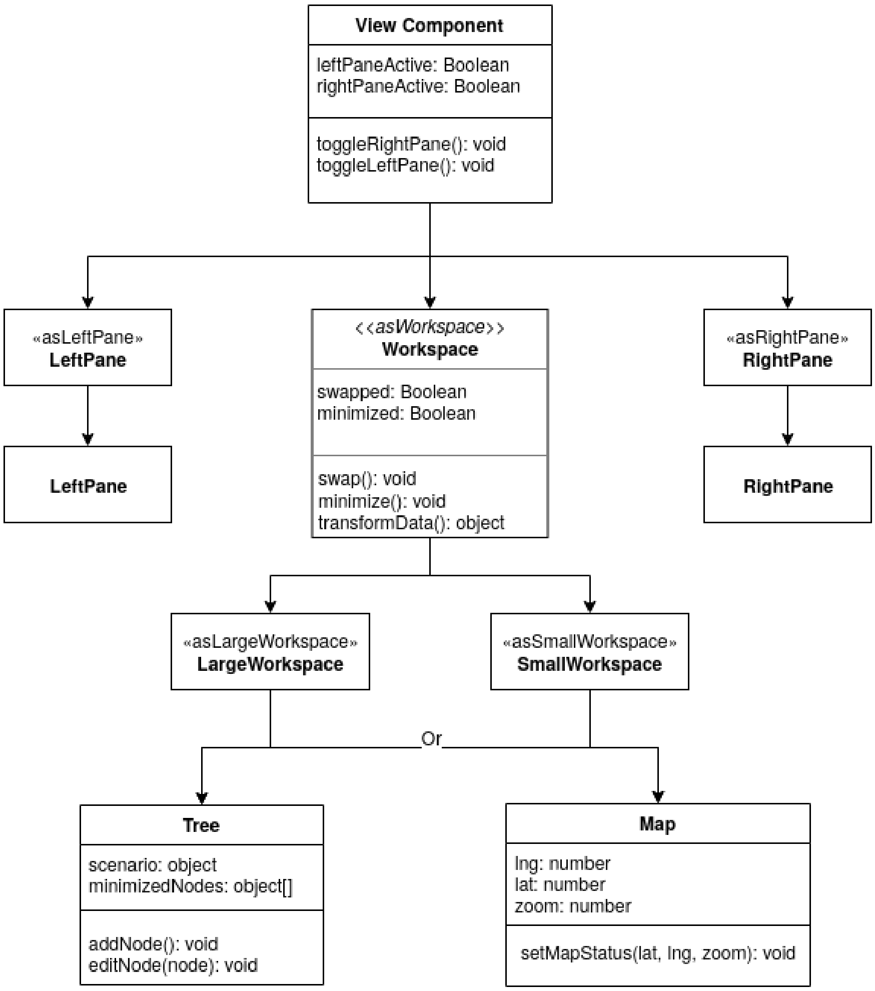
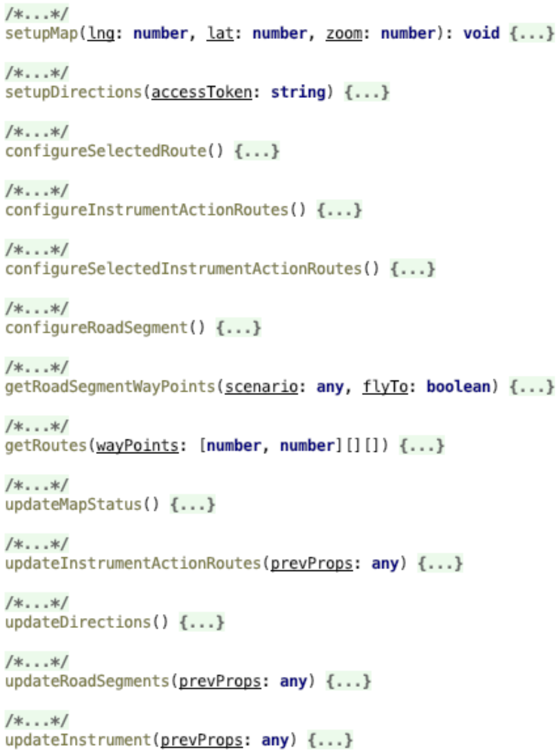


Figure 3.3: High-order architecture of view components

Map Architecture

In React, a commonly used practice is to update the state of the component when changes are made to that component. This will result in a re-render of the component, with the changes visible after the re-render. However, when using MapBox in a component in React, a state update of the Map is undesirable. Since a state update would result in a re-render of the entire Map. Therefore, React's life cycle method *componentDidUpdate()* is used in the Map component.

To illustrate, the Map is divided into three parts, see Figure 3.4. The first part is the setup of the Map. In this part the Map gets instantiated, using the *setup* methods. In the second part, MapBox's layers[12] and sources[13] for each of the visual parts of the Map (e.g. the routes) are defined using the *configure* methods. The *configure* methods make use of the *get* methods. These methods call the MapBox API for detailed routes between two coordinates. The third part, using the *update* methods, is used to update the MapBox sources. These methods are called by the *componentDidUpdate()* method. This causes the Map to change its visuals, without re-rendering the map. Since having each methods' implementation in one file would result in a huge file, each part is implemented in a separate file. E.g. the *setup* methods use methods defined in the *setupMap.tsx* file.



```
/*...*/
setupMap(lng: number, lat: number, zoom: number): void {...}

/*...*/
setupDirections(accessToken: string) {...}

/*...*/
configureSelectedRoute() {...}

/*...*/
configureInstrumentActionRoutes() {...}

/*...*/
configureSelectedInstrumentActionRoutes() {...}

/*...*/
configureRoadSegment() {...}

/*...*/
getRoadSegmentWayPoints(scenario: any, flyTo: boolean) {...}

/*...*/
getRoutes(wayPoints: [number, number][][]) {...}

/*...*/
updateMapStatus() {...}

/*...*/
updateInstrumentActionRoutes(prevProps: any) {...}

/*...*/
updateDirections() {...}

/*...*/
updateRoadSegments(prevProps: any) {...}

/*...*/
updateInstrument(prevProps: any) {...}
```

Figure 3.4: Method declarations in Map

Tree Architecture

The functionalities in this component can be divided into two groups. Those which use React and those which use the D3.js library to implement their logic.

The first category contains features such as the ability to sort per level and to minimize nodes by passing down props.

The second, and larger, category, contains the visualization of a scenario, adding nodes, editing nodes and minimizing nodes locally. The tree is build using the D3.js

[12]https://docs.mapbox.com/mapbox-gl-js/style-spec/layers/
[13]https://docs.mapbox.com/mapbox-gl-js/style-spec/sources/

library. As is normal with D3, the tree is displayed using an SVG format. Because the data can change during the period in which the tree component is active, the tree has to be redrawn every time the data gets updated. Therefore, the ability to use animations when changing the data was sacrificed. In other systems, where the data is static, this SVG image can change by using animations. Upon creating the tree, buttons are created and certain functions are bound to these buttons. For example, when clicking a node in the tree, the *id* and *typename* will be send to the local store such that another component can initiate the edit process.

Figure 3.5 displays an example of a tree with the node types indicated. Every node can be uniquely identified by a combination of its *id* and *typename*. In the following paragraphs the properties relevant to the tree of each of the types will be discussed. For more elaborate information about the different types see subsection 3.2.2.

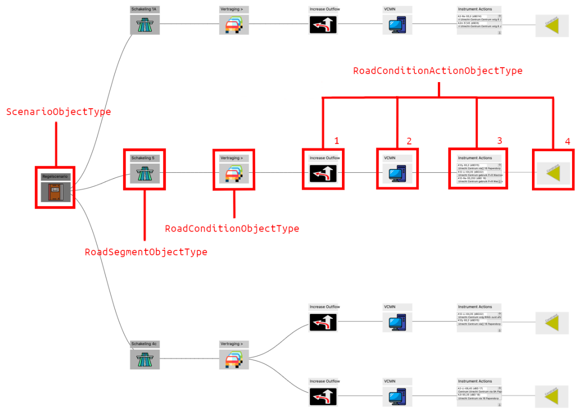


Figure 3.5: An example of the decision tree with the node types indicated

Starting at the root, the *ScenarioObjectType* node is at the base of the decision tree. As a scenario can span over multiple roads, this node can have multiple *RoadSegmentObjectType* nodes as children.

The *RoadSegmentObjectType* node will display what kind of road type it is (tunnel, highway, bridge, etc.). When selected, the map will display this segment. This node can have multiple *RoadConditionObjectType* nodes as children. These conditions will only apply for this given road segment.

A *RoadConditionObjectType* node will apply a certain condition upon its parent road segment. There are various types of conditions (congestion, event, accident, etc.), these can be displayed with an icon within the tree. This node can have multiple new *RoadConditionObjectType* and/or *RoadConditionActionObjectType* nodes as children.

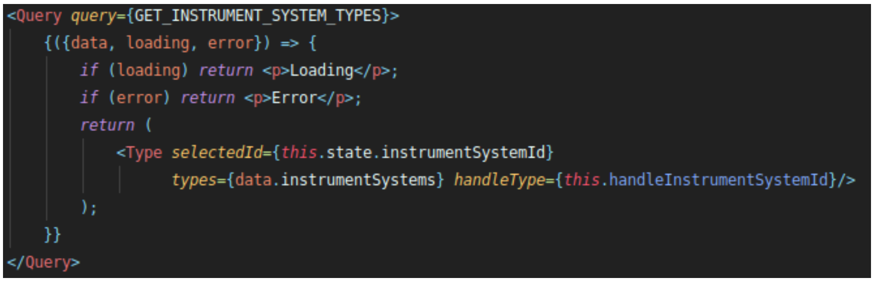
Finally, the *RoadConditionActionObjectType* is a special node as it is displayed as four different nodes. However, because these nodes always appear in the same sequence, these are modelled as a single node. In Figure 3.5 the nodes are numbered. Number 1 is the goal which should be achieved according to the response plan and the current situation on the road. This 'node' can have several types such as *reducing inflow, in-*

creasing outflow or *informing the traffic*. Number 2 shows the central system which has to be used in order to achieve this goal. Number 3 displays the actions which should be performed on the roadside equipment. This will mainly be displaying text on certain screens besides the road or adjusting the speed limit on relevant roads. Number 4 is a final constraint node, this node is for adding an exemption to the rules given by this node. It can indicate technical and traffic related problems (e.g. a malfunctioning road sign). If such a problem occurs, the related actions will be deactivated. This node still has a very broad purpose and has been added late in the development stage in order to handle the inevitable edge cases.

All these nodes have full CRUD functionality. Every node can be separately edited and after each of these edits, the full tree will update. Therefore, it will stay up to date with the server. In the following section this connection with the server is explained. The separate modelling of each node is further elaborated in subsection 3.2.2.

CONNECTION WITH THE SERVER

Using the *graphql-tag* package it is possible to write GraphQL queries in JavaScript. When the correct query is constructed, Apollo has a pattern for fetching the data from the server. In Figure 3.6, an example is displayed of such a pattern. The GraphQL query variable names are always in capitals and stored in separate files. When a component is loaded, the query is called. The same pattern is used for queries to the local store. However, when fetching the local store, the loading parameter can be removed as there is no latency.



```
<Query query={GET_INSTRUMENT_SYSTEM_TYPES}>
    {({data, loading, error}) => {
        if (loading) return <p>Loading</p>;
        if (error) return <p>Error</p>;
        return (
            <Type selectedId={this.state.instrumentSystemId}
                  types={data.instrumentSystems} handleType={this.handleInstrumentSystemId}/>
        );
    }}
</Query>
```

Figure 3.6: Example of Apollo design pattern

IMPLEMENTATION CHALLENGES

React Hooks

Initially, some of the group members had some prior knowledge of React. Recently, React had its update which included React Hooks. However, wanting to have a prototype as quickly as possible, the decision was made to use the 'old fashioned' React way of using classes. Subsequently, Apollo was integrated into the project. Problem being, Apollo assumes the usage of React hooks for a seamless integration. Apollo can also be used using React with classes, but this results in 'ugly' code. The deliberate decision was made to use React classes, taking the less sophisticated code for granted.

Combination of multiple frameworks
Another implementation challenge was the use of multiple frameworks and libraries in one project. E.g. the D3.JS library assumes it to be used in a standard JavaScript project. However, since the Apollo framework was used, D3.JS could not be used to its full extent. For example, the decision tree could have had some fancy animations. This requires the data to be part of the tree. Nevertheless, the data is fetched from the back-end using Apollo queries. Therefore, when the tree is adjusted, the data will be sent to the back-end. Consecutively, the data will be fetched again and displayed in the tree. This results in the tree having to be drawn again. This means that the D3.JS library can not execute an animation between the two states, since it is not a data manipulation within the tree itself.

3.2. BACK-END

3.2.1. DESIGN CHOICES

FRAMEWORK

The Python framework that is used in the back-end is Django. Django follows a Model-View-Template (MVT) architecture, which is a variation on the Model-View-Controller (MVC) architecture [1]. The model part of Django makes it easier to deal with the data in the relational database. It provides an Object-Relational Mapping (ORM) to the underlying database, which means that it maps an object to the database. This prevents the use of complex Structured Query Language (SQL) in the back-end for retrieving and manipulating data, which simplifies the ability to create and update nodes of the decision tree [2].

LANGUAGE

Based on the responsibilities of the back-end and the experience of the group members, the decision for the programming language was made between Python and Java. The reason Python is used in the project is that firstly, it has a less steeper learning curve than Java. Secondly, Python requires less lines of code for the same functionality. This results in increased readability in the software. Thirdly, Python has easy integration with other components that will be used in the application. And lastly, the client wants to extend this project with machine learning in the future. Python provides more and better options for machine learning than Java.

API

In order to communicate within the application, GraphQL is used. GraphQL uses a schema as well as a type system. The types are written in a schema using the Schema Definition Language (SDL). This schema can be seen as a contract between the client and server, which defines how the client can access the data. Unlike REST, GraphQL has only a single endpoint to receive queries, including the concrete data requirements, from the client. The response is a JSON object containing the desired requirements. The advantage of this compared to REST is that there is no over- and underfetching [3], since the client can decide what information it wants from the server. This feature makes it easy for the application to fetch single nodes such as simulations, and scenarios and instruments from the decision tree, instead of fetching the entire decision tree.

Graphene

In order to implement the GraphQL API within the Python back-end, the Graphene library is used. Graphene offers integration with the Django framework. Graphene-Django is built on top of Graphene and simplifies the addition of GraphQL functionality in Django [4].

DATABASE

In the prototype of the application, MongoDB was used as database. Instead of using a non-relational database (e.g. MongoDB), a relational database is used in the current application. A relational database uses tables and rows to store data. Information between the different tables is linked via foreign keys. The main reason for changing to a relational database is that updating or retrieving a scenario within a non-relational database requires the application to send the whole decision tree in a single String. When using a relational database and single tables for single nodes, it is easier to update or retrieve single elements of the decision tree. There are a lot of different relational databases, where MySQL is one of the most used databases. Furthermore, MySQL is robust and provides a multitude of functionalities. Most importantly, Django provides official support for MySQL, which simplifies the integration of the database in the back-end.

SERVER

In the prototype of the application NodeJS was used to create a REST like communication with the Database and the Client. Because one of the request in the future has to do with machine learning we preferred Python over Java and NodeJS to use as our back-end language. The only GraphQL library, with reasonable support, that was available was Graphene which is used in combination with Django. However because GraphQL is a young framework the Subscription feature was not included in the graphene libary at the time of development so we had to use websockets to send the data to the client. Because of the websocket the application was changed from a synchronous server to a asynchronous server.

3.2.2. ARCHITECTURE

The back-end of the application handles modifications and retrievals of the data from the database. Furthermore, the simulations (with NDW data) are also handled in the back-end. Lastly, the back-end provides the ability to import and export a single scenario as a JSON file and multiple instruments simultaneously as CSV files.

DATABASE STRUCTURE

The response plans are represented as decision trees. The individual nodes of the decision tree and the connection between the nodes are stored in the database. A complete EER diagram of the database can be found in Appendix I.

Decision tree

The decision tree is modeled with a scenario as root node. A *scenario* represents a specific response plan. It contains a location based on the connected *road segments*. A *scenario* can have multiple *road segments* as child nodes (Figure 3.7). A *road segment* represents the location of interest and contains a *route* (defined by multiple *route segments*) and a *road segment type* (road segment, bridge, tunnel, motorway, road crossing or connection road).

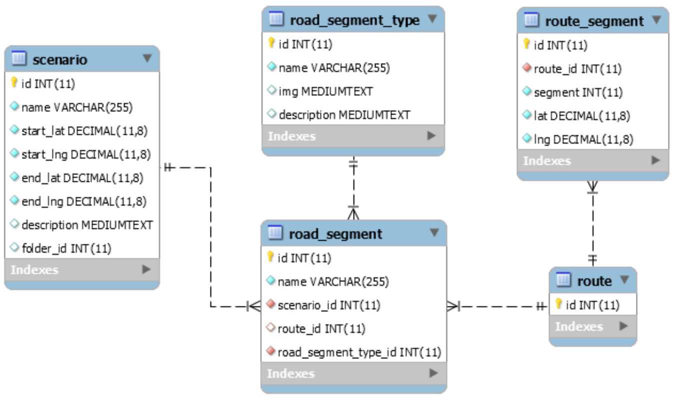


Figure 3.7: The relation between a scenario and its road segments.

Each *road segment* has a *road condition* as child node (Figure 3.8). A *road condition* represents the condition of the trigger. It contains a start- and end-time, a *road condition type* (congestion, accident, event, roadwork or broken car) and a value. For congestion, the value represents the current speed, scaled to a integer between 1 and 10. For other *road condition types* the value is set to 10.

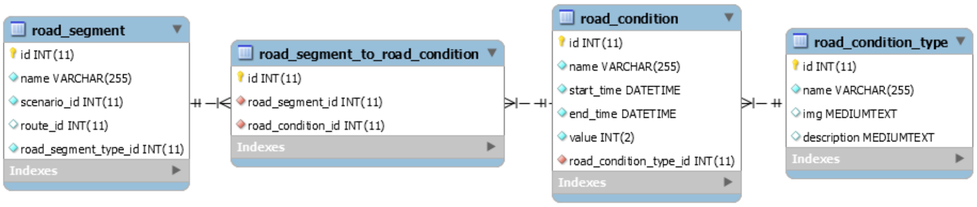


Figure 3.8: The relation between a road segment and its road conditions.

A *road condition* can have another *road condition* as a child node as well, since there can be multiple conditions on a *road segment*. The last *road condition* has a *road condition action* as child node (Figure 3.9). This *road condition action* represents the action to be taken when the condition is met. It contains a *constraint* with a *constraint type* (traffic related or technical), the *instrument system* (MobiMaestro, CDMS or MTM) it is assigned to and a *road condition action goal* (reduce inflow, increase outflow, diversion route or provide inform). The *road condition action goal* represents the actual goal of the action that needs to be taken.

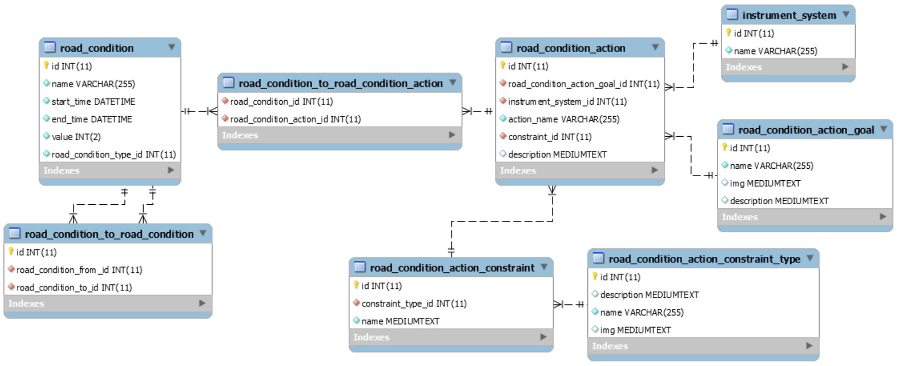


Figure 3.9: The relation between a road condition and its road condition actions.

A *road condition action* has at least one *instrument action* as child node (Figure 3.10). An *instrument action* represents how the action should be executed. It contains an *instrument* it is assigned to and a text that should be displayed on the *instrument*. Additionally, it contains at least one *route*, which serves as an alternative route for the aforementioned *road segment*. An *instrument* represents an existing road instrument. It contains a location, an *instrument system* it belongs to and an *instrument type* (DRIP, VRI, TDI, MTM, Tekstkar, Verkeersregelaar or Overig).

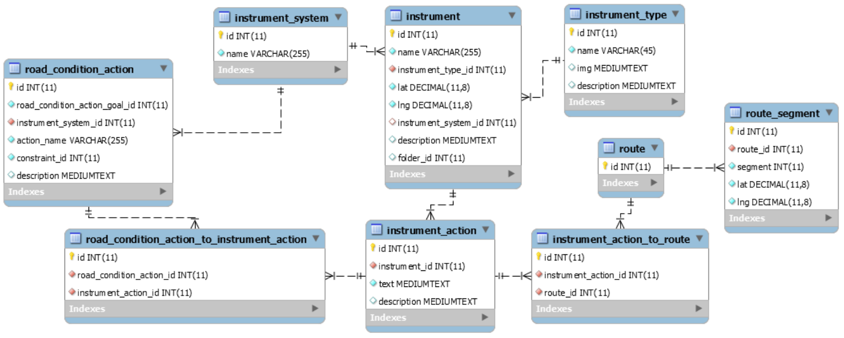


Figure 3.10: The relation between a road condition action and its instrument action(s).

