SmartRoads 2.0

Final Report

J.L. Buijnsters
D. Hofman
J.G.P. Klein Kranenbarg
C. El Moussaoui
K. Zheng

SmartRoads 2.0

Final Report

by

J.L. Buijnsters
D. Hofman
J.G.P. Klein Kranenbarg
C. el Moussaoui
K. Zheng

Project duration:	April 20, 2020 – July 1, 2020	
Supervisors:	Ir. K. F. Chan	Client
	Dr.ir. B.H.M. Gerritsen	Coach
	Dr.ir. H. Wang,	TU Delft, BEP Coordinator
	Ir. O.W. Visser,	TU Delft, BEP Coordinator

TU Delft

Preface

This report is meant to conclude ”TI3806: Bachelorproject” and was carried out by Jan Buijnsters, Daan Hofman, Jasper Klein Kranenbarg, Chakir el Moussaoui and Kawin Zheng.

Over the last 11 weeks, we have researched and developed a new version for SmartRoads 1.0. Although the work circumstances were different than usual because of the COVID-19 pandemic, were we still able to deliver a high-quality product.

We would first like to express our gratitude to Ir. K.F. Chan for providing us with a project and the proper guidance and support. Secondly, we would also like to thank Dr.ir. B.H.M. Gerritsen for the continuous support and without whom we would never have been able to deliver the final product.

J.L. Buijnsters
D. Hofman
J.G.P. Klein Kranenbarg
C. El Moussaoui
K. Zheng
Delft, June 2020

Summary

ScenWise is an innovative company that specializes in data science revolving around traffic management. ScenWise strives to use the newest and best technologies and practices when it comes to web applications, data science and traffic management. The reason for this is that they provide tools to analyse and visualise a variety of situations that occur in traffic management. One such tool is SmartRoads 1.0, which allows users to analyse traffic data and situations via a web application.

Unfortunately SmartRoads 1.0 does not perform as desired. Additionally, ScenWise itself has the problem of not being able to integrate previously made products by student groups into their own existing products. During the research aimed to resolve these problems another issue arose; the software development life cycle of ScenWise is very lacking. Research on the SmartRoads 1.0 performance problem showed that the bottleneck of its performance is due to the front-end.

The outdated SmartRoads 1.0 front-end was thus replaced with a new and better SmartRoads 2.0 front-end. The integration problem and development life cycle problem are both addressed in the Long-term evolution (LTE) design found in appendix I. This LTE design contains the architecture migration plan. This plan will transform the current software architecture to a Service-oriented architecture (SOA) providing a solution for the current integration problems. A result of the first steps of this architecture migration plan is the Application Programming Interface (API) Gateway, which has been implemented in the aforementioned SmartRoads 2.0. Next to the migration plan, guidelines for ScenWise to improve their software development life cycle are elaborated in the LTE design. In this report the identified problems, their solutions and executions are explained, discussed and evaluated.

Contents

1 Introduction **1**

2 Problem Analysis **2**
2.1 Problem definition . . . 2
2.2 Limitations to overcome . . . 4

3 Research **5**
3.1 Research Approach . . . 5
3.2 Methodology . . . 8
3.3 Results . . . 8
3.4 Data analysis . . . 9
3.4.1 API . . . 9
3.4.2 database . . . 9
3.4.3 Back end . . . 9
3.4.4 Front end . . . 10
3.5 Conclusion . . . 10
3.5.1 Requirements revised . . . 11
3.6 Success Criteria . . . 12

4 Design **13**
4.1 Long-term evolution design . . . 13
4.1.1 Architecture migration . . . 13
4.1.2 Software Development Lifecycle Guidelines . . . 13
4.1.3 Communication protocols . . . 13
4.1.4 Adding a new Service . . . 13
4.2 API Gateway . . . 14
4.3 Front end . . . 14
4.3.1 Framework . . . 14
4.3.2 Map . . . 14
4.3.3 support for mobile browsers . . . 15
4.4 Parser and Central Database . . . 15