The relation between the decision tree nodes and the corresponding database tables can be found in Figure 3.11.

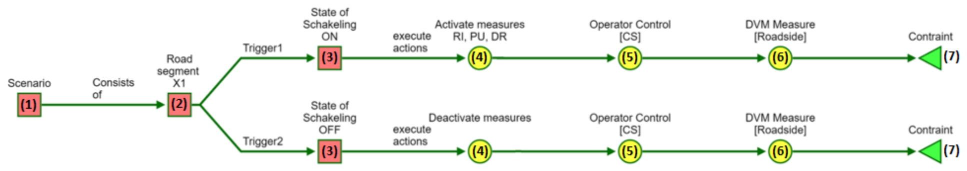


Figure 3.11: (1) scenario. (2) road_segment. (3) road_condition. (4) road_condition_action & road_condition_action_goal. (5) instrument_system. (6) instrument & instrument_action. (7) road_condition_action_constraint.

Folders and labels

Scenarios and *instruments* can have a *label* assigned to them and can be put in (sub-)*folders* (Figure 3.12). A folder has a *folder type* to ensure that *scenarios* and *instruments* do not end up in one folder.

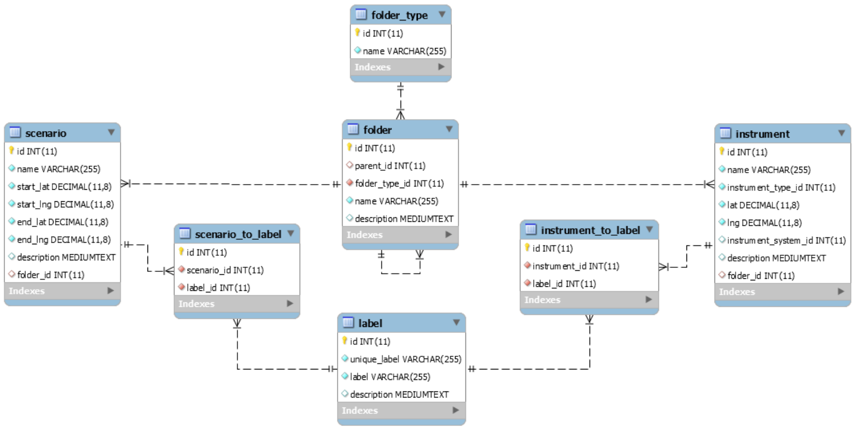


Figure 3.12: The folder and label structure.

Simulation

Besides the decision trees, simulations are stored in the database as well (Figure 3.13). A *simulation* can have multiple *simulation scenes.* These represents a part of a simulation based on time and can have multiple *simulation scene events.* A *simulation scene event* shows what happened during the *simulation scene.* It contains a *road condition type* (to indicate what happened), a value (to indicate the severity) and a *road segment* (to indicate where it happened).

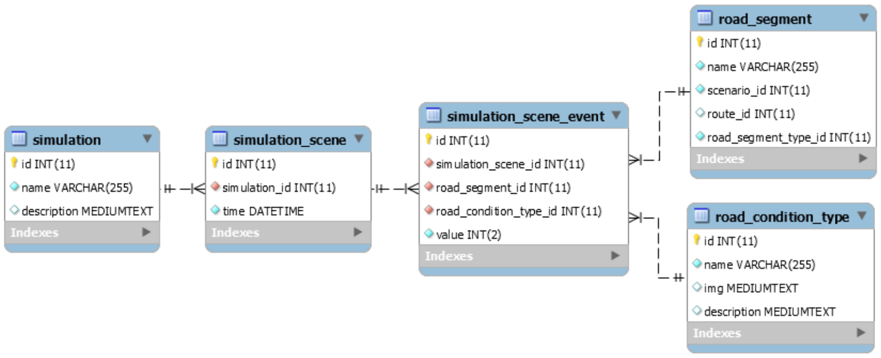
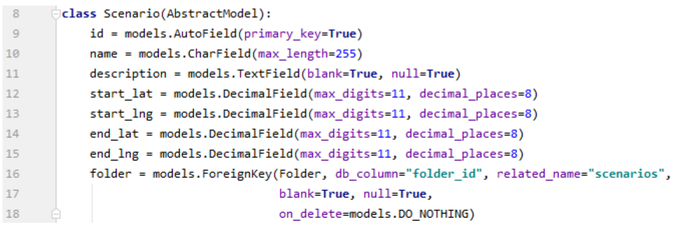


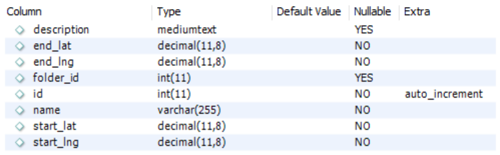
Figure 3.13: The simulation-related tables in the database.

DATABASE CONNECTION

As mentioned in subsection 3.2.1, Django is used to deal with the data in the database. A model in Django represents a table in the database, and is necessary in order to manipulate data from the database. Figure 3.14 shows an example of a model in Django.



```python
class Scenario(AbstractModel):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255)
    description = models.TextField(blank=True, null=True)
    start_lat = models.DecimalField(max_digits=11, decimal_places=8)
    start_lng = models.DecimalField(max_digits=11, decimal_places=8)
    end_lat = models.DecimalField(max_digits=11, decimal_places=8)
    end_lng = models.DecimalField(max_digits=11, decimal_places=8)
    folder = models.ForeignKey(Folder, db_column="folder_id", related_name="scenarios",
                               blank=True, null=True,
                               on_delete=models.DO_NOTHING)
```



Column	Type	Default Value	Nullable	Extra
description	mediumtext		YES	
end_lat	decimal(11,8)		NO	
end_lng	decimal(11,8)		NO	
folder_id	int(11)		YES	
id	int(11)		NO	auto_increment
name	varchar(255)		NO	
start_lat	decimal(11,8)		NO	
start_lng	decimal(11,8)		NO	

Figure 3.14: The Python code above shows the model in Django that represents the corresponding table in the MySQL database.

In order to manipulate data, so-called schemas are composed. These schemas can contain queries and mutations, which are called when the client sends a request to the GraphQL endpoint. Queries are used to retrieve data from the database. When the client sends a query to the server, the corresponding query method in the back-end is called. The method will then retrieve the desired data from the database. The client can use predefined filters in the query to retrieve only specific objects. Mutations are used to either create, update or delete data from the database. When the client sends a mutation to the server, the corresponding mutation method in the back-end is called. This method will perform the desired modifications on the database. Every responsibility of these methods has its own corresponding function. This ensures that the mutation methods themself stay organized and can be reused in other methods. Figure 3.15 shows an example of how a query is sent to the back-end. Mutations are handled in a similar fashion.

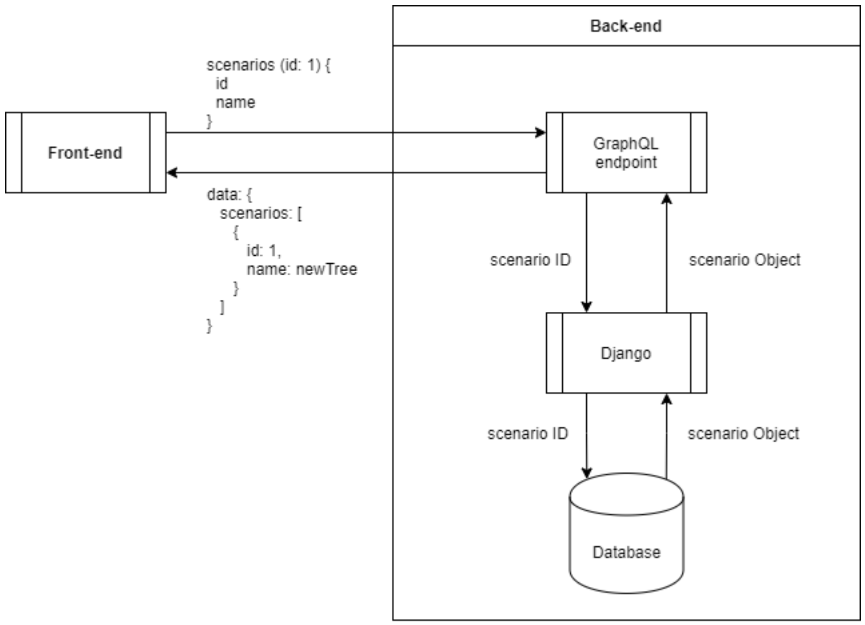


Figure 3.15: Example of a sent query.

IMPORT AND EXPORT

Scenarios

Since parsing is the most convenient method of parsing the decision tree and its nodes, scenarios are imported and exported in JSON format. When an import request containing a JSON file is received, it decodes the JSON and converts it into objects according to the aforementioned models. Those objects are then stored in the database, using the same methods that are used for the mutations described previously. When an export request containing the scenario ID is received, it retrieves the decision tree that corresponds to the scenario ID. The objects of this decision tree is then parsed into a JSON file.

Instruments

Instruments are imported and exported in CSV format, since this is the most convenient method of adding a large number of instruments. When an import request containing a CSV file is received, it decodes the CSV and converts each row to an instrument object. A column should exist of a name (or naam), an instrument type, a latitude (or breedtegraad), a longitude (or lengtegraad) and a description. The instrument object is then stored in the database. When an export request containing the instrument ID(s) is received, it retrieves the corresponding instruments. The instrument objects are then encoded into a CSV file.

SIMULATION

Every 5 minutes the NDW status is received from the server and locally stored in the application. When a live simulation is started on the front-end the back-end receives the IDs of all the road segments on this scenario. The MapBox coordinates are calculated for every road segment, which road segments are checked with the local NDW status. If a NDW situation is found to be on the road segment a representation of these situation are send to front-end.

IMPLEMENTATION CHALLENGES

Scenario constraints

The current traffic scenario model, which is used by the traffic center, shows that the constraint of a certain road action has a possibility to be in any format. Example of these formats are: "if the gate is closed", "broken lamp in the electronic gantries", "queue length of size bigger than X", etc. These constraints do not have a specific format and allow for any input type. To deliver a flexible model to the end-user while creating scenarios, this product allows the user to specify the constraint in a text input. Some of these constraints are difficult to verify automatically. While some others of can not be verified automatically because there is no data about it available(for example the checking whether a gate is closed). This is the reason why the product does not support automatic constraint checking and is something what can be worked on in the future.

Opendata road status

Opendata is a server which hosts public data available via an FTP server containing gzip.xml files. These files are refreshed every minute. Files have an average size of 106MB when uncompressed. Some files contain more than 160.000 entries. Due to the tremendous amount of data it was not possible to store and fetch the road status every minute. The NDW data is fetched every five minutes and is only stored in the back-end, not in the database. While testing a scenario on live data, the data which is relevant to the simulated scenario is stored in the database. To make sure this can be re-winded in the future.

Single coordinate

It was quickly discovered that the lines drawn between the points retrieved from MapBox do not collide with the points received from NDW. NDW points are mapped to the correct road segment by checking whether its location is within a certain radius from the lines drawn between the Mapbox points. If this is the case for such an NDW point, this NDW point is counted as a point on the road. In Figure 3.16 a visible representation of the situation is drawn.

After an error was noticed in these calculations, this methods was revised. The calculation errors were caused by NDW points that in the radius of a road segment line but on the road segment that is in the opposite direction.

The first solution was lowering the radius distance. However, the lines drawn between the MapBox point have inaccuracies relative to the actual road. That is the reason why this solution did not solve all the inaccuracies.

A new calculation process was introduced. The way this new process checks whether a certain point (Point A) is on the same direction as the road is by asking MapBox for the point array of the route between the start of the road segment, point A, and the end of the road segment. If point A is on the same road, the array received has only 1 different point in comparison to the array received from the road segment. However, if point A is not on the same direction as the road, it will have much more additional points in the array. A visible representation of the situation is drawn in Figure 3.17.

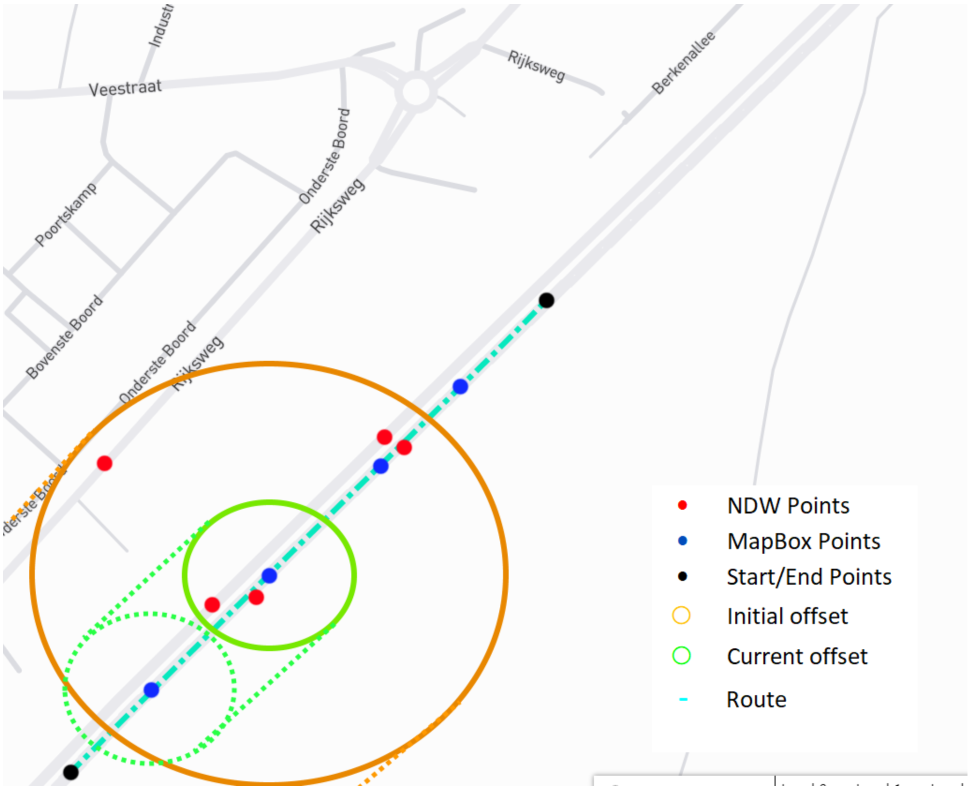
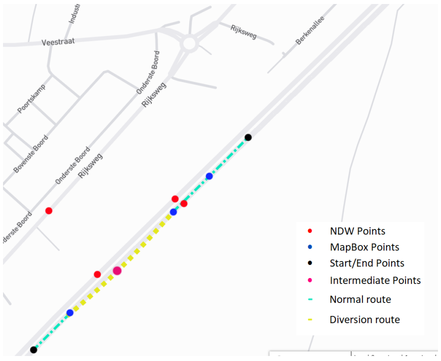


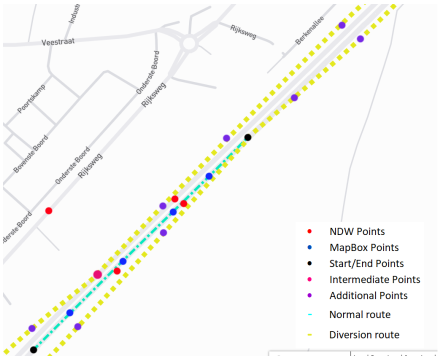
Figure 3.16: Plotted the MapBox points of a route and a location of a measurement point.

Multiple coordinates

Road situations mapped by NDW can have one or more points defined as their location. In case of multiple points, it means that these points define a route on which the situation occurs. Such a route can be mapped by asking MapBox for the given points. These points can be validated with the relevant road segment (Road Segment B) by the single coordinate method Figure 3.2.2. If one of these points is on Road Segment B, the situ-



(a) MapBox points if point is on road



(b) MapBox points if point is not on road

Figure 3.17: Difference between MapBox geolines

ation is considered as relevant to the segment. Because of the limited amount of MapBox calls (related to the free account) the solution that was implemented only checks whether the start or end point collides with Road Segment B.

4
Functionalities

In this chapter the functionalities of the application are discussed. A substantial amount of guiding images are added in Appendix C. These images act as examples of each of the functionalities.

4.1. Features

The following section will briefly discuss each of the features implemented. These features are grouped by the three views of which the application consists. Each view and its features are discussed.

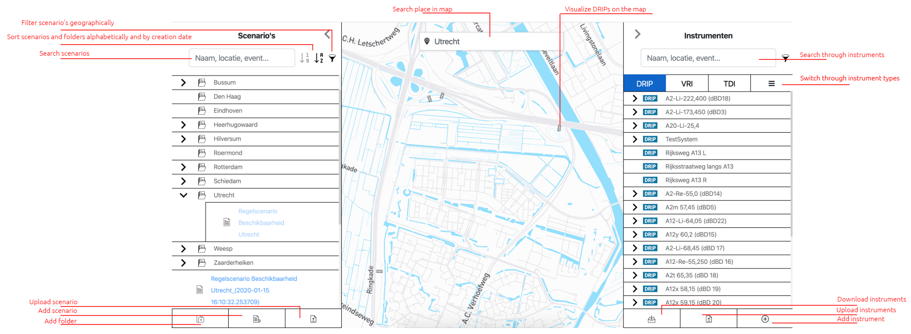
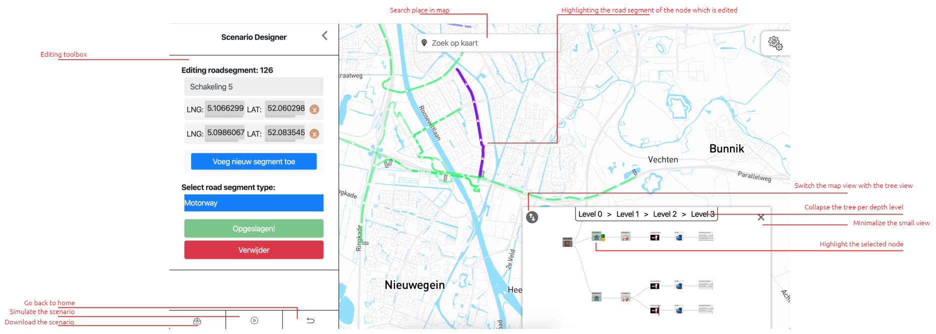


Figure 4.1: The home view.

4.1.1. The Home View

The home view consists of three parts: the left pane, the map and the right pane. In the left pane, the scenarios as well as their folders are visible. On the top of the left pane, the user is able to filter scenarios geographically. This means that the pane only lists the scenarios which are within the bounding box of the map. Furthermore, the user is able to search through the scenarios along with sorting the scenarios alphabetically and by creation date. Additionally, the body of the left pane contains CRUD functionality for the scenarios and the folders. Finally, on the bottom of the left pane, the user is able to upload scenarios from a JSON file, see Figure C.5b. In the map, the user is able to search for certain places, which will result in setting a marker on the map and ‘flying’ to the designated location. Furthermore, the map displays the instruments created by the right pane, see Figure C.4. In the right pane, instruments as well as their instrument actions are visible. On the top of the right pane, the user is able to search through the instruments and to switch between each type of instrument. Additionally, the right pane has CRUD functionality for the instruments and the instrument actions, see Figure C.3. Finally, on the bottom of the right pane, the user can upload/download instruments from/to a JSON file, see Figure C.5a.


Search place in map
Highlighting the road segment of the node which is edited
Editing toolbox
Scenario Designer
Zoek op kaart
Voeg nieuw segment toe
Opgeslagen!
Verwijder
Switch the map view with the tree view
Collapse the tree per depth level
Minimalize the small view
Highlight the selected node
Level 0 > Level 1 > Level 2 > Level 3
Go back to home
Simulate the scenario
Download the scenario
Bunnik
Vechten
Nieuwegein
Parallelweg

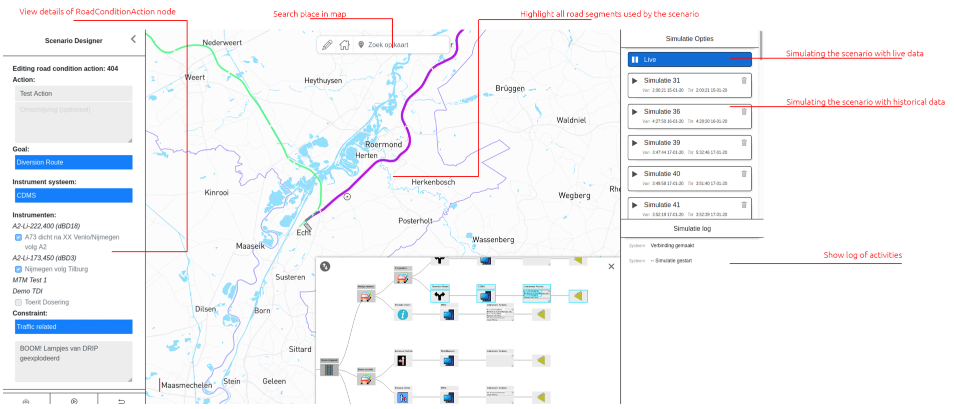
Editing roadsegment: 126
Schakeling 5
LNG: 5.1066299 LAT: 52.060298
LNG: 5.0986067 LAT: 52.083545
Select road segment type:
Motorway

Figure 4.2: The designer view.

4.1.2. THE DESIGNER VIEW

The designer view can be accessed by selecting a scenario in the home view. The designer view consists of four parts: the left pane, the map, the decision tree and the right pane. The right pane is identical to the right pane in the home view, except that it only shows the instruments which are used by the selected scenario. In the left pane, different CRUD functionality toolboxes are displayed depending on the selected type of node in the decision tree, see Figure C.11c. Noticeable is the time input in the toolbox for the *roadCondition* type node. This input allows recurrent events, see Figure C.12. Moreover, the map displays distinctive routes depending on the selected type of node in the decision tree. Initially when a scenario is loaded, the map displays all the routes of the scenario. These include routes for the road segments described in the scenario as well as detours for these routes. When the user selects a node of type RoadSegment, the selected road segment will be highlighted in the map. When the user selects a node of type RoadCondition, the map highlights the road segment on which this condition is applicable, as well as highlighting the detours which will be activated if the condition would be met, see Figure C.7. The decision tree consists of all the nodes used by the scenario.

For a clear overview, each node has an icon indicating the type or subtype of the node. Additionally, some nodes display additional information when hovering them, see Figure C.8 and Figure C.10. Moreover, the user is able to minimize and maximize each of the branches of the decision tree, see Figure C.9. Another feature is that the user can collapse the decision tree per depth level, see Figure C.6. Finally, the decision tree view and the map view can be switched. The smaller view can be minimized, to maximally utilize the available screen.



Scenario Designer

Editing road condition action: 404
Action:
Test Action
Omschrijving (optioneel)
Goal:
Diversion Route
Instrument systeem:
CDMS
Instrumenten:
A2-Li-222,400 (dBD18)
☑ A73 dicht na XX Venlo/Nijmegen volg A2
A2-Li-173,450 (dBD3)
☑ Nijmegen volg Tilburg
MTM Test 1
Demo TDI
☐ Toerit Dosering
Constraint:
Traffic related
BOOM! Lampjes van DRIP geexplodeerd

Simulatie Opties
Live
Simulatie 31 Van 2:00:21 15-01-20 Tot 2:00:21 15-01-20
Simulatie 36 Van 4:27:50 16-01-20 Tot 4:28:20 16-01-20
Simulatie 39 Van 3:47:44 17-01-20 Tot 5:32:46 17-01-20
Simulatie 40 Van 3:49:58 17-01-20 Tot 3:51:40 17-01-20
Simulatie 41 Van 3:52:19 17-01-20 Tot 3:52:39 17-01-20

Simulatie log
Systeem Verbinding gemaakt
Systeem -- Simulatie gestart

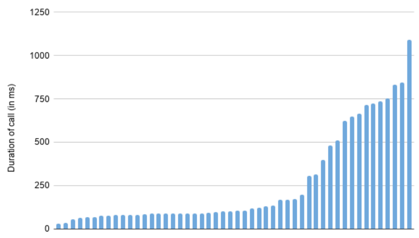
Figure 4.3: The simulator view.

4.1.3. THE SIMULATOR VIEW

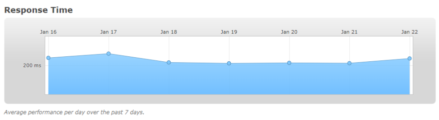
The simulator view can be accessed by selecting a scenario, and pressing on the simulate button. The simulator view consists of four parts: the map, the decision tree, the right pane and the left pane. In the map all the road segments as well as the detours are displayed in blue and green respectively. The decision tree shows the tree of the selected scenario. In the right pane, the user is able to simulate the scenario to live or historical data. When a user is simulating, a log will be displayed indicating what is happening, see Figure C.13. Additionally, when a certain condition is met in the node of type RoadCondition, the branch of the decision tree in which this node occurs will be collapsed. The left pane will slide out and display the details of a selected node. It is not possible to edit this information. Therefore, the full simulator view is read-only.

4.1.4. GRAPHQL API

The communication interface between the front-end and the back-end is done by a GraphQL API. CRUD operation can be made by end user to receive data. The documentation of the GraphQL calls is online available on the same URL. Because of the way GraphQL is set up, the end user can define what part of the data he wants to receive and what part of the data he does not want the receive. The response of the API will return JSON objects representing parts of the database.



(a) Average speed of the API call of the product



(b) Average speed of the API call in the EU

Figure 4.4: Average API speed

4.1.5. Importing & Exporting

The import and export function are created for scenarios and instruments. The import and export function for scenario are in CSV format. The reason for this feature is that end-user can create multiple personal templates which can be used as a basis for creating a new tree. This was a feature of the old prototype and the client requested to add it in the new application. The import and export for instruments is made for the convenience for creating roadside equipment. The roadside equipment are registered at the municipality who own them. They can give the client a CSV containing information about them. After analysing multiple exported CSV files, five field were determined to be required when importing to the application. If the CSV files does contain these fields the instruments are created and stored in the application.

4.1.6. Simulation

The back-end of the application can simulate created scenarios on live road status and stored road status. The live traffic data is mapped to the traffic event for the sake of simplicity in the prototype. However, this can be extended easily by creating new types and mapping these types to the live road events. The back-end sends the road statues by the use of a web-socket and can handle multiple clients at the same time. A new status is sent every 10 seconds and the road status of Opendata is refreshed every minute. Every road status that is simulated by a client is stored in the back-end, thus the client can replay passed events.

4.2. Technical details

4.2.1. Speed

The average API call made by the front-end, to the back-end of the product was measured multiple times by letting an end user execute different tasks in the system while recording the network activity in the browser. In Figure 4.4a the average data-set was used with a size of 50 calls.

The minimum response time of a GraphQL call is 33 ms and the maximum time is 1090 ms. The median of the dataset is 102 ms and the mean is 206 ms. When comparing these statistics to the average speed generated by the WSO2[1] API Cloud, the average speed of the API calls of the product is faster than the average calculated by the WSO2 for their server in Frankfurt. In Figure 4.4b the average speed is displayed from 15-01-2020 until 22-01-2020.

4.2.2. LIVE COORDINATES

The product receives the current traffic situation of the Netherlands using the Opendata FTP link containing .xml.gz files. The traffic situation is defined as the average vehicle speed. Road accidents, events and road works are also stored in the files. All of these events contain a latitude and a longitude which are then mapped to the road.

This was initially resolved by calculating the distance of the point relative to the line

4.2.3. MAPBOX API

To retrieve routes between the defined points in the *RoadSegment* nodes, the MapBox Directions API is used. This is a REST API which is called using the Axios HTTP Client package for Node.JS[2]. These requests take approximately 100 ms per request. The front-end is programmed in such a way that there is a maximum of only two requests simultaneously. Thus, in the worst case scenario, the user will only wait 200 ms for the map to update its road segments.

[1]https://wso2.com/
[2]https://www.npmjs.com/package/axios

5 Code quality & testing

5.1. Code Quality

Django

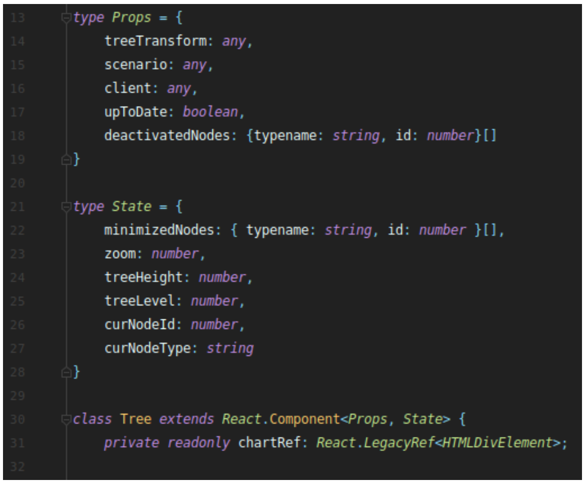
To make sure that the code of the back-end is easy to read and easy to understand, the *pycodestyle* tool was used to format and check the code. When the code reached a certain point of unreadability, the developers enforced this tool and made it obligated on every developer to only push code that is conform to the codestyle. *Pycodestyle* is also known as PEP8, the name was changed to prevent confusion. After this codestyle was enforced, pull-request were reviewed faster due to the readability of the code. PEP8 is one of the most popular codestyle tools for python, so new developers will not have a hard time reading the syntax of the code.

TypeScript

To maintain a clean and clear codebase, the choice for TypeScript was made. It was tough to start with as no team members had any experience with this language. Additionally, the combination of TypeScript and React caused some startup complications. However, after a bit of getting use to, the advantages became apparent.

A major benefit of using TypeScript was that the props and a state of a class had to be defined and typed upfront. Therefore, it was impossible to forget passing a prop or to call an uninitialized state. This probably prevented a large amount of bugs. Additionally, it was easy to grasp what kind of functionality a class had by looking at the interfaces of the props and the state.

However, the group did not manage to make full use of the TypeScript language and often had to revert to no typing or adding a type *any* to a variable or method. This was happened for multiple reasons. Firstly, a third party might not have its types correctly defined. Therefore, it was impossible for the developers to define a type when using data from this third party library or framework. Secondly, GraphQL often returns large queries with vast amounts of varying data. It is possible to generate TypeScript types for a GraphQL schema when the server is written in Node. Sadly, this was not the case.



```
type Props = {
    treeTransform: any,
    scenario: any,
    client: any,
    upToDate: boolean,
    deactivatedNodes: {typename: string, id: number}[]
}

type State = {
    minimizedNodes: { typename: string, id: number }[],
    zoom: number,
    treeHeight: number,
    treeLevel: number,
    curNodeId: number,
    curNodeType: string
}

class Tree extends React.Component<Props, State> {
    private readonly chartRef: React.LegacyRef<HTMLDivElement>;
```

Figure 5.1: Example of React class typing when using TypeScript

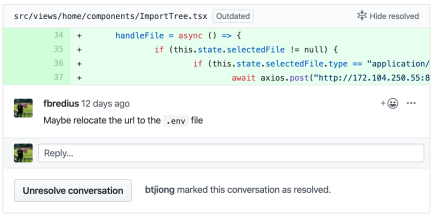
Finally, when using large objects, especially the tree objects, it was often time consuming and hard to define the type. In practice, for nested objects, the type *any* was often used.

Considering these pros and cons, TypeScript did certainly have a positive effect on the codebase. In some cases the group's hand was forced to use no typing or the *any* type. Luckily, this inconvenience did not outweigh the positive effects.

5.2. TESTING

5.2.1. FRONT-END

With regards to the front-end, testing was performed manually. After each adjustment to the project, hot-reloading ensured that the results of the adjustment were immediately visible. Therefore, it was directly clear whether the results of the adjustment were desirable. Moreover, after a feature was completed, a pull-request was created to merge the feature into the master branch of the project. A pull-request would only be accepted after thorough testing by the reviewer. Reviewing a pull-request meant that the reviewer checked each change in the code (Figure 5.2a) as well as testing the adjustments in the browser (Figure 5.2b).



src/views/home/components/ImportTree.tsx Outdated

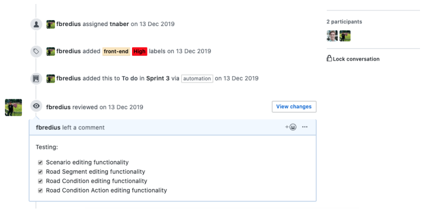
```
+        handleFile = async () => {
+                if (this.state.selectedFile != null) {
+                        if (this.state.selectedFile.type == "application/
+                                await axios.post("http://172.104.250.55:8
```

fbredius 12 days ago

Maybe relocate the url to the `.env` file

btjiong marked this conversation as resolved.

(a) Reviewing each line of code.



fbredius assigned tnaber on 13 Dec 2019

fbredius added front-end High labels on 13 Dec 2019

fbredius added this to To do in Sprint 3 via automation on 13 Dec 2019

fbredius reviewed on 13 Dec 2019

fbredius left a comment

Testing:

- [x] Scenario editing functionality
- [x] Road Segment editing functionality
- [x] Road Condition editing functionality
- [x] Road Condition Action editing functionality

2 participants

(b) Testing the changes made in the pull request.

Figure 5.2: Front-end pull-request review

5.2.2. BACK-END

To write tests in Django, it is recommended to use the built-in unit-test module in the Python standard library. To get an overview of the total code coverage, *coverage.py* is used. *Coverage.py* indicates if a part of the code is executed by the tests or not.

The aim was to achieve > 80% branch coverage for the back-end, which is a high but feasible percentage. The API functions (i.e. queries, mutations and additional methods) had the highest priority for testing, since those functions were the base of the back-end. Most of these functions were unit-tested, which led to an overall branch coverage of 89%, as can be seen in Appendix H. Unfortunately, due to a lack of time and other priorities, the import and export, and live simulation could not be fully unit-tested.

Moreover, the API methods were tested manually by using GraphiQL, which is an in-browser tool to write queries and mutations. The import and export functions were manually tested by sending POST and GET requests to the server. For importing scenarios and instruments this would mean sending a JSON and CSV file, respectively, to the server and checking if the correct data is stored in the database. Exporting was tested by sending the ID(s) to the server and checking if the correct data was sent back. Live simulation was manually tested by logging the information the server sends based on the live data.

5.3. SIG FEEDBACK

In week 6 (week 4 of development) the intermediate code of the system was evaluated by Software Improvement Group (SIG). The results of the evaluation can be found in Appendix G. The code received 3.3 stars from the maintainability model, which means it is market average maintainable. The main points of improvement that were given are Unit Interfacing and Unit Complexity. Furthermore, it was recommended to increase the amount of tests, since it was little compared to the amount of production code.

5.3.1. UNIT INTERFACING

Unit Interfacing refers to code with an above average amount of parameters. Besides that it makes the call to the method more confusing, it might also lead to longer and more complex methods. The solution for this is to substitute groups of parameters with parameter-objects. In this case, the occurrences of methods with too many parameters were *graphene* mutations and queries. The parameters of these methods represent the parameters of the *graphql* mutations and queries. The *graphql* mutations and queries are executed in the front-end, which means that it would take a lot of refactoring in both the back-end and front-end in order to reduce the amount of parameters in these methods. After considering the remaining time of the project, it was decided to prioritize the other points of improvement and functionalities of the application.

5.3.2. UNIT COMPLEXITY

Unit Complexity refers to code with an above average complexity. In other words, methods that contain too many responsibilities or methods with an unnecessary complex implementation of the logic. The solution for this is to split complex methods into smaller methods, each with a different sub-functionality or sub-problem. In this case, the occurrences of methods that are too complex were mostly *graphene* mutations. To solve these complex methods, the mutations were refactored. The functionalities of the original methods were divided into methods, with separate responsibilities.

5.3.3. UNIT TESTING

The submitted code contained a decent amount of tests, however, it was advised to increase the amount of tests compared to the amount of production code, since this would increase the flexibility and stability of the code. More about the test coverage can be found in subsection 5.2.2.

6 PROCESS

In this chapter the project methodology will be discussed. In the beginning of the project roles were assigned and a nonspecific planning was made (see section A.3 and section A.4). In this section these parts will be extended.

6.1. PROJECT PLANNING

Table 6.1 is a detailed planning developed during the course of the project. The planning was kept up to date during the sprint meetings. Throughout the project, the group managed to finish most of the tasks in the corresponding week. Tasks that were postponed, were finished in the subsequent week, without interfering with other tasks planned for that week.

Week	Finished
Week 1	Project Plan (see Appendix A)
Week 2	Research Report (see Appendix B)
Week 3	Initial application design Implement application structure at the front-end Initial database model Django and GraphQL setup
Week 4	Prototype of decision tree (without CRUD) Setup of Apollo and demo server call Database model design WebSocket setup for live simulating Server hosting setup
Week 5	CRUD functionality for decision tree Implementation of primitive map functionality Simulation with live NDW Data (accidents, roadworks)
Week 6	Final version of *designer view* Refactor schemas on server-side Back-end testing First SIG code submission
Week 7	Initial version of *simulation view* Export/Import functionality for a scenario Improve code by using SIG feedback
Week 8	Final version of *designer view* Export/Import functionality for instruments Improve tree by showing more info
Week 9	Bug fixing Finalizing based on feedback of client Back-end testing Project Report Submission Second SIG code submission
Week 10	Final Presentation

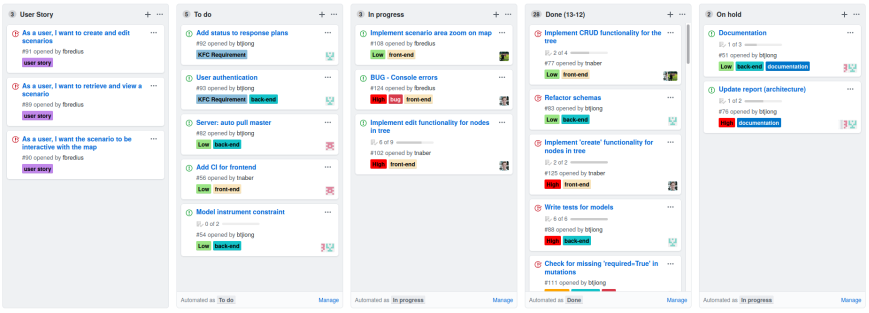
Table 6.1: Detailed project planning

6.2. WORKFLOW

The group uses an agile development style. The weekly sprint meetings consist of the sprint evaluation and the sprint planning.

The sprint evaluation is structured as following. Firstly, the meeting with the client is discussed by looking at the shortages of the demo and the feedback of the client. Then, the unfinished issues on the issue board are discussed. Finally, some points of improvements are constructed.

The sprint planning starts after the evaluation is finished. Taking the points of improvement into account, a new scrumboard is set up. An example of such a scrumboard can be seen in Figure 6.1. This board contains the unfinished and new issues. Addition-


3 User Story

As a user, I want to create and edit scenarios
#91 opened by fbredius
user story

As a user, I want to retrieve and view a scenario
#89 opened by fbredius
user story

As a user, I want the scenario to be interactive with the map
#90 opened by fbredius
user story

5 To do

Add status to response plans
#92 opened by btjiong
KFC Requirement

User authentication
#93 opened by btjiong
KFC Requirement back-end

Server: auto pull master
#82 opened by btjiong
Low back-end

Add CI for frontend
#56 opened by tnaber
Low front-end

Model instrument constraint
0 of 2
#54 opened by btjiong
Low back-end

Automated as To do Manage

3 In progress

Implement scenario area zoom on map
#108 opened by fbredius
Low front-end

BUG - Console errors
#124 opened by fbredius
High bug front-end

Implement edit functionality for nodes in tree
6 of 9
#102 opened by tnaber
High front-end

Automated as In progress Manage

28 Done (13-12)

Implement CRUD functionality for the tree
2 of 4
#77 opened by tnaber
Low front-end

Refactor schemas
#83 opened by btjiong
Low back-end

Implement 'create' functionality for nodes in tree
2 of 2
#125 opened by tnaber
High front-end

Write tests for models
6 of 6
#88 opened by btjiong
High back-end

Check for missing 'required=True' in mutations
#111 opened by btjiong

Automated as Done Manage

2 On hold

Documentation
1 of 3
#51 opened by btjiong
Low back-end documentation

Update report (architecture)
1 of 2
#76 opened by btjiong
High documentation

Automated as In progress Manage

Figure 6.1: Example of the scrumboard in week 3

ally, a small story about the expectations of the new demo is written so this can be compared to the actual demo during the next sprint evaluation. Everyday a daily stand-up was held to evaluate the status of this board, this will be further elaborated on in 6.3.2.

6.3. INTERNAL COMMUNICATION

6.3.1. ROLE DIVISION

In the project plane (see Appendix A) one or two roles were assigned to each team member. These roles were:

Role	Name
Team Leader	Frank Bredius
Lead Communication	Bailey Tjiong
Lead Programmer	Bailey Tjiong
Lead Designer	Frank Bredius
Lead Testing	Leroy Velzel
Scrum master	Titus Naber
Secretary	Leroy Velzel
Product Owner	Kin Fai Chan
Coach/Supervisor	Asterios Katsifodimos

For a more clear definition of the roles, see subsection A.3.4

During the project extra roles were assigned in the form of a front-end team and a back-end team. Moreover, these roles are discussed in 6.3.3

Team	Name
Front-end	Frank Bredius Titus Naber
Back-end	Leroy Velzel Bailey Tjiong

6.3.2. Group meetings
Every morning a daily stand-up starts. In this daily stand-up every group member discusses their status with the group by answering the following questions: what did I do yesterday?, what problems did I face or what held me back yesterday?, and what am I going to do today. In this way every group member is up to date on what everybody is doing. This has a positive influence on the group and the overall participation of all members. Additionally, if anybody gets stuck it is quickly noticed and this person can be helped by other team members. During the span of the project, at work hours, everybody works together at the TU Delft. Since, if there are any questions or problems the team member is quickly helped out.

6.3.3. Intergroup Challenges
The collaboration between group members was pleasant. In fact, there were no intergroup conflicts. Due to the project being a full-stack application a division had to be made between the back-end and front-end developers. Initially, two group members were assigned to the back-end and two to the front-end. This meant there was a natural division in the group. The daily stand-up meetings prevented from a gap growing between these two teams. Later in the project, the back-end team also started working on the front-end, as the server was finished. This was a challenge because this team had to work in a fully new environment. Nevertheless, this was solved by holding a short presentation about the front-end code and frequently asking and answering questions.