5 Process **16**
5.1 Workflow . . . 16
5.1.1 Sprint meetings . . . 16
5.1.2 Daily scrum . . . 16
5.1.3 Gitlab workflow . . . 16
5.2 Internal communication . . . 17
5.2.1 Meetings . . . 17
5.2.2 Role division . . . 17
5.3 External communication . . . 17
5.3.1 Meetings client . . . 17
5.3.2 Meetings coach . . . 18

6 Code Quality and Testing **19**
6.1 Code quality . . . 19
6.1.1 Documentation . . . 19
6.1.2 Code size . . . 19
6.1.3 Front-end . . . 19
6.1.4 Back-end . . . 19
6.1.5 OpenAPI documentation . . . 19
6.1.6 Continuous Integration/Continuous Deployment . . . 20

6.2 Testing . . . 20
6.2.1 Unit tests . . . 20
6.2.2 Integration tests. . . . 21
6.2.3 System testing . . . 22
6.3 SIG Feedback . . . 22
6.3.1 Initial feedback . . . 22
6.3.2 Second feedback . . . 23

7 Deliverables 26
7.1 Features . . . 26
7.1.1 Front-end design . . . 26
7.1.2 Layers . . . 27
7.1.3 DRIP . . . 27
7.1.4 Speed Map . . . 28
7.1.5 MSI . . . 29
7.1.6 Status messages . . . 30
7.1.7 Measurement points . . . 31
7.1.8 Replay. . . . 31
7.1.9 3D view . . . 31
7.2 Long-term evolution design . . . 32
7.3 API Gateway . . . 32
7.4 Parser and Central Database . . . 33
7.5 Technical details . . . 33
7.5.1 React . . . 33
7.5.2 Graphs . . . 33
7.5.3 Replay. . . . 33
7.5.4 Docker . . . 33

8 Product evaluation 34
8.1 Implementation challenges. . . . 35
8.1.1 Unique project . . . 35
8.1.2 Mapbox . . . 35
8.1.3 Docker . . . 35
8.1.4 API Gateway . . . 35
8.2 Requirements assessment. . . . 35
8.2.1 Requirements to increase performance . . . 35
8.2.2 Requirements for new features . . . 38
8.2.3 Requirements for in-house development . . . 38
8.2.4 Requirements to increase configurability . . . 39

9 Project Evaluation 41
9.1 Workflow evaluation . . . 41
9.1.1 Inter group challenges . . . 41
9.2 Client satisfaction . . . 41
9.3 Course of the project . . . 42
9.4 Evaluation of Ethical implications . . . 42

10 Conclusion 43
10.1 Success criteria assessment. . . . 43
10.1.1 Documentation . . . 43
10.1.2 Tests . . . 43
10.1.3 Performance . . . 43
10.1.4 Modularity . . . 43
10.2 Conclusion . . . 44

11 Recommendations **45**
11.1 Requirements . . . 45
11.2 Long-term evolution design . . . 45
11.3 Improvements . . . 45
11.3.1 Moving conversions from the front end to the backend . . . 45
11.3.2 Separate data in the Analytics.SmartRoads backend . . . 45
11.3.3 Support iPhones . . . 46
11.4 Extra features . . . 46
11.4.1 New services . . . 46
11.4.2 User accounts . . . 46
11.4.3 Event-driven communication . . . 46
11.4.4 Server-sent events . . . 46
11.4.5 Kubernetes, Docker Swarm . . . 46

Bibliography **46**

Glossary **49**

Acronyms **51**

A Info Sheet **52**

B ProjectForum Description **54**

C Evaluation Success Criteria **56**

D MoSCoW Requirements **59**

E MoSCoW Evaluation **63**

F Course of events **67**

G Project Plan **68**

H Research Report **83**

I Long-term Evolution Design **131**

1 Introduction

Scenwise B.V.[1] is a company with experience in the data science and smart mobility domain. They work together with partners to develop software for the domain of Traffic management (e.g. automatic incident detection, response plans, etc.), and Data Science (e.g. traffic monitoring, data fusion, Big Data, Machine Learning). Their customers include: Rijkswaterstaat, Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, IT system suppliers, event organisers such as Feyenoord, and recently also the city of Edmonton in Canada.