6.4. External Communication
6.4.1. Meetings with Client
Every Friday there is a meeting with the client. This meeting starts with a demo of the application to show the progression made during the week. After this status update, the client gives feedback on the new features. Furthermore, the client often passes his domain knowledge on to the team. This is useful as most software developers do not know a lot about traffic management. Optionally, new requirements and their feasibility are discussed.

6.4.2. Meetings with TU Coach
The team aimed for bi-weekly meetings with coach, in this meeting the groups' progress is discussed. Additionally, the groups' communication and with the client is discussed and whether the client is satisfied with the progress or not.

7 EVALUATION

7.1. EVALUATION ACCORDING TO REQUIREMENTS

At the start of the project, requirements were specified using the MoSCoW method. Throughout the project, some requirements were neglected along with the addition of new requirements. In the following table, all requirements (including the neglected ones) are discussed.

7.1.1. MUST HAVES

Requirement	Status	Evaluation
Search scenarios by title	Done	In the top of the *left pane* in the *home view*, the user is able to search through the scenarios. The difficulty of this requirement was that scenarios could exist within a folder, but the search results should also reveal the folder.
Update the map when clicking on a scenario, add a small view of the selected scenario	Done	When a user selects a scenario, the *designer view* will be loaded. In this view, the map shows the routes corresponding to the scenario as well as the decision tree corresponding to the scenario. Initially, the idea was to remain in the same view, since there was no notion of views yet. However, in a later stage of the development, it was clear that for clarity of the user, the application had to be divided into three separate views.

Requirement	Status	Evaluation
When clicking on part of the scenario tree the map indicates...		
The part of the road	Done	Clicking on the nodes of type *RoadSegment* and *RoadCondition*, the map highlights the corresponding routes. Initially, the idea was to only show the routes corresponding to the *RoadSegment* type node. However, during the development, the client requested an additional feature. This included that when a user clicked on the *RoadCondition* type node, the map should highlight the route of the *RoadSegment* node corresponding to the *RoadCondition*, as well as highlighting the routes of the InstrumentActions corresponding to the RoadCondition.
Which instruments are used in the scenario.	Done	In the *right pane*, the instruments belonging to the selected scenario are filtered. When clicking on such an instrument, it will be displayed on the map in addition to its corresponding routes. Moreover, when selecting a node of type *RoadCondition*, the map shows the corresponding instruments on the map.
Which alternative road is proposed to the road-users by the DRIP	Done	These alternative road are initially displayed in green when an user selects a scenario. The user is then able to highlight the alternative roads by either selecting a node of type *RoadCondition* or by clicking an *InstrumentAction* in the right pane.
Which arm of the road-crossing is controlled by the traffic light or traffic officer	Done	The arms of the road are displayed in then, just like the alternative roads on the DRIP, however the instrument that produces this action is of another type.
How the road users should be informed...	Done	This can be created by adding a new instrument type (for example, twitter feed). The name of the action and the description of the action can be added to the scenario (for example, post "TEXT" on twitter account "Scenroads"). A possibility exists automatically send these messages or post in a later stadium. Also the alternative roads are added just like the DRIPS.
Display Instruments		

Requirement	Status	Evaluation
Update the map when clicking on an instrument	Done	When clicking on an instrument, the map ‘flies’ to the selected instrument, and displays it. Then the corresponding *Instrument Actions* will be visible, which can also be selected. Consequently, the map will display the *Instrument Action* in the selected instrument. Moreover, the user can select the routes corresponding to the *Instrument Action*.
Create-Read-Update-Delete instruments	Done	As indicated in the row above, the instrument has a hierarchy with *Instrument Actions* and *Instrument Action Routes* as its children and leaf nodes respectively. To model these instruments was a burden, especially naming each of the relationships resulted in cumbersome entity names such as *RoadCondition-ActionObjectType*
Loading a list of instruments	Done	The *right pane* in the *home* and the *designer view* lists the instruments per type of instrument. During the development, the client wanted additional instruments. These were added to a drop-down menu indicated by a hamburger menu in the *right pane*. The client was highly satisfied with this solution.
Switch between DRIPs, TDIs, VRIs, Matrices, etc.	Done	See the row above.
Create/edit scenarios		
Move components of a scenario in fixed order	Partly done	Initially, the idea was to have a pane with all the node types. The user could then drag and drop a node onto the decision tree. However, it turned out that the client wanted, to some extent, have the decision in a fixed order in which the nodes should occur. Therefore, it was decided that next to each node in the decision tree, an ‘add button’ should be visible, indicating which node could be added after the node. Therefore, the requirement is not explicitly met, considering its initial definition. However, an enhanced solution is implemented.
Simulate scenarios	Done	A challenging and extensive requirement, but finished eventually. At the beginning, the plan was to create simulations by hand. Thus, a user had to click on the map, indicating a time and a type of traffic event. Then the user could test its scenario by simulating it. However, during the development phase, it was clear that the client did not want this functionality anymore. The client wanted to test the scenario against live data and historical data. Therefore, this was implemented instead of the initial plan.

7.1.2. SHOULD HAVES

Requirement	Status	Evaluation
Search scenarios by label	Done	In the search bar in the *left pane* when searching, it will also search through the labels
Move Scenarios between folders	Partly Done	When clicking on the edit button next to the scenario name, the user is able to select a folder in which he/she wants to move the scenario. Initially, drag and drop was desirable. However, since the React Drag&Drop package required that the application used React Hooks, which it did not, the package could not be used. Other packages seemed not feasible to comprehend and implement within the given time span of the project.
CRUD folders	Done	The user is able to create folders, read the folders, update the folders and delete the folders. Unfortunately, since the user is not able to move the scenarios by dragging them between the folders, it is therefore recommended in chapter 9.
Sort Scenarios	Done	The user is able to sort the scenarios alphabetically (a to z, z to a) and by creation date (old to new, new to old).
Sort Instruments	Not done	During the development phase, other requirements had higher priorities than this requirement. Also, the client did not expect this feature to be implemented in the application.
GEO-filter	Done	Both in the left and right pane is an icon of a globe. If the user hovers it, a tool-tip shows stating 'Geo-Filter'. The GEO-filter filters both scenarios and instruments geographically. This means that only the entities are listed which are located within the current view of the map.
Search instruments by label	Not done	The label functionality for instruments was only created in the back-end of the application. In the front-end this functionality is yet to be implemented.
Add labels to scenarios	Done	When creating a scenario, the user is able to add multiple labels to the scenario.
Simulate scenarios with real time data	Done	See ‘simulate scenarios’ in the must-haves

Requirement	Status	Evaluation
Send messages with an alternative route to users	Not done	This requirement was initially proposed by the client. However, it did not have anything to do with the rest of the web-application. It was more of a vision of what the client would have wanted eventually, in a later stage of the development of the web application. Therefore we chose to not implement this feature.

7.1.3. COULD HAVES

Requirement	Status	Evaluation
Create the export function for UML	Done	Users are able to import and export scenarios and instruments. The file type is JSON. Initially the client did not prioritize this functionality, but in a later stage the client put a higher priority on this functionality. Therefore this feature has been implemented instead of some of the features from the should haves.
Save a certain simulated timeframe	Done	Same as the export function, the client did initially not indicate that this functionality was appreciated. However, in a later stage this functionality had a high priority.
Add incident detection to application	Not done	The application was not fully operative yet. That is the reason why it was not used as input to the program. However, it can be added easily to the program, the format of the incident requires only a location and a time. The current incident detection is done by received NDW status messages.

7.2. EVALUATION OF PROCESS

The process of the project can be found in chapter 6. As seen in Table 6.1, the development process started in week 3 and ended in week 9. During this time the group set certain goals and prototype descriptions for each week during the weekly sprint meetings. Setting those goals gave the group a clear vision of what needed to be done during that week. Before the start of the new week, the previous sprint would be evaluated. The group would then discuss if everything for that week was finished and if there were things that could be done better for the next sprint. Besides the weekly sprint meetings, the daily stand-ups helped the group during the development as well. Since the group was generally divided in a front-end and back-end team, the daily stand-ups would keep the developers up to date on the other parts of the project. Moreover, group members could ask for help during these daily stand-ups when struggling with a specific task. The communication between the front-end and back-end teams, and the group overall went very well, which led to a smooth process and a pleasant group atmosphere.

7.3. EVALUATION OF ETHICAL IMPLICATIONS

The application has no user profile or authentication feature. End-users of the product do not provide sensitive information to the server. However, the data that is stored are the response plans, these plans describe what should happen when a certain part of the road cannot be used. The road instruments used are also stored in the application with their corresponding location. If these plans would become public it might be abused by groups or individuals to cause traffic problems or benefit themselves on the road. The group discussed this and decided that abuse of these response plans could not have major consequences on the public. Furthermore, due to these plans not being of high value there is a low change of someone trying to breach the system.

To conclude, the project has a low-profile when discussing ethical implications. When the application is released the server should have proper protection in order to prevent the response plans from leaking.

8 Conclusion

8.1. Goal of the project

The goal at the start of the project was to develop a Smart Traffic Management System for traffic operators. The client already had a prototype available. However, the prototype was not maintainable and not extensible. After thorough consideration, see Appendix A, the decision was made to start from scratch. During the 10 weeks of the Bachelor End Project, a new working system has been designed and implemented.

8.2. Requirements

The requirement evaluation in chapter 7 shows that all *must-haves* have been implemented. Some requirements are implemented differently than initially stated. However, this has all been done after careful consideration and communication with the client. Some requirements from the *should-haves* are not yet implemented. These will be discussed in chapter 9.

8.3. Learning objectives

Starting the project, the team members had almost no experience with the domain knowledge as well as the frameworks used in the project. This resulted in a steep learning curve throughout the project, which makes the project successful in terms of education. This was also made possible by the client, who freed up time regularly, giving feedback and educate us about the domain knowledge.

8.4. Concluding Remarks

Thus, the *must-haves* are met, the client is satisfied with the application, and the project was educational. Therefore, the project can be concluded as a successful project. Satisfaction from the client is evident from the fact the client has already demonstrated the system to their potential clients. The client is also planning on extending the application with the recommendations given in chapter 9. Furthermore, the client has stated that the application is superior to the aforementioned prototype and is therefore planning to continue with our application.

9
Recommendations

In the following sections, recommendations to the application are given which will enhance the product and make it applicable for deployment.

9.1. Requirements

In chapter 7 the requirements were discussed. As stated, some *should haves* were not finished. These requirements will be discussed briefly.

9.1.1. Sort Instruments

In the home view as well as in the designer view, the instruments should be able to be sorted in the right pane. During the development stage of the application, there were a small amount of instruments. Therefore, having a sort functionality was not needed. However, when the application gets deployed, the number of instruments may increase into hundreds of instruments. Having a sort functionality and additional filters in addition to the search functionality can help the user select the instrument more quickly. The sort functionality is already implemented for the scenarios. Therefore, it can quite easily be implemented for the instruments.

9.1.2. Search Instruments by label

To increase the accessibility of the instruments, a search by label for the instruments can be implemented. This is already implemented for the scenarios, so copying this feature can be done effortlessly.

9.1.3. Send messages with an alternative route to users

The ability to send messages with an alternative route could eventually be implemented since, in the future, the client wants the drivers on the road as an additional target-audience.

9.2. ADDITIONAL FEATURES

Additionally to the requirements that were not done, we would recommend to include the following features into the application to enhance it even further.

9.2.1. ADD USER LOGIN

Eventually, the product will have different types of users. These were not mentioned by the client until a late stage of the development phase. Due to time constraints, it was not viable to implement it anymore. Having different roles depending on the type of user will make it possible for people with different jobs to use the application differently. Currently, everyone can access the same functionalities in the application, since the application is built with one type of user in mind.

9.2.2. ADD READ-ONLY FUNCTIONALITY TO SCENARIOS.

In addition to the different users, a possibility is to add read-only functionality to scenarios when they are finished. E.g. the admin can lock scenarios, such that others are not able to make adjustments to the scenario. This would be easy to implement as a read-only functionality is already implemented for the simulator view.

Additional features regarding different types of users can be implemented. However, in order to implement those, the target-audience has to be clear. During the first five weeks of the development phase, it was given that the target audience was the traffic operator itself. However, it appeared that there were different stakeholders for the product. Future developers should be able to extend the application with these features easily, as the application is developed in a maintainable as well as an extendable manner.

9.2.3. DIGITAL CIRCULATION

Digital circulation of the scenarios to a different person for approval of the scenarios (instead of paper circulation) is a requirement of the road authorities like Rijkswaterstaat.

9.3. TECHNICAL RECOMMENDATIONS

Apart from features, there are also technical aspects which could be improved in the future.

9.3.1. REFACTOR WORKSPACE

The *Workspace.tsx* file has the tendency to grow when additional features are implemented. This would result in a file that will not be readable and should thus be refactored. As stated in chapter 4, currently the file is comprehensible.

9.3.2. USING REACT HOOKS

Stated in chapter 4, the decision was made to use React Classes instead of React Hooks. As can be seen with the React Drag and Drop framework[1], React Hooks is expected to be used in combination with the framework. It is inevitable that more frameworks will

[1]https://react-dnd.github.io/react-dnd/about

assume React Hooks to be used in the code. Therefore, when extending the application, it is recommended to use React Hooks.

REFERENCES

REFERENCES

[1] Y. Nader, *What is django? advantages and disadvantages of using django,* (2019).

[2] *Django models,* (2018).

[3] *Graphql vs rest - a comparison,* (2017).

[4] *Graphene-django docs,* (2019).

[5] D. Clegg and R. Barker, *Case method fast-track: a RAD approach* (Addison-Wesley Longman Publishing Co., Inc., 1994).

[6] *General guide for tu delft ti3806 computer science bachelor project,* (2019).

[7] H. van Lint, *Lecture notes in ct2710,* (2010).

[8] N. D. Wegverkeersgegevens, *Actuele verkeersgegevnes,* (2019).

[9] M. Jarke, X. T. Bui, and J. M. Carroll, *Scenario management: An interdisciplinary approach,* Requirements Engineering **3**, 155 (1998).

[10] P. K. Dey, *Project risk management: a combined analytic hierarchy process and decision tree approach,* Cost Engineering **44**, 13 (2002).

[11] N. Senroy, G. T. Heydt, and V. Vittal, *Decision tree assisted controlled islanding,* IEEE Transactions on Power Systems **21**, 1790 (2006).

[12] J. Verdiesen, M. Loot, and A. Smienk, *Regelscenario's: wat levert het op?* (Colloquium Vervoersplanologisch Speurwerk, 2011).

[13] E. Scheerder, J. van Kooten, G. Martens, J. Birnie, and M. Westerman, *Werkboek regelscenario's* (Rijkswaterstaat, 2006).

[14] *Stack overflow developer survey 2019,* (2019).

[15] E. Wohlgethan, *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js,* Ph.D. thesis, Hochschule für Angewandte Wissenschaften Hamburg (2018).

[16] T. Williams, *Relational sql vs. non-relational nosql databases,* (2019).

[17] J. Homan, *Relational vs. non-relational databases: Which one is right for you?* (2014).

[18] C. Arsenault, *The pros and cons of 8 popular databases,* (2017).

[19] M. Fowler and M. Foemmel, *Continuous integration,* Thought-Works) http://www.thoughtworks. com/Continuous Integration. pdf **122**, 14 (2006).

[20] *Testing overview,* (2019).

[21] *Testing in django,* (2019).

Glossary

instrument A (technical) tool to inform and/or directing road users.. 3

matrix A digital traffic sign above a highway to inform drivers. Also known as Matrix Signaalgever Informatie (MSI). 55, 64

MoSCoW A prioritization method where M is Must have, S is Should have, C is Could have and W is Won't have. 2, 4

response plan A predefined set of steps by traffic managers in order to keep optimize the traffic flow in a specific scenario. In Dutch known as *regelscenario*. 1, 51, 53, 61

scenario See response plan. 4

traffic jam Traffic congestion where the vehicles are fully stopped for a period of time. 60

traffic manager The traffic manager in the traffic centre is responsible for managing the traffic fast and safely.. 61

user stories Small stories describing, in steps, how a user uses different features within the application. 66

Acronyms

AI	Artificial Intelligence. 54
BEP	Bachelor End Project. iv
CRUD	Create-Read-Update-Delete. 14
DRIP	Dynamisch Route-informatiepaneel. 4, 55, 64
FCD	Floating Car Data. 54, 60
GIS	Geographic Information Systems. 54
HOC	high-order components. 8
LOC	lines of code. 66
MSI	Matrix Signaalgever Informatie. 51
MTM	Motorway Traffic Management. 4
NDW	Nationale Databank Verkeersgegevens. 16, 53, 60
REST	Representational state transfer. 15, 16
SIG	Software Improvement Group. 32
SQL	Structured Query Language. 68
TCC	Traffic Control Central. 59
TDI	Toeritdoseringsinstallatie. 4, 55, 64
VRI	Verkeersregelinstallatie. 4, 55, 64

A

Project Plan

A.1. Company Background

Scenwise B.V. is company with a lots of experience in traffic management & smart mobility domain. We work together with our partners to develop innovative software for the domains: - traffic management (e.g. automatic incident detection, response plans, etc.) - data science & visualisation. (e.g. traffic monitoring, data fusion, Big Data) Our customers are the Dutch Highway Agency (Rijkswaterstaat), Nationale Databank Verkeersgegevens (NDW), provinces, large cities, ITS system suppliers, Feyenoord stadium and also the city of Edmonton in Canada.

A.2. Project

A.2.1. Project Description

Scenwise has developed the *ScenarioDesigner* to support road authorities in creating response plans. The response plans describe which actions the operator should take in order to manage different traffic situations such as accidents, large scale events. The first prototype is an offline application with some limitations, e.g. no real-time traffic data can be used within the current software and there is no mechanisms to automatically trigger a plan. At the same time, the demand for response plans with automatic triggers and seamless integration with in-car information grows. In order to meet the future needs of our customers, we plan to start the development of a Decision Support Application using a new approach to incorporate different new and state-of-the art technologies. This project is to develop a Smart Traffic Management System to help traffic operators to make fast and safe decisions. Within this project, a team of students will work together to conduct research, make design decisions, develop the application inclusive testing. The application will incorporate a decision tree or a rule engine approach. The back-end should be a big data platform using different open data (both real-time and historical). The front-end should be web-based using modern graphical interfaces. This should include a sophisticated decision tree editor which interacts with a 3D map supported with real-time graphics to monitor and interact with the traffic conditions. The development of a simulator to trigger the different traffic situations is also a part of the project. This is an innovative project with a lots of technical challenges. We are looking for enthusiastic students with skills and interest in Geographic Information Systems (GIS), Artificial Intelligence (AI), visualization techniques and algorithms who want to take the challenge to develop the next generation of decision support application. We intend to put this application into operation to support our customers. Our business partner in Canada also intend to integrate the functionality into their ITS (Intelligent Transport System) platform for their customers in North America and Asia.

A.2.2. PROJECT GOAL

The goal of the project is to develop a Smart Traffic Management System to help traffic operators to make fast and safe decisions. When a specific situation is recognized, a predetermined set of actions (scenario) is suggested. This system should also be able to detect traffic accidents.

A.2.3. ASSIGNMENT SPECIFICATION

In the first two weeks the group will research the problem and the optimal approach to the problem. This will be motivated in the research report. After the research phase the group will develop a smart traffic management system, which supports digital response plans is also able to detect traffic accidents.

A.2.4. FINAL PRODUCTS

The final product which will be delivered is a comprehensive tool which analyses a traffic situation and proposes a predetermined scenario accordingly. There will be a deeper focus on accident detection. An already existing accident-detection system for highways using loop-data will be improved. For provincial roads, a different approach using Floating Car Data (FCD) will be implemented. The final product is meant as a prototype.

A.2.5. MUST-, SHOULD-, COULD-HAVES

As proposed by [5] , we divide the demands and constraints regarding the project through the MoSCoW method.

SCENWISE APPLICATION; SCENARIODESIGNER

MUST:

- Search and view scenarios.
  - Search scenarios by title.
  - Update the map when clicking on a scenario, add a small view of the selected scenario.
  - When clicking on part of the scenario tree the map indicates:
    - The part of the road
    - Which instruments are activated
    - Which alternative road is used
- Display instruments.
  - Update the map when clicking on an instrument.
  - Create-Read-Update-Delete instruments.

    - Switch between Dynamisch Route-informatiepaneel (DRIP), Toeritdoseringsinstallatie (TDI), Verkeersregelinstallatie (VRI).
- Create/edit scenarios
    - Drag & drop components of a scenario (with location) in fixed order
        - Parts of the road, Current segments
        - Condition
        - Actions:
            - DRIP messages
            - Matrix messages
            - TDI, VRI
        - Constraint
    - When clicking on part of the scenario tree the map indicates:
        - The part of the road
        - Which DRIPs are activated
        - Which alternative road is used
- Simulate scenarios (Congestion, Accident, Roadworks, Events)
    - Create congestion, accidents, roadworks and events by clicking on the map.
    - Time input.
    - Congestion level input.

SHOULD:

- In the search bar:
    - Search scenarios by label.
    - Move scenarios between folders.
    - Create-Read-Update-Delete folders.
    - Sort scenarios.
- In the instruments bar:
    - Sort instruments.
    - Search instruments by label.
- In the create/edit scenario view:
    - Add labels to scenarios; such as the city, road name, event name.
- Simulate scenarios with real time data.
- Send messages with an alternative route to users.

COULD:

- Create and extend the export function for UML.
- Save a certain simulated time-frame.
- Add incident detection to application.

WON'T:

- Build the navigation app.
- Control the roadside equipment.

NDW DATA COLLECTION

MUST:

- Query NDW every minute, and store in a DB
- Collect Travel time (traveltime.xml.gz)
- Collect Traffic speed (trafficspeed.xml.gz)

- Collect Incidents (incidents.xml.gz)
- Run on an Ubuntu 16.04 or higher
- Report:
  - How much the space on the HDD increases each day.
  - How much the query time increases if the size of the DB increases.
  - How much the save time increases if the size of the DB increases.
  - How much RAM and CPU is used while and saving and querying using HDD.
  - How much RAM and CPU is used while and saving and querying using SSD.
  - SSD vs. HDD (Storage/query time)

Incident Detection Validation

MUST:

- Validate a hit with of the algorithm with NDW, Waze or both in a certain time interval.
- Run on an Ubuntu 16.04 or higher
- Report showing percentages correct with in 1, 2, 5, and 10 minutes.
- Report showing percentages not detected.

A.3. Project Setup

At the start of the project, various agreements regarding the project have to be made. These agreements are noted in the following subsections.

A.3.1. Research

The project will start with a two week research phase. During this phase we will look into the data-sets and documentation given by the Client, as well as watching video lectures regarding the subject. Design choices will be motivated in the research report.

A.3.2. Development

During the 8 weeks lasting development phase, we will further extend the prototype created by the Client. This prototype has various limitations, such as the fact that it is only available offline. These limitations will be written into *user stories*, following the Scrum Method. User stories are further extended into Issues described on the GitHub page of the repository. Every two weeks a new sprint is started. At the start of each sprint a sprint backlog is made, and at the end a sprint retrospective will be written.

A.3.3. Client and TU Coach

The project is accompanied by a Client and a TU Coach [6]. The Client is the real-world stakeholder, who commissions the team to develop a software solution that addresses a specific problem. Kin Fai Chan is the Client of our project. The role of the TU Coach (EWI Coach) is to represent the educational interests of the TU Delft. This role is fulfilled by Asterios Katsifodimos.

A.3.4. TEAM-MEMBERS

Each team member is required to work a total amount of 420 hours divided over 10 weeks. During these weeks, it is expected that each of the team members will behave professionally. The time spend on the project does not necessarily have to be at the TU Delft campus. We have defined various roles important during the project. Each team member is assigned to one or two roles.

Role	Name
Team Leader	Frank Bredius
Lead Communication	Bailey Tjiong
Lead Programmer	Bailey Tjiong
Lead Designer	Frank Bredius
Lead Testing	Leroy Velzel
Scrum master	Titus Naber
Secretary	Leroy Velzel
Product Owner	Kin Fai Chan
Coach/Supervisor	Asterios Katsifodimos

Definition of the roles:

Role	Definition
Team Leader	Makes sure group is working optimal. Keeps an eye on the deadlines. Makes sure every team member is doing their tasks.
Lead Communication	Communicates between group, TU Coach and Client.
Lead Programmer	Makes sure code quality is as high as possible. Checks merge requests.
Lead Designer	Makes sure user interface is intuitive as well as appealing.
Lead Testing	Makes sure everyone is testing their code sufficiently.
Scrum master	Is responsible for the scrum process. Leads the sprint backlog and retrospective meetings.
Product Owner	Real-world stakeholder, who commissions the team to develop a software solution that addresses a specific problem
Coach/Supervisor	Represent the educational interests of the TU Delft

A.3.5. NECESSARY SKILLS & TECHNIQUES USED

The following items are necessary skills or interests for this project:

- HTML, CSS, Javascript (Node.JS and React Framework)
- PostgreSQL, MongoDB
- Working with XML files

- Geographic Information Systems (GIS)
- Working with Scrum Methodology

A.4. Approach
The project will be approached according to our planning as well as the agreements made in the meetings section.

A.4.1. Planning
A thorough planning has been made via Google Drive.

In summary:

1. Week 1 - Project Plan
2. Week 1 & 2 - Research Phase
3. Week 3 to 10 - Main Phase
4. Week 6 - First submission of software to SIG.
5. Week 9 - Deadline Final Report, second submission of software to SIG.
6. Week 10 - Final Presentation.

A.4.2. Meetings
Group meetings
The group will meet, when possible and necessary, at the TU Delft during weekdays. Team members are obliged to be available during workdays between 9AM and 5PM. On Mondays the group will determine the requirements for the upcoming week and divide the tasks. If possible, the group will meet once a week with the coach.

Client meetings
The group will meet once a week with the client to discuss the progress and future work. As agreed with the client, the meetings will take place on Friday afternoons.

Location
The meetings will take place in meeting rooms in either Building 35 (Drebbelweg) or the TU Delft Library, if possible. If there are no meeting rooms available, the group will find another working place in Building 36 (EWI) or Building 35.

A.4.3. Financing
As agreed with the Client, the students will receive internship compensation and travel allowance conform market prices.

B
Research

B.1. Introduction
The first two weeks of the Bachelor End Project are dedicated to the research phase. During this phase, the goal is to get an overview of the current situation, to learn how others have solved related problems, to define the problem clearly, and to deduce which design choices should be made. The following sections will further elaborate on the aforementioned topics.

B.2. Preliminaries
B.2.1. Traffic congestion
Traffic flow is a complex concept due to a lot of factors having an influence on it [7]. One of the major problems in traffic management is that the traffic flow has a negative feedback loop. When a small source of congestion appears on the road this congestion can grow into a large traffic jam. An example of such a small congestion is someone breaking too hard or interference from an on or off ramp. Therefore, it is crucial to maintain a steady traffic flow before any congestion can arise.

B.2.2. Traffic management in the Netherlands
Traffic in the Netherlands is managed by the five Traffic Control Central (TCC). The TCCs are part of 'Rijkswaterstaat', the Ministry of Infrastructure and Water Management. Next to the TCCs, the larger municipalities in the Netherlands manage their own traffic. They do this in cooperation with Rijkswaterstaat. Cooperation is of significant importance since most of the congestion can not be solved by a municipality itself. Their method with regards to traffic congestion and management is described in subsection B.4.1.

B.2.3. Data available

In the Netherlands a traffic database is kept up to date by Nationale Databank Verkeersgegevens (NDW). This organization has multiple sources where it gathers its data from [8].

Firstly, scattered across the country are 37,000 measuring points, these point send their data every minute to this central database. Such measuring points can measure the following values:

- Traffic intensity (the amount of vehicles passing the measuring point)
- Point speed (average speed of the vehicles passing the measuring point)
- Realised or estimated travel time
- Vehicle category (category derived from the length of the passing vehicle)

All data released by the NDW is converted to DATEX II [1] format.

Secondly, Floating Car Data (FCD) is used to track the current state of roads. FCD data is derived from mobile phones, which send their location or whose locations is detected by sensors next to the road. Using this data an estimate can be made of the situation on the road.

B.3. Related work

This work mainly has related work in the traffic management and process management fields. The following papers have been found with an interesting overlap with the current project.

Firstly, [9] takes an in-depth look at scenario management. It states that a scenario is a broad term and can be conceived in different ways. The authors focus on strategic management, human–computer interaction, and software and systems engineering as the three main disciplines of scenario management [9]. This paper more clearly defines what kind of scenario's exist and which scenario will be used in this project.

Secondly, [10] focuses on the use of decision trees in risk analysis. This paper mainly looks at a basic management project, however, the model proposed in this paper on risk analysis is also applicable in traffic management. Especially the focus on decision trees is useful for this project.

Finally, [11] proposes a framework to perform controlled separation of electrical networks. A tool is proposed which uses a decision tree to determine the next, most beneficial, step. In this report a decision tree will be used in a similar fashion, to recommend a next step to benefit a larger system. A large difference in contrast to the system proposed in this paper is that the decision trees in [11] are trained and generated.

B.4. Problem Definition and Analysis

B.4.1. Problem Definition

Traffic jams arise for various reasons, for instance due to a collision, road works, weather, and much more. Some of these traffic jams resolve themselves, however most of them do not. Hence, to solve this problem, a traffic jam must first be located. To locate traffic jams, sensors on the highway are used to track the speed and density of the traffic. Once

[1]https://datex2.eu/

the severity of the traffic jam is measured, the traffic manager has to redirect the traffic heading towards the jam to an alternative route to reduce the problem. According to the client traffic managers have to react to this traffic situation within 1 hour to control the situation on the road. These processes are not automated and manual labor is required to check these situations and to act according to the different situations. The system is in need of a smart automated system that recognizes certain scenarios and act according to rules defined in these scenarios.

B.4.2. PROJECT GOAL

The goal of the project is to develop a Smart Traffic Management System to help traffic managers to make fast and safe decisions. End users of this product are able to define scenarios in the front-end of the system. If these scenarios are active, the system should keep track of the current traffic situation and recognize what state the traffic is in. When a specific state is recognized, a predetermined set of actions in the scenario is executed or suggested. If an action cannot be executed or does not show the wanted result, a message should be displayed to the operator. This system should eventually be able to detect traffic accidents.

B.4.3. CURRENT STATE

Currently there is no automated way of managing traffic congestion. Rijkswaterstaat introduced so called response plans in 2006 [12]. These plans are still part of the method used currently. Response plans are predefined scenarios that describe how a traffic situation should be handled as well as what goal should be pursued in that situation [13]. Furthermore, it describes when a certain scenario should come into effect and which actions should be taken in order to achieve the determined goal. Response plans focus on bottlenecks and are introduced for the following situations:

- Traffic congestion in regular road networks
- Road works
- Events
- Emergencies

Response plans are composed, managed and optimized according to the process described in Table B.1. Examples of response plans that are currently used are showed in Appendix D.

Phase	**Description**
Startup	During the first phase the involved parties, targeted situations, goals, and conditions are determined.
Preparation	Before the development of the response plan, it is necessary to check the policy principles, the control strategies along with the reference framework. Consecutively, the clusters of bottlenecks are selected for which the response plan will be developed. It is important to analyze the selected clusters to know what the causes and consequences of the bottlenecks are.
Development	During this phase the response plan is developed. Specific actions are worked out in a flowchart. This flowchart indicates when a specific action is taken and ended.
Implementation	This is the last phase before operating the response plans in practice. During this phase agreements are made about technical demands and who is responsible for the management.
Employment and evaluation	After all the preparation of the previous phases is done, the response plan can be employed. After the employment, the response plan is evaluated and possibly updated and corrected. This last phase should be constantly repeated.

Table B.1: Process of composing, managing and optimizing response plans [13]

CURRENT PROTOTYPE

Scenwise B.V. has already implemented a prototype to deal with the issue of automating traffic congestion management. After thorough research, which included more than 4 years of work, the conclusion is that the current method is infeasible. Response plans are made of flowcharts, which interchange actions and conditions. This method of describing a plan is impractical when trying to automate it. Therefore, the solution was to use a decision tree instead. The current prototype consists solely of a basic decision tree maker. This prototype has to be extended extensively in order to be useful. See Figure B.1 for the current prototype. The requirements of the prototype can be found in subsection B.5.2.

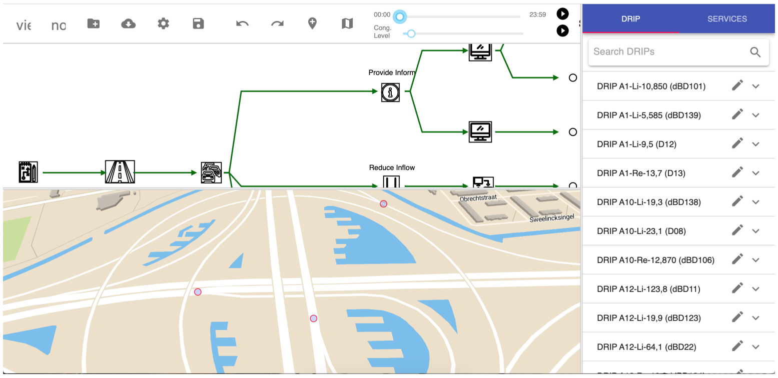


Figure B.1: Initial prototype

B.5. Requirement Analysis

Requirement analysis focuses on the tasks that determine the needs or conditions to meet the project. It takes into account the possible conflicting requirements of the various stakeholders. Additionally, requirements analysis is about analyzing, documenting, validating, and managing software/system requirements.

B.5.1. Stakeholders

A stakeholder is one or multiple persons which have interests concerning the system. In the project, five different stakeholders are identified. These stakeholders are the project team, the traffic users, the traffic managers, Scenwise B.V., and the TU Coach [6].

Project Team

Firstly, the project team is a stakeholder. Since the team members all have their strengths and weaknesses, it is important that the project is divided accordingly.

Traffic Users

The traffic users are group indirectly benefiting from the product. If the product works correctly, it should improve the speed and accuracy by which roads will be managed. Consequently, the flow on the highways increases.

Traffic Managers

The traffic managers are the end users of the product. The current situation in which traffic managers operate is cumbersome. If implemented correctly, the product will increase the efficiency and ease in which the traffic managers function. Intuitiveness and completeness of the software is of high importance. Therefore, usability tests are needed to test whether the software fulfills these requirements.

Scenwise B.V.

Scenwise B.V. is the Client. As mentioned in [6], the Client is the real-world stakeholder, who delegates the team to develop a software solution which addresses an unequivocal problem. The Client, as well as the traffic managers, decide whether the product meets the requirements.

TU Coach

The role of the TU Coach is to represent the educational interests of the TU Delft. The TU Coach makes sure that the learning objectives of the bachelor project [6] are met. The primary function of this role is guiding the bachelor students in applying their skills and knowledge within the bachelor project. The TU Coach should support the team in choosing and making use of the appropriate software development methodology.

B.5.2. Requirements

As proposed by [5], the requirements regarding the project were divided through the MoSCoW method. In this manner, the various needs of the different stakeholders are prioritized. Since the project consists of three sub-problems, three MoSCoW lists are described. The scenario designer has a large priority over the other two projects. Two non-functional requirements have been stated by the client:

- Run the program on a Ubuntu 16.04 machine.
- Make the program decide no which nodes of the scenarios are be active within 2 seconds.

Scenwise application; ScenarioDesigner

MUST:

- Search and view scenarios.
  - Search scenarios by title.
  - Update the map when clicking on a scenario, add a small view of the selected scenario.
  - When clicking on part of the scenario tree the map indicates:
    - The part of the road
    - Which instruments are activated
    - Which alternative road is used
- Display instruments.
  - Update the map when clicking on an instrument.
  - Create-Read-Update-Delete instruments.
  - Switch between Dynamisch Route-informatiepaneel (DRIP), Toeritdoseringsinstallatie (TDI), Verkeersregelinstallatie (VRI).
- Create/edit scenarios
  - Drag & drop components of a scenario (with location) in fixed order
    - Parts of the road, Current segments
    - Condition
    - Actions:
      - DRIP messages
      - Matrix messages
      - TDI, VRI
    - Constraint
  - When clicking on part of the scenario tree the map indicates:
    - The part of the road
    - Which DRIPs are activated
    - Which alternative road is used

- Simulate scenarios (Congestion, Accident, Roadworks, Events)
  - Create congestion, accidents, roadworks and events by clicking on the map.
  - Time input.
  - Congestion level input.

SHOULD:

- In the search bar:
  - Search scenarios by label.
  - Move scenarios between folders.
  - Create-Read-Update-Delete folders.
  - Sort scenarios.
- In the instruments bar:
  - Sort instruments.
  - Search instruments by label.
- In the create/edit scenario view:
  - Add labels to scenarios; such as the city, road name, event name.
- Simulate scenarios with real time data.
- Send messages with an alternative route to users.

COULD:

- Create and extend the export function for UML.
- Save a certain simulated time-frame.
- Add incident detection to application.

WON'T:

- Build the navigation app.
- Control the roadside equipment.

NDW DATA COLLECTION

MUST:

- Query NDW every minute, and store in a DB
- Collect Travel time (traveltime.xml.gz)
- Collect Traffic speed (trafficspeed.xml.gz)
- Collect Incidents (incidents.xml.gz)
- Run on an Ubuntu 16.04 or higher
- Report:
  - How much the space on the HDD increases each day.
  - How much the query time increases if the size of the DB increases.
  - How much the save time increases if the size of the DB increases.
  - How much RAM and CPU is used while and saving and querying using HDD.
  - How much RAM and CPU is used while and saving and querying using SSD.
  - SSD vs. HDD (Storage/query time)

INCIDENT DETECTION VALIDATION

MUST:

- Validate a hit with of the algorithm with NDW, Waze or both in a certain time interval.
- Run on an Ubuntu 16.04 or higher

- Report showing percentages correct with in 1, 2, 5, and 10 minutes.
- Report showing percentages not detected.

B.6. Development Method

The project will start with a two week research phase. During this phase the datasets and documentation given by the Client will be evaluated, as well as watching video lectures regarding the subject.

During the 8 week lasting development phase, a new prototype will be developed. The most prominent features will be written into user stories, following the Scrum Method. These user stories are further extended into Issues described on the GitHub page of the repository. Every week a new sprint is started. At the start of each sprint a sprint backlog is made, and at the end a sprint retrospective will be written.

B.7. Design Choices

Due to a number of factors the decision has been made to abandon the existing codebase and start from scratch. Firstly, the existing code was not build in a maintainable or scalable manner. There are no comments, the classes exceed a 1000 LOC and most data structures are hardcoded. Secondly, the repository does not contain any test code or any proof of user integration tests. Finally, the database model is not the optimal choice for the given data. Additionally, the original project is written in React[2], this framework will also be used in the new project. If it is needed to use code from the old codebase, React components can be transferred relatively easily. In the following sections all design choices will be elaborated on.

B.7.1. Version Control

The current prototype is deployed on Beanstalk [3]. However, this service does not come with free access for students. Therefore, GitHub is used for version control. Different repositories will be used for both the back-end and the front-end of the web application. Pull requests to the master should be reviewed by at least two other team members. Other metrics regarding the code quality are described in subsection B.5.2.

B.7.2. Language

The front-end code will be written in JavaScript, HTML5, and CSS. The back-end code is written in Python 3.0. In the following sections, the decisions for the front-end as well as the back-end will be further elaborated on.

B.7.3. Front-end

Focusing on the front-end part of the web application, a variety of frameworks and libraries exist to choose from. The decision is based on the comparison between three widely adopted technologies, as can be seen in Figure B.2. The current prototype is written using the React Library. It is worth mentioning that two team members already had

[2] https://reactjs.org/
[3] https://beanstalkapp.com/

experience with React.

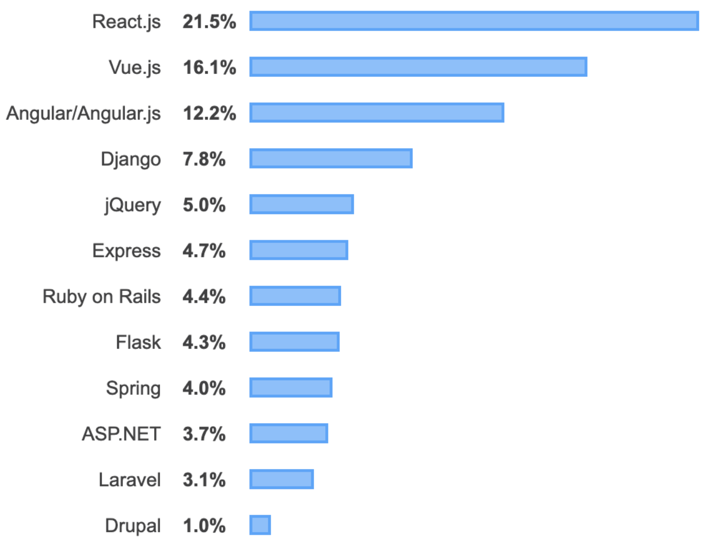


Figure B.2: Most wanted web frameworks according to Stack Overflow [14]

COMPARISON BETWEEN ANGULAR, VUE AND REACT

Angular[4] is somewhat restricted in its development process, e.g. Webpack[5] is standard used for bundling. Thus, if another module bundler is preferred, it could get tricky. Whereas React leaves the *M* and *C* part of the Model-View-Controller (MVC) design-pattern to the developers [15]. Furthermore, React tends to be more flexible than Vue[6], which leaves much responsibility to the developers. Since this restriction of Angular, the flexibility of React, the original code being written in React (which leads to easier transferability) as well as the current experience in React, makes React the most natural choice.

In addition to the React framework, Redux will be used. Redux helps with state management when the application eventually gets too large to manage state transfer between components directly. Additionally, to visualize the decision tree, which can be created in the application, d3js[7] is used. Finally, MapBox GL JS[8] is used to retrieve the map used in the application. Mapbox GL JS is a library which uses WebGL to show an interactive Map, which can be expanded by using various plugins. As an alternative to MapBox GL JS is to use Google Maps. However, since MapBox GL JS is highly customizable, as well

[4]https://angular.io/
[5]https://webpack.js.org/
[6]https://vuejs.org/
[7]https://https://d3js.org/
[8]https://www.mapbox.com/

as free for our purposes, it is preferred over Google Maps. A brief overview of the other popular Map APIs can be found in Table B.2.

Map API	Pro	Con
Google Maps	Biggest in the field	Pricey, limited to 10 Queries per second
Mapbox GL JS	Highly customizable, good for live event data	Weak coverage in places such as India
Bing Maps	Integration with Azure	Pricey
Foursquare	N.a.	Used mainly for venues
Yelp	N.a.	Used mainly to find nearby hotels etc.
Yandex	N.a.	Mainly for Russian customers
Fencer	N.a.	Used for IoT
Mapillary	Ability to recognize traffic signs	Long loading time
Naver	N.a.	Mainly used in Korea
CartoDB	Integration with CRM	Only 14 day free trial

Table B.2: Overview of popular Map APIs

B.7.4. BACK-END

The back-end of the web application is responsible for storing and retrieving information from the database, in order to manage the scenarios, DRIPs and services. The existing back-end code is written in NodeJS, it has a few open web endpoints accessible for everyone. The current back-end is not easily extendable with features which are not web development related. The new model lets the back-end handle calculations as a finite state machine. With these requirements two options remain, Python and Java. The reason Python is used in the project is that firstly, it has a steeper learning curve than Java. Secondly, Python requires less lines of code for the same functionality. This results in increased readability in the software. And thirdly, the client wants to extend this project with machine learning in the future. Python provides more and better options for machine learning than Java.

DATABASE

The first step in choosing a database is deciding whether to use a relational or non-relational database as well as which database model is most applicable. The advantages and disadvantages of both of database architectures will be further elaborated on [16] [17] [18]. Ultimately, the choice of architecture is substantiated.

Relational databases

A relational database uses tables (relations) and rows to store data. It links information between different tables via foreign keys (indexes). Relational databases are also known as Structured Query Language (SQL) databases, since they use SQL to query and manage the database. Relational databases are predominantly used if the structure of the data is not changing and when complicated querying is needed. Table B.3 gives an overview of popular relational databases.

Database	Pros	Cons
MySQL	Multitude of functionalities; Compatible with other databases; Robust	Time consuming; No XML support
MariaDB	Fast and stable; Customizable; Encryption at network/server/application level	Fairly new
SQLite3	Easy setup and configuration; File based system; Great for development and testing	Serverless; Not built for large-scale applications
Microsoft SQL server	Fast and stable; Adjustable and trackable performance levels	Pricey
Oracle Database	Latest innovations and features; Robust; Can handle enormous database and has a variety of features	Pricey
PostgreSQL	Scalable; JSON support	Configuration
IBM Db2	For enormous databases	Pricey

Table B.3: Overview of popular relational databases

Non-relational databases
In contrast to a relational database, a non-relational (or NoSQL) database can store unstructured data instead of using tables. Non-relational databases is often used if large amounts of data with little to no structure needs to be stored. Table B.4 gives an overview of popular non-relational databases.

Database	Pros	Cons
MongoDB	Fast and easy to use; JSON support	Default settings not secure; Setup can take a lot of time
CouchDB	Easy to use; JSON support; Flexible data structure	Arbitrary queries are expensive

Table B.4: Overview of popular non-relational databases

Database choice
The database that is currently used in the back-end is MongoDB. MongoDB is a non-relational document-based database. Table B.5 shows how scenarios are currently stored. It is decided to switch to a relational database, since the data of a tree structure has increased maintainability using a relational database. Furthermore, modifying table entries and querying will take less effort. However, it is important to clearly define the structure of the database beforehand. The relational database that will be used for the

system is MySQL, since it has the functionalities that are needed and it is robust. Additionally, all team members have some experience with MySQL.

Field	Type	Description
_id	String	The generated ID of the scenario.
name	String	The given name of the scenario.
folderId	Mixed	The ID of the folder the scenario is located in. *null* if the scenario is located in the root directory.
createdAt	Date	The date the scenario is created.
updatedAt	Date	The date the scenario is last updated.
dtJson	String	The JSON string representing the scenario decision tree.

Table B.5

B.7.5. COMMUNICATION BETWEEN CLIENT AND SERVER

The isolated back-end and front-end will be connected by the GraphQL[9] architure. To make this the decision the following architectures were considered; REST[10], Falcor[11] and GraphQL. The RESTful interface is the traditional way of API development. This architecture fetches data by accessing multiple endpoints, this causes a lot of over- and underfetching (fetching too much or too little data) [3]. Therefore, both GraphQL and Falcor have been developed to solve this problem as web applications are becoming increasingly more complex and the data-need in these applications larger. These architecture only perform a single POST query in which is specified what data is needed. The advantage of GraphQL over Falcor are that GraphQL can be easily integrated in a Python back-end and a React front-end, has good developer tools, is well documented and is more clear and strict in which data is requested and returned.

B.7.6. CODE QUALITY, CONTINUOUS INTEGRATION

To maintain the highest code quality as possible, continuous integration (CI) is used. [19] describes CI as a software development practice, where code checks are ran automatically after every code push to the repository. The GitHub Actions module [12] for CI is used, as it is free to use and has the same functionalities as other CI tools.

B.7.7. TESTING FRAMEWORKS

On of the reasons to abandon the original codebase is the lack of test code. There are 0 lines of test code in the 15.000 lines of code in the current prototype (B.4.3).

To test the React front-end, the Jest Testing Framework[13] and the React Testing Library[14] can be considered to use as recommended by [20]. The React Testing Library is

[9]https://graphql.org/
[10]https://restfulapi.net/
[11]https://netflix.github.io/falcor/
[12]https://github.com/features/actions
[13]https://jestjs.io/
[14]https://testing-library.com/docs/react-testing-library/intro

a lightweight and straightforward testing library for React components which relies less on the implementation details of the components than Jest. Jest is a more extensive library and has the ability to mock. Therefore, Jest is used in this project as well as user integration testing.

The Django back-end will be tested by using a unittest[15] test suite, as recommended by the Django framework [21]. It is possible to use other libraries but there is no reason to ignore the recommendation.

[15]https://docs.python.org/3/library/unittest.html

C

Figures

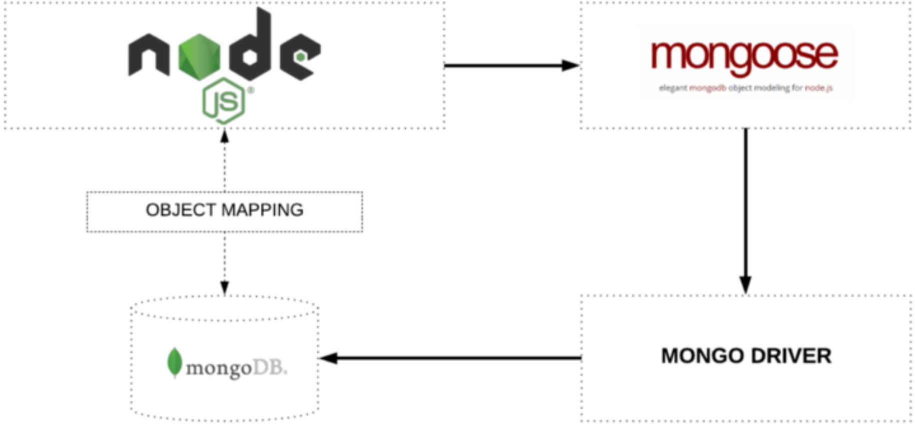


Figure C.1

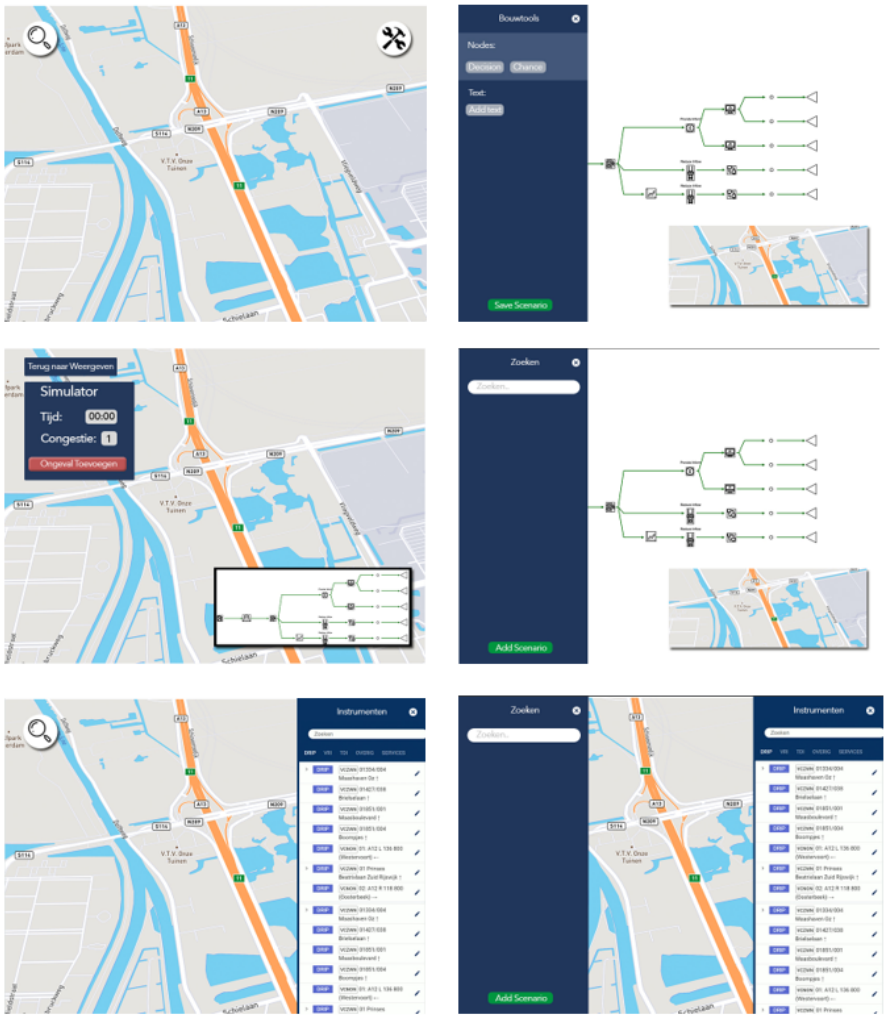
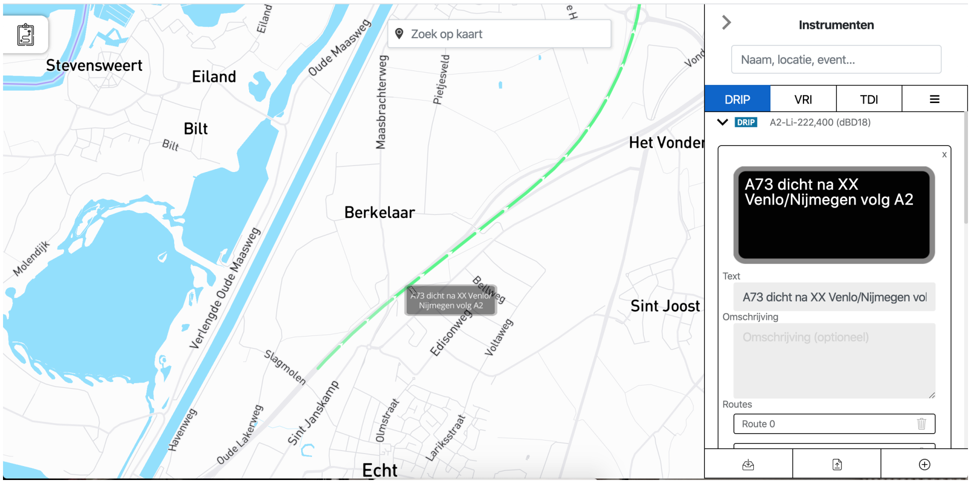


Figure C.2: States proposed in the research phase.



DRIP A2-Li-222,400 (dBD18)

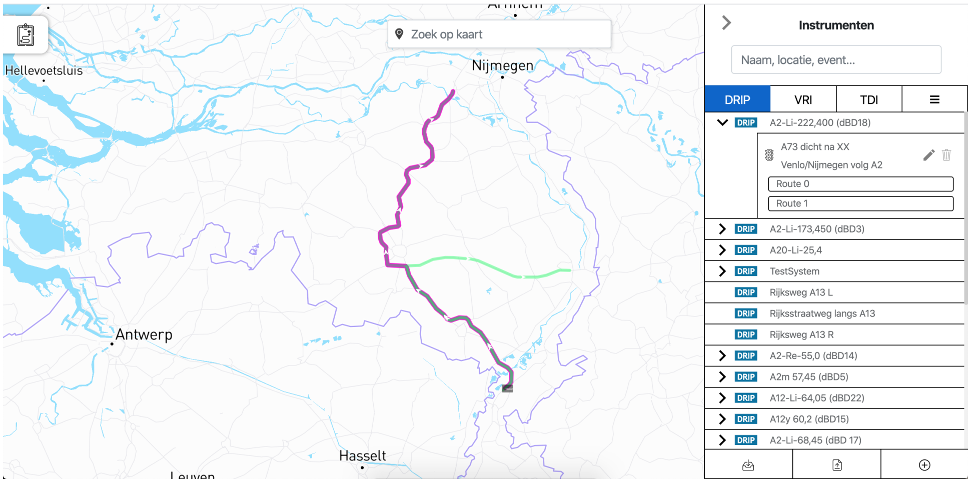
A73 dicht na XX Venlo/Nijmegen volg A2

Text: A73 dicht na XX Venlo/Nijmegen vol

Omschrijving

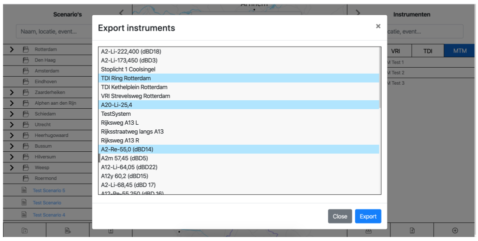
Routes

Figure C.3: Updating the *InstrumentAction* type



DRIP A2-Li-222,400 (dBD18)
A73 dicht na XX Venlo/Nijmegen volg A2
Route 0
Route 1
DRIP A2-Li-173,450 (dBD3)
DRIP A20-Li-25,4
DRIP TestSystem
DRIP Rijksweg A13 L
DRIP Rijksstraatweg langs A13
DRIP Rijksweg A13 R
DRIP A2-Re-55,0 (dBD14)
DRIP A2m 57,45 (dBD5)
DRIP A12-Li-64,05 (dBD22)
DRIP A12y 60,2 (dBD15)
DRIP A2-Li-68,45 (dBD 17)

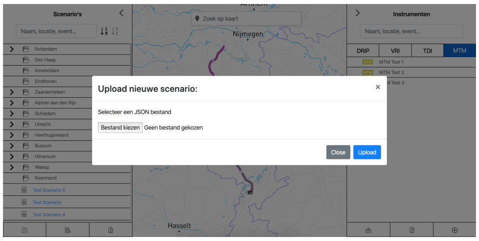
Figure C.4: Displaying an instrument on the map.



Export instruments

A2-Li-222,400 (dBD18)
A2-Li-173,450 (dBD3)
Stoplicht 1 Coolsingel
TDI Ring Rotterdam
TDI Kethelplein Rotterdam
VRI Strevelsweg Rotterdam
A20-Li-25,4
TestSystem
Rijksweg A13 L
Rijksstraatweg langs A13
Rijksweg A13 R
A2-Re-55,0 (dBD14)
A2m 57,45 (dBD5)
A12-Li-64,05 (dBD22)
A12y 60,2 (dBD15)
A2-Li-68,45 (dBD 17)

(a) Exporting instruments.



MTM Test 1
MTM Test 2
MTM Test 3

Upload nieuwe scenario:

Selecteer een JSON bestand

Geen bestand gekozen

(b) Uploading scenario.

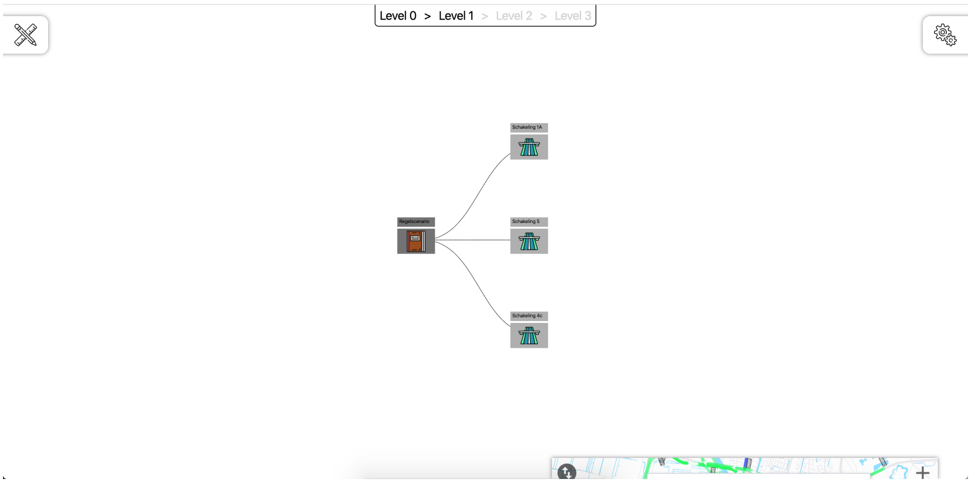
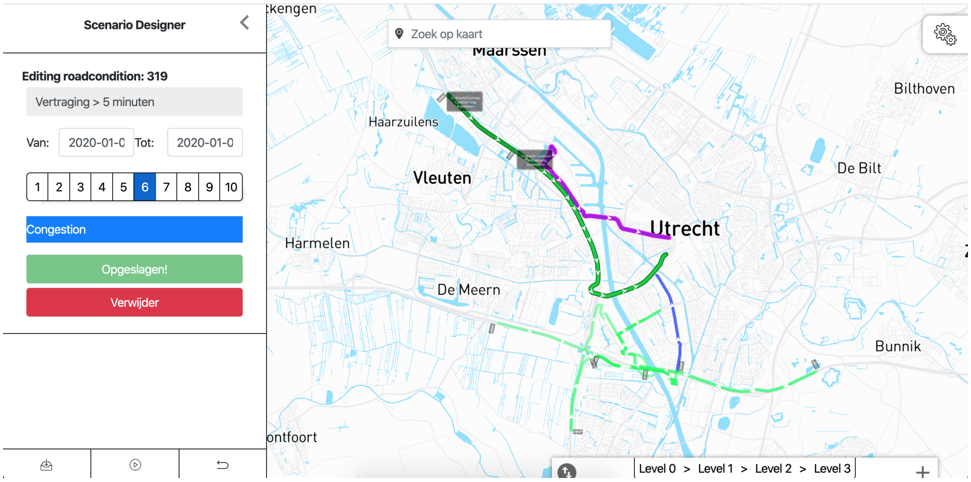


Figure C.6: Collapse the decision tree per level.



Scenario Designer

Editing roadcondition: 319

Vertraging > 5 minuten

Van: 2020-01-0 Tot: 2020-01-0

1 2 3 4 5 [✓ 6] 7 8 9 10

Congestion

Opgeslagen!

Verwijder

Figure C.7: Editing a *RoadCondition* type node.

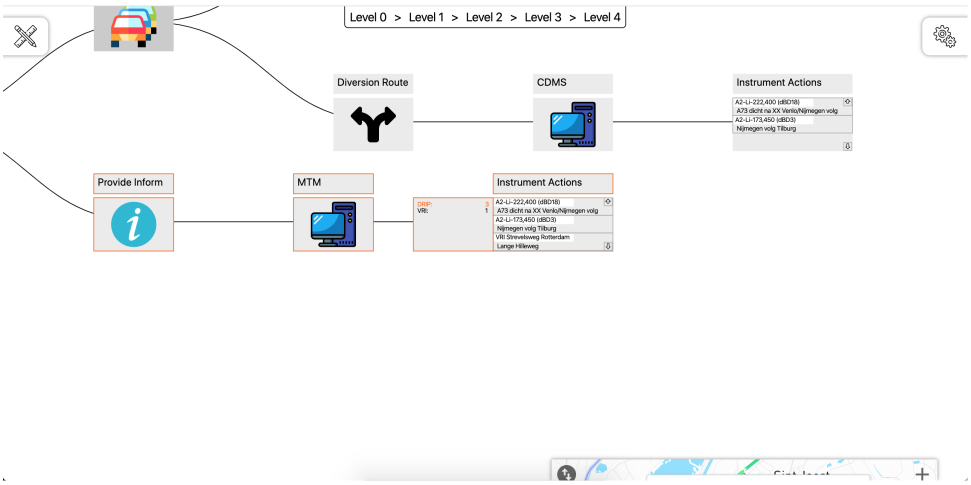


Figure C.8: Instrument action hover.

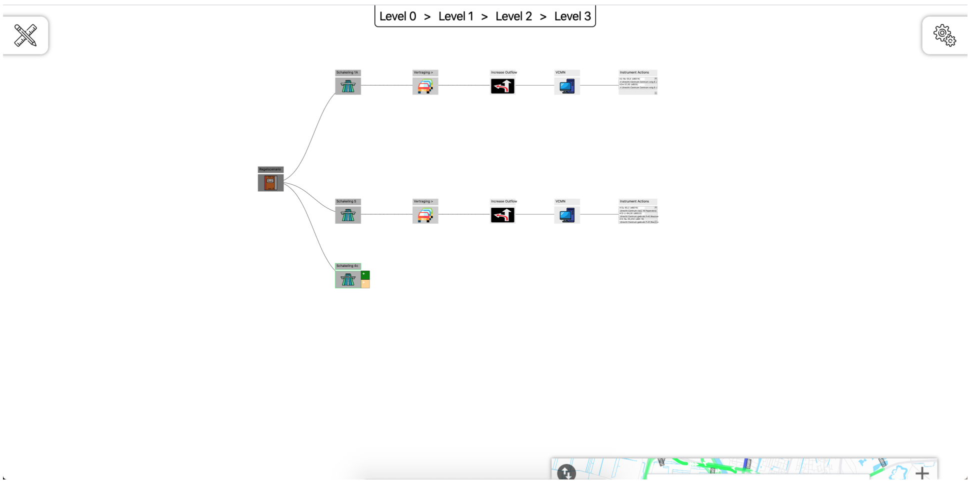


Figure C.9: Minimizing a branch of the decision tree.

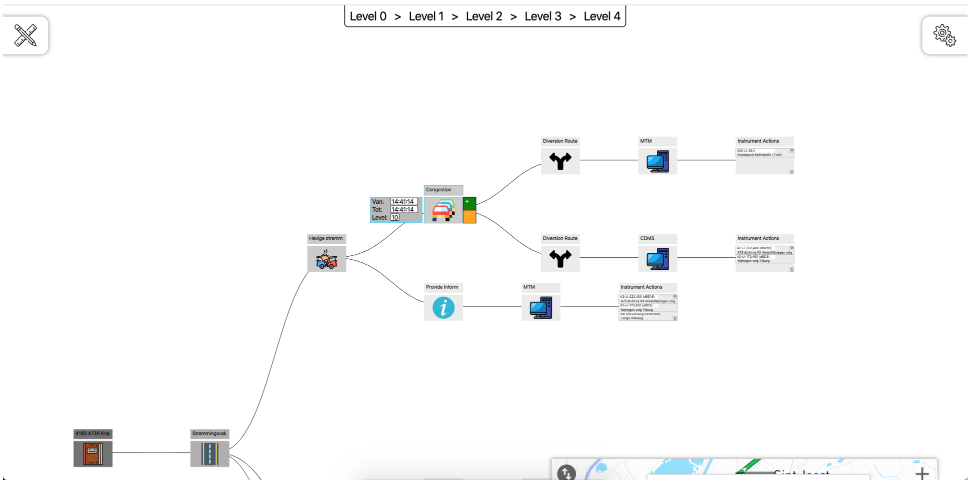
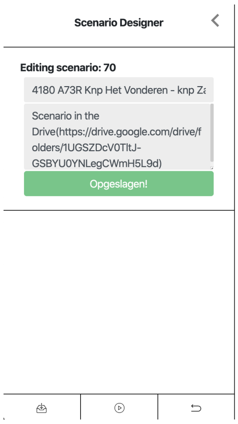


Figure C.10: Hovering *RoadCondition* type of node



Scenario Designer

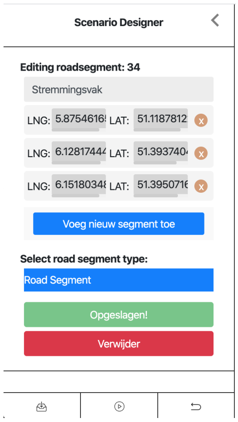
Editing scenario: 70

4180 A73R Knp Het Vonderen - knp Zε

Scenario in the Drive(https://drive.google.com/drive/folders/1UGSZDcV0TltJ-GSBYU0YNLegCWmH5L9d)

Opgeslagen!

(a) Toolbox of type *Scenario*



Scenario Designer

Editing roadsegment: 34

Stremmingsvak

LNG:	5.8754616	LAT:	51.1187812	x
LNG:	6.1281744	LAT:	51.393740	x
LNG:	6.1518034	LAT:	51.395071	x

Voeg nieuw segment toe

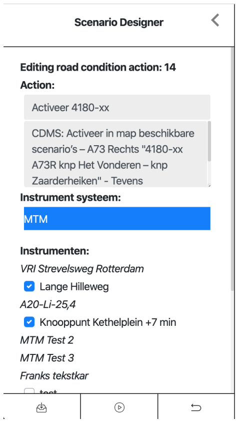
Select road segment type:

Road Segment

Opgeslagen!

Verwijder

(b) Toolbox of type *RoadSegment*



Scenario Designer

Editing road condition action: 14

Action:

Activeer 4180-xx

CDMS: Activeer in map beschikbare scenario's – A73 Rechts "4180-xx A73R knp Het Vonderen – knp Zaarderheiken" - Tevens

Instrument systeem:

MTM

Instrumenten:

VRI Strevelsweg Rotterdam

☑ Lange Hilleweg

A20-Li-25,4

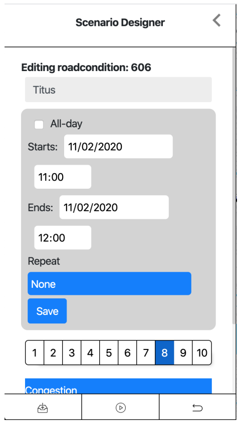
☑ Knooppunt Kethelplein +7 min

MTM Test 2

MTM Test 3

Franks tekstkar

(c) Toolbox of type *RoadConditionAction*



Scenario Designer

Editing roadcondition: 606

Titus

☐ All-day

Starts: 11/02/2020

11:00

Ends: 11/02/2020

12:00

Repeat

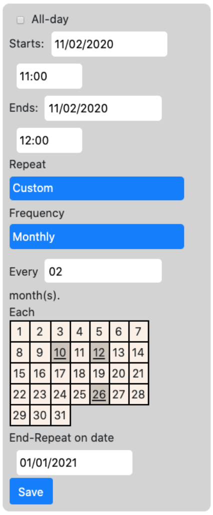
None

Save

1 2 3 4 5 6 7 [8] 9 10

Congestion

(d) Toolbox of type *RoadCondition*



All-day

Starts: 11/02/2020

11:00

Ends: 11/02/2020

12:00

Repeat

Custom

Frequency

Monthly

Every 02 month(s).

Each

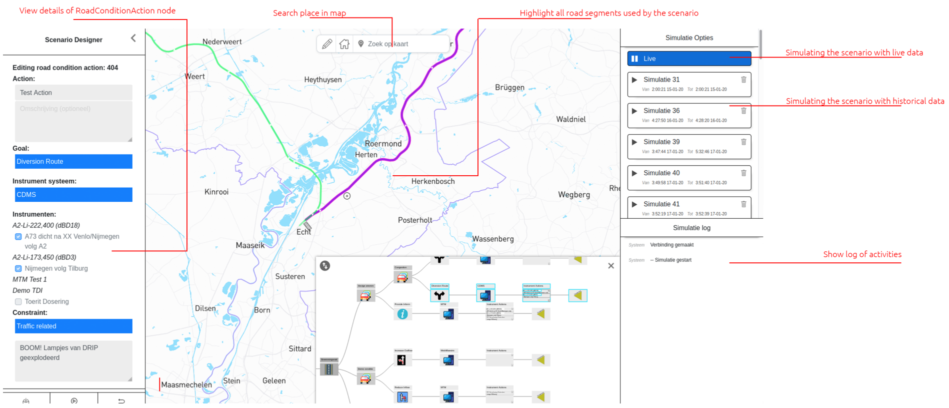
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

End-Repeat on date

01/01/2021

Save

Figure C.12: Time input in RoadCondition toolbox.



Scenario Designer

Editing road condition action: 404

Action:

Test Action

Omschrijving (optioneel)

Goal:

Diversion Route

Instrument systeem:

CDMS

Instrumenten:

A2-Li-222,400 (dBD18)

[✓] A73 dicht na XX Venlo/Nijmegen volg A2

A2-Li-173,450 (dBD3)

[✓] Nijmegen volg Tilburg

MTM Test 1

Demo TDI

[ ] Toerit Dosering

Constraint:

Traffic related

BOOM! Lampjes van DRIP geexplodeerd

Simulatie Opties

Live

Simulatie 31
Van 2:00:21 15-01-20 Tot 2:00:21 15-01-20

Simulatie 36
Van 4:27:50 16-01-20 Tot 4:28:20 16-01-20

Simulatie 39
Van 3:47:44 17-01-20 Tot 5:32:46 17-01-20

Simulatie 40
Van 3:49:58 17-01-20 Tot 3:51:40 17-01-20

Simulatie 41
Van 3:52:19 17-01-20 Tot 3:52:39 17-01-20

Simulatie log

Systeem Verbinding gemaakt

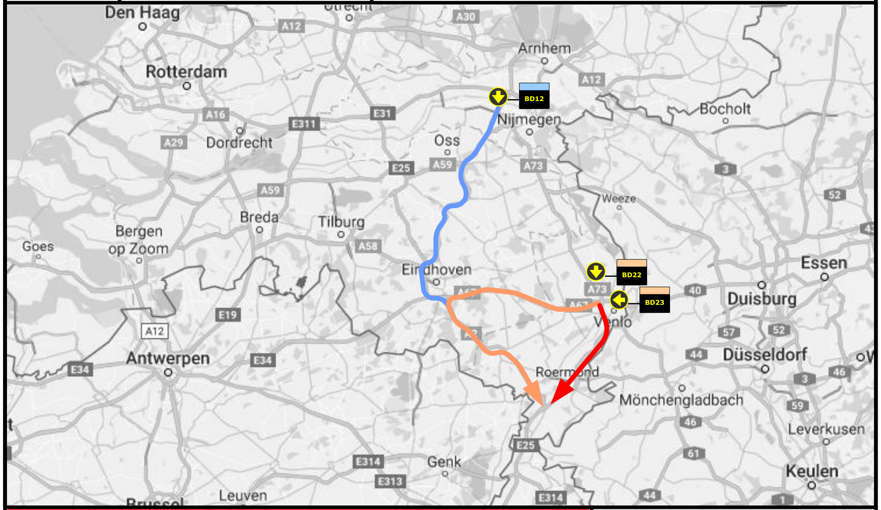
Systeem -- Simulatie gestart

Figure C.13: Live simulation.

D
Response plans

In this section examples of currently used response plans can be found.

4181	A73L	Knp Zaarderheiken → knp Tiglia



Werkwijze:
- Neem contact op met VCNL over de inzet van de GAR route 4181
- Na goedkeuring activeert VCNON de DRIP
- CDMS: Activeer in map beschikbare scenario's – A73 Links
  "4181-xx A73L knp Zaarderheiken– knp Het Vonderen"
- Tevens beschikbare omleidingen: 4741-4751 (KAR)

🟥 →	Stremmingsvak		
🟧 →	Omleiding VCZN		
🟩 →	VCMN	🟦 →	VCNON
🟪 →	VCZWN	🟨 →	VCNWN
● ■	(berm)DRIP		
● ■	Tekstwagen		
U32	U-route		
■	Verkeersregelaar		
●	Klapbord		

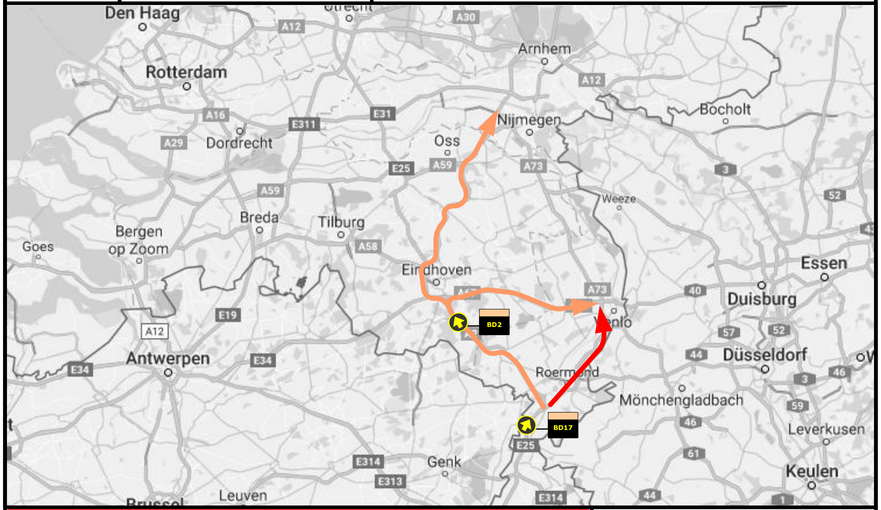
A67L 70,115 (dBD23)
A73 dicht na
XX
Maastricht volg
A67/A2

A73L 47,730 (dBD22)
A73 dicht na
XX
Maastricht volg
A67-Eindhoven

A50L hm 150,70 (dBD12)
A73 dicht na
Venlo
Maastricht volg
Eindhoven

	XX =
4181-01	Z'heiken
4181-02	Venlo-West
4181-03	Maasbree
4181-04	Venlo-Zuid
4181-05	Tiglia
4181-06	Belfeld
4181-07	Beesel
4181-08	Roermond
4181-09	Roermond-O
4181-10	Linne
4181-11	Maasbracht

4180	A73R	Knp Het Vonderen → knp Zaarderheiken



Werkwijze:
- CDMS: Activeer in map beschikbare scenario’s – A73 Rechts
  “4180-xx A73R knp Het Vonderen – knp Zaarderheiken”
- Tevens beschikbare omleidingen: 4726-4750 (KAR)

🟥	Stremmingsvak		
🟧	Omleiding VCZN		
🟩	VCMN	🟦	VCNON
🟪	VCZWN	🟨	VCNWN
	(berm)DRIP		
	Tekstwagen		
U32	U-route		
	Verkeersregelaar		
●	Klapbord		

A2L hm 222,400 (dBD17)

A73 dicht na
XX
Venlo/Nijmegen
volg A2

A2L hm 173,450 (dBD2)

Nijmegen
volg Tilburg

	XX =
4180-01	Het Vonderen
4180-02	Linne
4180-03	Roermond-O
4180-04	Roermond
4180-05	Beesel
4180-06	Belfeld
4180-07	Tiglia
4180-08	Venlo-Zuid
4180-09	Hout-Blerick
4180-10	Maasbree
4180-11	Venlo-West

E
INFO SHEET

The info sheet can be found on the next page.

Title of the project: Smart Traffic Management System
Name of the client organization: ScenWise B.V.
Date of the final presentation: 11-02-2020

Description We have created a web-application for traffic operators. In this application, a traffic operator can transform a response plan into a decision tree. This decision tree describes which actions an operator should take in order to manage different traffic situations. Besides the visualization of the plan, its components are visualized on a map. Furthermore, a traffic operator can simulate the plan on live and historical data to test whether the response plan works as planned.
Challenge The main challenge was creating an architecture with the ability to implement the broad spectrum of functionalities in a clear and intuitive manner, as well as modelling the decision tree on the back-end.
Research During the research phase, the main focus was to get the requirements clear from the client. Additionally, research in the current technologies was conducted.
Process The scrum methodology was used during the project, including daily stand-ups and weekly sprints and sprint evaluations. The team was divided into a back-end and front-end team. Each member also had its own role in the team. The client had various changes regarding the requirements, as well as additional requirements. After reasonable deliberation we agreed on which requirements had a higher priority.
Product The product consists of a React front-end desktop web application, using a back-end server with a GraphQL. Unit tests were written for the Python back-end, having an 89 % of branch coverage. Regarding the requirements, all of the must-haves and should-haves have been implemented.
Outlook The client is planning to extend the product in order to bring it to production. Recommendations consist of adding different roles for users, as well as modelling minor parts, such as a sort functionality.

Team Roles

Name	Interests	Contributions
Frank Bredius	Design, entrepreneurship, front-end development	Front-end; map interactivity, decision tree construction
Titus Naber	Data Science	Front-end; decision tree construction, decision tree simulation.
Leroy Velzel	Embedded Systems	Back-end; modeling decision tree, decision tree simulation, import and export.
Bailey Tjiong	Back-end development, computational intelligence	Back-end; modeling decision tree, testing. Front-end; import and export.

All team members contributed to the research report, final report as well as the final presentation.

Client Kin Fai Chan, ScenWise B.V.
Coach Asterios Katsifodimos, Assistant Professor EEMCS/EWI, TU Delft

Contacts
Frank Bredius: frankbredius@hotmail.com
Bailey Tjiong: btjiong@outlook.com

The final report for this project can be found at The TU Delft Repository

F
Project Description in BEPSys

F.1. Company Background

Scenwise B.V. is company with a lots of experience in traffic management & smart mobility domain. We work together with our partners to develop innovative software for the domains:

1. traffic management (e.g. automatic incident detection, response plans, etc.)
2. data science & visualisation. (e.g. traffic monitoring, data fusion, Big Data) Our customers are the Dutch Highway Agency (Rijkswaterstaat), Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, ITS system suppliers, Feyenoord stadium and also the city of Edmonton in Canada.

F.2. Development project(s) for the students

F.2.1. Project description

Scenwise has developed the ScenarioDesigner to support road authorities in creating response plans. The response plans describe which actions the operator should take in order to manage different traffic situations such as accidents, large scale events. The first protoype is an offline application with some limitations, e.g. no real-time traffic data can be used within the current software and there is no mechanisms to automatically trigger a plan. At the same time, the demand for response plans with automatic triggers and seemless integration with in-car information grows. In order to meet the future needs of our customers, we plan to start the development of a Decision Support Application using a new approach to incorporate different new and state-of-the art technologies. This project is to develop an smart Traffic management system to help traffc operators to make fast and safe decisions. Within this project, a team of students will work together to conduct research, make design decisions, develop the application inclusive testing. The application will incorporate a decision tree or a rule engine approach. The backend should be a big data platform using different open data (both real-time and historical).

The front-end should be web-based using modern graphical interfaces. This should include a sophisticated decision tree editor which interacts with a 3D map supported with real-time graphics to monitor and interact with the traffic conditions. The development of a simulator to trigger the different traffic situations is also a part of the project. This is an innovative project with a lots of technical challenges.

We are looking for enthousiastic students with skills and interest in Geographic Information Systems (GIS), Artificial Intelligence (AI), visualization techniques and algoritmes who want to take the challenge to develop the next generation of decision support application.

We intend to put this application into operation to support our customers. Our business partner in Canada also intend to integrate the functionality into their ITS (Intelligent Transport System) platform for their customers in North America and Asia.

G
SIG Feedback

In this section the feedback given by the SIG are noted. Originally, the feedback was written in Dutch.

G.1. Feedback SIG Week 6

The code of the system receives 3.3 stars from our maintainability model. This means that the code is market average maintainable. We see Unit Interfacing and Unit complexity, due to the lower partial scores, as possible points of improvement.

With regards to Unit Interfacing, the percentage of code in units with an above average parameters are being inspected. Usually an above average number of parameters indicates a lack of abstraction. Furthermore, a large number of parameters could lead to confusion in calling a method. In most of the cases it leads to longer and more complex methods as well. This can be solved by introducing parameter-objects, in which some logically connected parameters are transferred into one new object. This is also the case for constructors with a large number of parameters, which could be the reason to split up the data structure into several data structures. E.g. when a constructor has eight parameters which could be divided into two logically groups of four parameters, it is logical to introduce two new objects.

Examples in your project:

1. UpdateRoadCondition.mutate
2. Query.resolve_instruments
3. Query.resolve_scenarios

With regards to Unit Complexity, the percentage code that has an above average complexity is being inspected. This does not necessarily mean that the functionality itself is complex: often this kind of complexity arises when a method contains to many responsibilities, or due to an unnecessary complex implementation of the logic. Splitting these methods into smaller parts makes sure that each component is easier to comprehend, easier to test, and thus easier to maintain. By placing the functionalities into separate methods with a descriptive name, each of the component can be tested easily, and the overall flow of the method is easier to understand. With large and complex methods this can be done by dividing the problem, that the method is trying to solve, into smaller sub-problems, giving each sub-problem an own method. Consequently, the original method can call these new methods, combine the outcomes into the final result.

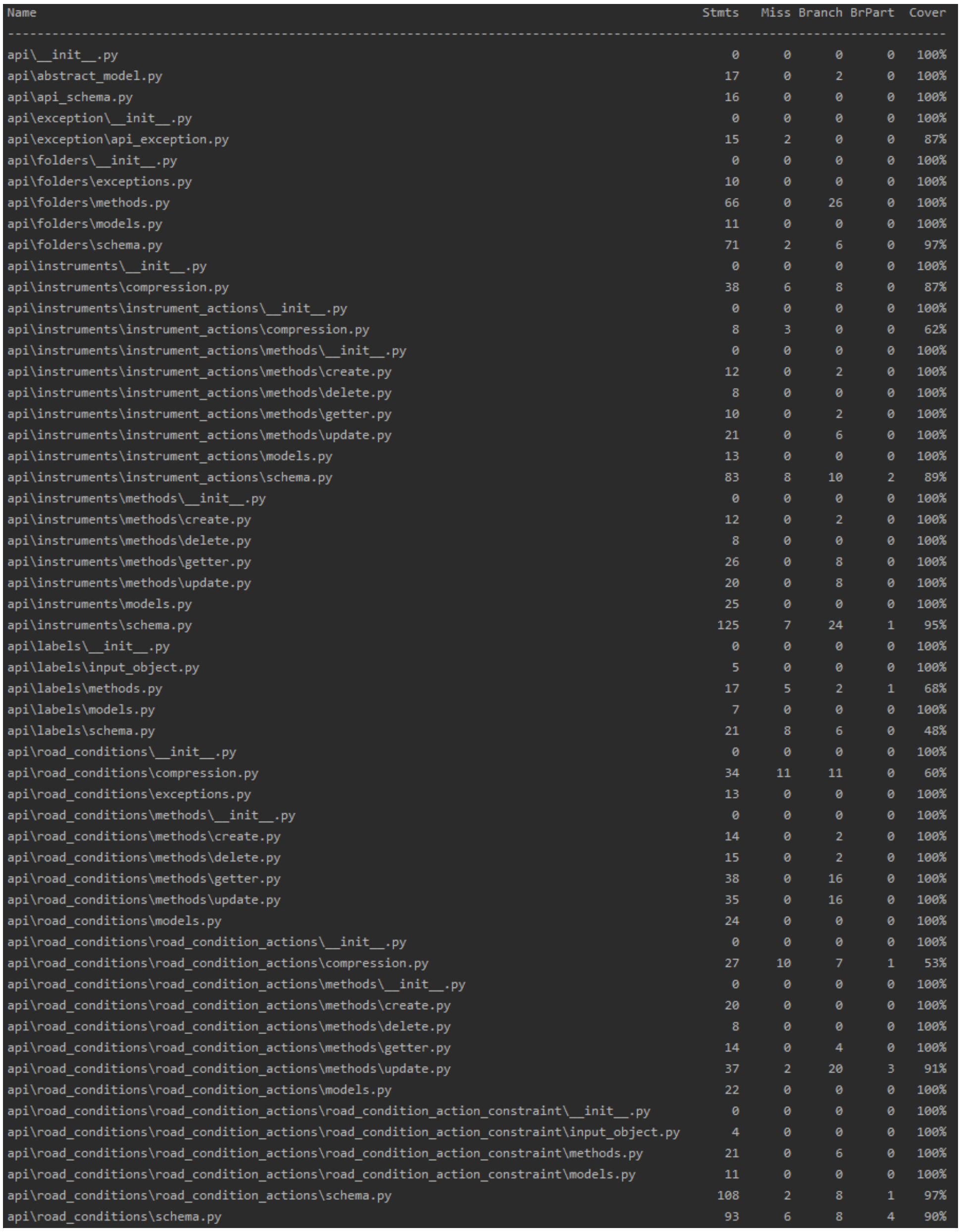
Examples in your project:

1. UpdateRoadConditionAction.mutate
2. model.py:create_measurement_site
3. trafficspeed.py:handle_measurement

The presence of test code is promising. The amount of tests however, is little compared to the amount of production code, hopefully it is possible to increase the amount of tests during the continuation of the project. In the long term, the presence of unit tests increases the flexibility of the code, since adjustments can be done without harming the stability.

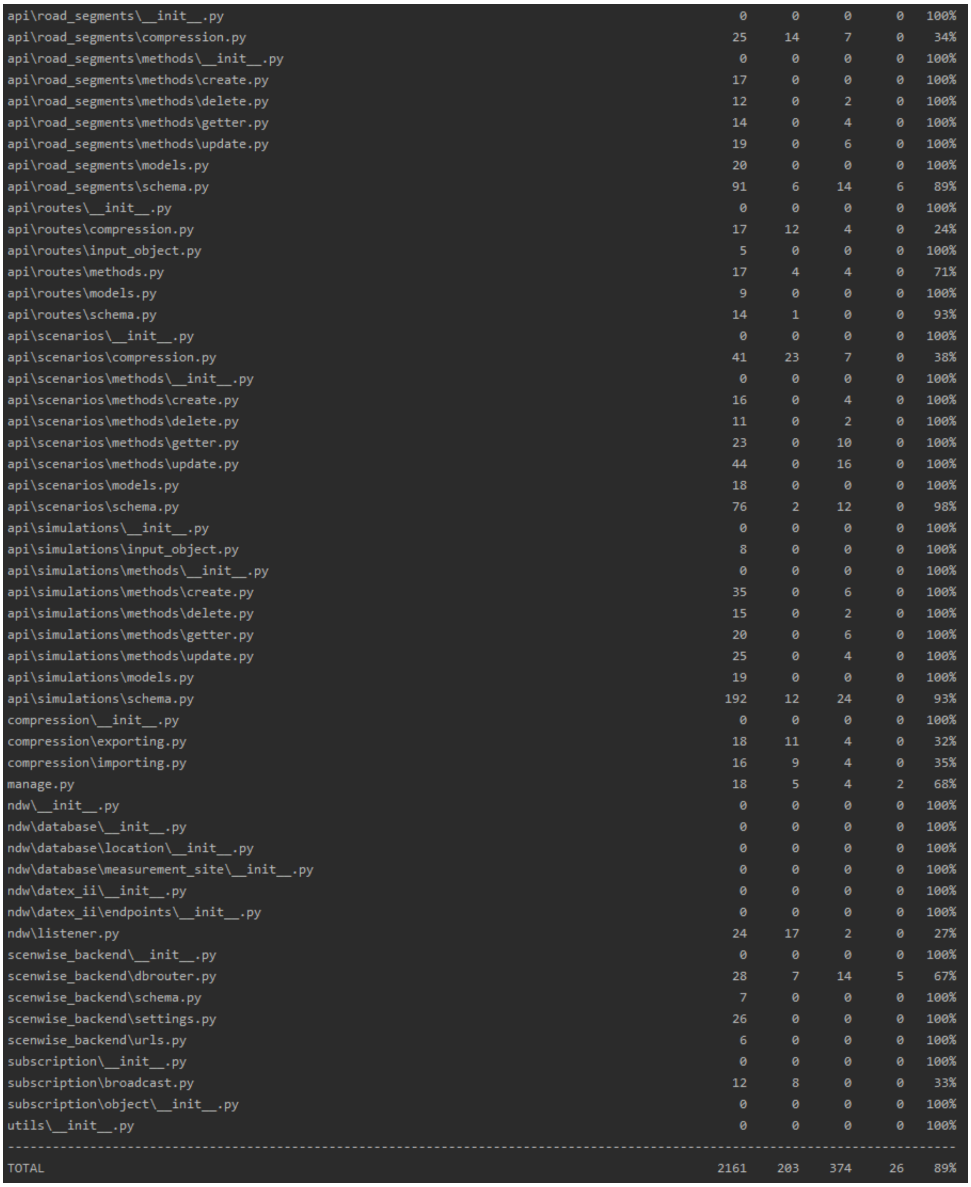
Generally, there are some improvements possible. Hopefully these improvements can be realised during the remaining time of the development phase.

H
Test coverage



```
Name                                                                                          Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------------------------------------------------------------
api\__init__.py                                                                                   0      0      0      0   100%
api\abstract_model.py                                                                            17      0      2      0   100%
api\api_schema.py                                                                                16      0      0      0   100%
api\exception\__init__.py                                                                         0      0      0      0   100%
api\exception\api_exception.py                                                                   15      2      0      0    87%
api\folders\__init__.py                                                                           0      0      0      0   100%
api\folders\exceptions.py                                                                        10      0      0      0   100%
api\folders\methods.py                                                                           66      0     26      0   100%
api\folders\models.py                                                                            11      0      0      0   100%
api\folders\schema.py                                                                            71      2      6      0    97%
api\instruments\__init__.py                                                                       0      0      0      0   100%
api\instruments\compression.py                                                                   38      6      8      0    87%
api\instruments\instrument_actions\__init__.py                                                    0      0      0      0   100%
api\instruments\instrument_actions\compression.py                                                 8      3      0      0    62%
api\instruments\instrument_actions\methods\__init__.py                                            0      0      0      0   100%
api\instruments\instrument_actions\methods\create.py                                             12      0      2      0   100%
api\instruments\instrument_actions\methods\delete.py                                              8      0      0      0   100%
api\instruments\instrument_actions\methods\getter.py                                             10      0      2      0   100%
api\instruments\instrument_actions\methods\update.py                                             21      0      6      0   100%
api\instruments\instrument_actions\models.py                                                     13      0      0      0   100%
api\instruments\instrument_actions\schema.py                                                     83      8     10      2    89%
api\instruments\methods\__init__.py                                                               0      0      0      0   100%
api\instruments\methods\create.py                                                                12      0      2      0   100%
api\instruments\methods\delete.py                                                                 8      0      0      0   100%
api\instruments\methods\getter.py                                                                26      0      8      0   100%
api\instruments\methods\update.py                                                                20      0      8      0   100%
api\instruments\models.py                                                                        25      0      0      0   100%
api\instruments\schema.py                                                                       125      7     24      1    95%
api\labels\__init__.py                                                                            0      0      0      0   100%
api\labels\input_object.py                                                                        5      0      0      0   100%
api\labels\methods.py                                                                            17      5      2      1    68%
api\labels\models.py                                                                              7      0      0      0   100%
api\labels\schema.py                                                                             21      8      6      0    48%
api\road_conditions\__init__.py                                                                   0      0      0      0   100%
api\road_conditions\compression.py                                                               34     11     11      0    60%
api\road_conditions\exceptions.py                                                                13      0      0      0   100%
api\road_conditions\methods\__init__.py                                                           0      0      0      0   100%
api\road_conditions\methods\create.py                                                            14      0      2      0   100%
api\road_conditions\methods\delete.py                                                            15      0      2      0   100%
api\road_conditions\methods\getter.py                                                            38      0     16      0   100%
api\road_conditions\methods\update.py                                                            35      0     16      0   100%
api\road_conditions\models.py                                                                    24      0      0      0   100%
api\road_conditions\road_condition_actions\__init__.py                                            0      0      0      0   100%
api\road_conditions\road_condition_actions\compression.py                                        27     10      7      1    53%
api\road_conditions\road_condition_actions\methods\__init__.py                                    0      0      0      0   100%
api\road_conditions\road_condition_actions\methods\create.py                                     20      0      0      0   100%
api\road_conditions\road_condition_actions\methods\delete.py                                      8      0      0      0   100%
api\road_conditions\road_condition_actions\methods\getter.py                                     14      0      4      0   100%
api\road_conditions\road_condition_actions\methods\update.py                                     37      2     20      3    91%
api\road_conditions\road_condition_actions\models.py                                             22      0      0      0   100%
api\road_conditions\road_condition_actions\road_condition_action_constraint\__init__.py           0      0      0      0   100%
api\road_conditions\road_condition_actions\road_condition_action_constraint\input_object.py       4      0      0      0   100%
api\road_conditions\road_condition_actions\road_condition_action_constraint\methods.py           21      0      6      0   100%
api\road_conditions\road_condition_actions\road_condition_action_constraint\models.py            11      0      0      0   100%
api\road_conditions\road_condition_actions\schema.py                                            108      2      8      1    97%
api\road_conditions\schema.py                                                                    93      6      8      4    90%
```

Figure H.1



```
api\road_segments\__init__.py                             0      0      0      0   100%
api\road_segments\compression.py                         25     14      7      0    34%
api\road_segments\methods\__init__.py                     0      0      0      0   100%
api\road_segments\methods\create.py                      17      0      0      0   100%
api\road_segments\methods\delete.py                      12      0      2      0   100%
api\road_segments\methods\getter.py                      14      0      4      0   100%
api\road_segments\methods\update.py                      19      0      6      0   100%
api\road_segments\models.py                              20      0      0      0   100%
api\road_segments\schema.py                              91      6     14      6    89%
api\routes\__init__.py                                    0      0      0      0   100%
api\routes\compression.py                                17     12      4      0    24%
api\routes\input_object.py                                5      0      0      0   100%
api\routes\methods.py                                    17      4      4      0    71%
api\routes\models.py                                      9      0      0      0   100%
api\routes\schema.py                                     14      1      0      0    93%
api\scenarios\__init__.py                                 0      0      0      0   100%
api\scenarios\compression.py                             41     23      7      0    38%
api\scenarios\methods\__init__.py                         0      0      0      0   100%
api\scenarios\methods\create.py                          16      0      4      0   100%
api\scenarios\methods\delete.py                          11      0      2      0   100%
api\scenarios\methods\getter.py                          23      0     10      0   100%
api\scenarios\methods\update.py                          44      0     16      0   100%
api\scenarios\models.py                                  18      0      0      0   100%
api\scenarios\schema.py                                  76      2     12      0    98%
api\simulations\__init__.py                               0      0      0      0   100%
api\simulations\input_object.py                           8      0      0      0   100%
api\simulations\methods\__init__.py                       0      0      0      0   100%
api\simulations\methods\create.py                        35      0      6      0   100%
api\simulations\methods\delete.py                        15      0      2      0   100%
api\simulations\methods\getter.py                        20      0      6      0   100%
api\simulations\methods\update.py                        25      0      4      0   100%
api\simulations\models.py                                19      0      0      0   100%
api\simulations\schema.py                               192     12     24      0    93%
compression\__init__.py                                   0      0      0      0   100%
compression\exporting.py                                 18     11      4      0    32%
compression\importing.py                                 16      9      4      0    35%
manage.py                                                18      5      4      2    68%
ndw\__init__.py                                           0      0      0      0   100%
ndw\database\__init__.py                                  0      0      0      0   100%
ndw\database\location\__init__.py                         0      0      0      0   100%
ndw\database\measurement_site\__init__.py                 0      0      0      0   100%
ndw\datex_ii\__init__.py                                  0      0      0      0   100%
ndw\datex_ii\endpoints\__init__.py                        0      0      0      0   100%
ndw\listener.py                                          24     17      2      0    27%
scenwise_backend\__init__.py                              0      0      0      0   100%
scenwise_backend\dbrouter.py                             28      7     14      5    67%
scenwise_backend\schema.py                                7      0      0      0   100%
scenwise_backend\settings.py                             26      0      0      0   100%
scenwise_backend\urls.py                                  6      0      0      0   100%
subscription\__init__.py                                  0      0      0      0   100%
subscription\broadcast.py                                12      8      0      0    33%
subscription\object\__init__.py                           0      0      0      0   100%
utils\__init__.py                                         0      0      0      0   100%
-----------------------------------------------------------------------------------------
TOTAL                                                  2161    203    374     26    89%
```

Figure H.2

I

Database EER Diagram

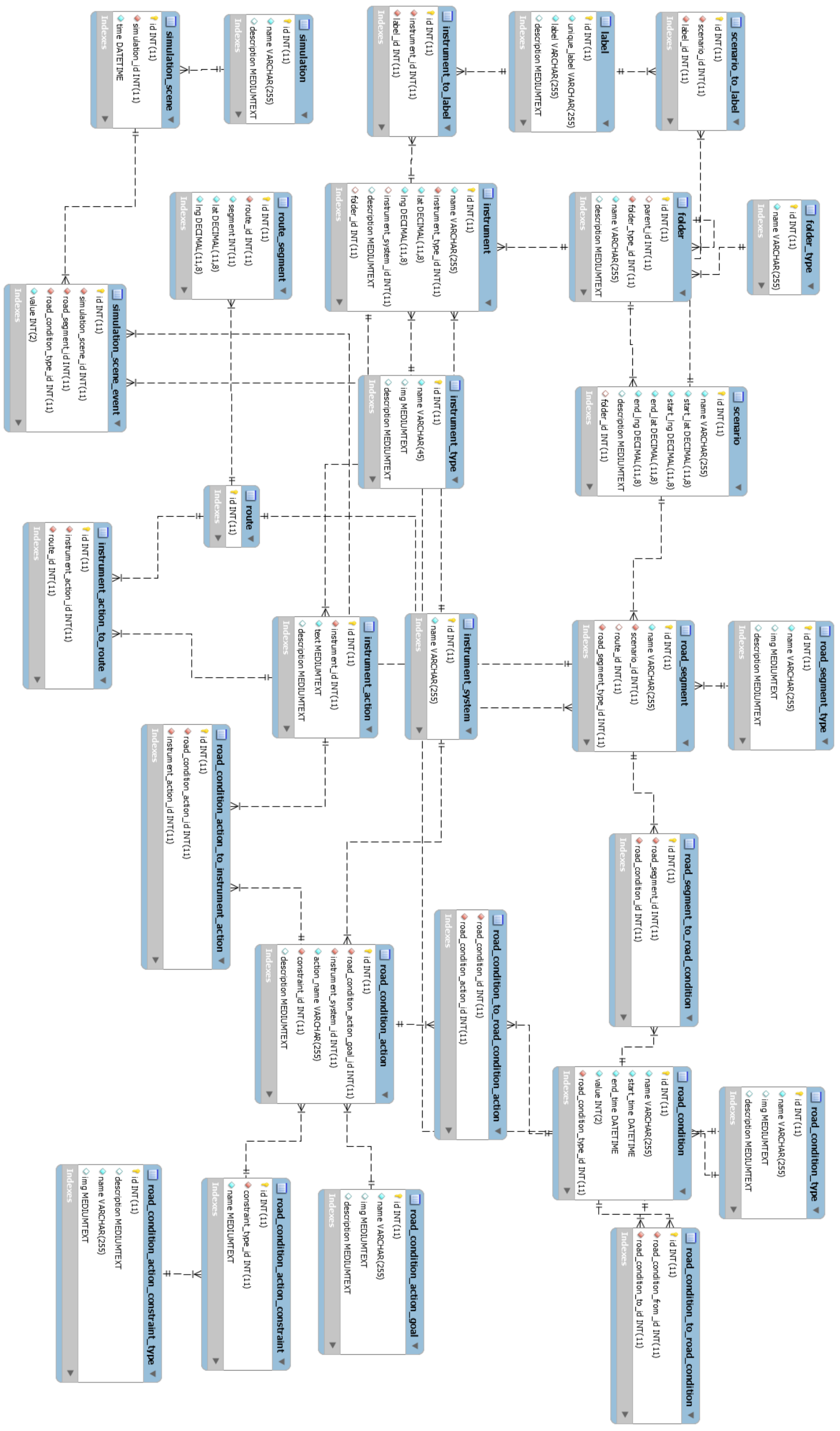

simulation_scene
id INT(11)
simulation_id INT(11)
time DATETIME
simulation
id INT(11)
name VARCHAR(255)
description MEDIUMTEXT
instrument_to_label
id INT(11)
instrument_id INT(11)
label_id INT(11)
label
id INT(11)
unique_label VARCHAR(255)
label VARCHAR(255)
description MEDIUMTEXT
scenario_to_label
id INT(11)
scenario_id INT(11)
label_id INT(11)
route_segment
id INT(11)
route_id INT(11)
segment INT(11)
lat DECIMAL(11,8)
lng DECIMAL(11,8)
instrument
id INT(11)
name VARCHAR(255)
instrument_type_id INT(11)
lat DECIMAL(11,8)
lng DECIMAL(11,8)
instrument_system_id INT(11)
description MEDIUMTEXT
folder_id INT(11)
folder
id INT(11)
parent_id INT(11)
folder_type_id INT(11)
name VARCHAR(255)
description MEDIUMTEXT
folder_type
id INT(11)
name VARCHAR(255)
simulation_scene_event
id INT(11)
simulation_scene_id INT(11)
road_segment_id INT(11)
road_condition_type_id INT(11)
value INT(2)
instrument_type
id INT(11)
name VARCHAR(45)
img MEDIUMTEXT
description MEDIUMTEXT
scenario
id INT(11)
name VARCHAR(255)
start_lat DECIMAL(11,8)
start_lng DECIMAL(11,8)
end_lat DECIMAL(11,8)
end_lng DECIMAL(11,8)
description MEDIUMTEXT
folder_id INT(11)
route
id INT(11)
instrument_action_to_route
id INT(11)
instrument_action_id INT(11)
route_id INT(11)
instrument_action
id INT(11)
instrument_id INT(11)
text MEDIUMTEXT
description MEDIUMTEXT
instrument_system
id INT(11)
name VARCHAR(255)
road_segment
id INT(11)
name VARCHAR(255)
scenario_id INT(11)
route_id INT(11)
road_segment_type_id INT(11)
road_segment_type
id INT(11)
name VARCHAR(255)
img MEDIUMTEXT
description MEDIUMTEXT
road_condition_action_to_instrument_action
id INT(11)
road_condition_action_id INT(11)
instrument_action_id INT(11)
road_segment_to_road_condition
id INT(11)
road_segment_id INT(11)
road_condition_id INT(11)
road_condition_to_road_condition_action
road_condition_id INT(11)
road_condition_action_id INT(11)
road_condition_action
id INT(11)
road_condition_action_goal_id INT(11)
instrument_system_id INT(11)
action_name VARCHAR(255)
constraint_id INT(11)
description MEDIUMTEXT
road_condition
id INT(11)
name VARCHAR(255)
start_time DATETIME
end_time DATETIME
value INT(2)
road_condition_type_id INT(11)
road_condition_type
id INT(11)
name VARCHAR(255)
img MEDIUMTEXT
description MEDIUMTEXT
road_condition_action_constraint_type
id INT(11)
description MEDIUMTEXT
name VARCHAR(255)
img MEDIUMTEXT
road_condition_action_constraint
id INT(11)
constraint_type_id INT(11)
name MEDIUMTEXT
road_condition_action_goal
id INT(11)
name VARCHAR(255)
img MEDIUMTEXT
description MEDIUMTEXT
road_condition_to_road_condition
id INT(11)
road_condition_from_id INT(11)
road_condition_to_id INT(11)
Indexes

Figure I.1