SmartRoads 1.0 is one of the applications ScenWise has developed in recent years. However, with the ever-changing technology, it has become relatively slow and outdated. To get the application back on track with the newest and fastest technology, the idea of a new version, SmartRoads 2.0, was created. As the team did research, which can be found in appendix H, the discovery was made that there were also underlying problems within the development of ScenWise that had to be addressed as well.

In chapter 2 a complete analysis of the specifics regarding these initially found problems are elaborated. Chapter 3 discusses the research done to define the exact problems to solve. The design for solving these problems is laid out in chapter 4. The process of how the team worked on this project is explained in chapter 5. Chapter 6 discusses the team's definition of good code quality and testing, followed by the implementation of the product in chapter 7. An evaluation of the requirements set in appendix H and the end product is made in chapter 8, followed by an evaluation of the project in chapter 9. Finally, an overall conclusion of this project and the team's recommendations for the future of ScenWise are addressed in chapter 10 and 11 respectively.

[1] `https://www.scenwise.com/`

2 Problem Analysis

To kick-off the project a problem analysis is needed on the initially found problems. This will make it clear what the current limitations of ScenWise and SmartRoads 1.0 are. These limitations will provide as indications where the underlying problems may come from, these will need thorough research.

2.1. Problem definition

The performance of SmartRoads 1.0 has been deteriorating. SmartRoads 2.0 was originally intended as the new version of SmartRoads 1.0. This new version should resolve the currently existing performance issues and add more functionalities to support a wider range of customers. ScenWise indicated that the performance problems from the back end are causing an unusable replay and the disability to display information about the whole Netherlands. In appendix B the original project description can be found. Multiple similar web applications are currently being sold by Scenwise, with different features and functionalities depending on the demands of the customer.

A more detailed insight into the company was achieved based on the initial meeting with the client and subsequent meetings with both client and one of their developers. Their existing solutions share the same base functionality, yet all of them have been built from scratch. A reason for this is the multiple student software projects running with ScenWise. This is an inefficient approach to software development and indicates a lack of modularity in their software. The lack of modularity is supported by the fact that their code-base has little to no tests and is not properly documented. This causes integration problems making integrating parts of the software products delivered by previous student groups more costly than necessary. Making their code-base modular and thus easily reusable should be beneficial for ScenWise.

Based on these findings, a proposal was made by the group. This proposed to design and set up a modular architecture for the whole company in which existing functionalities will easily be reusable. Such architecture would add significant long term value for ScenWise, compared to adding just another non-integratable application to the arsenal of ScenWise. After discussing this proposal with ScenWise they saw the value of finding solutions for these problems. However, delivering the new version of SmartRoads 1.0, ready for production, would still be priority number one. Additionally, having this done in a way which solves the more fundamental software problems was a close second priority. This project is unique in the sense that the group analyzed the initial problems at ScenWise, and instead of starting to implement features as specified by the client, performed extensive research and proposed an approach that provided more value to ScenWise as a whole.

To get a better understanding of what fits within the scope of this project, a Strengths Weaknesses Opportunities and Threats analysis (SWOT-analysis) of Scenwise was made (see table 2.1). ScenWise has two functional back-ends with functionalities applicable to SmartRoads 2.0. They have a clear view of what features need to be implemented in the near future and lots of possible features for the future. However, there is room for improvement in the workflow at Scenwise. During the initial product review, the team found that SmartRoads 1.0 has some unintuitive elements in the front end design. Their current solution has a large technical debt, caused mostly by the lack of tests, the lack of documentation and the lack of a modular structure. This makes their applications difficult to maintain.

And the earlier mentioned performance issues result in an unusable replay and displaying of information for more than a few regions. These weaknesses will be resolved by creating a modular and easily maintainable structure. This will allow combining functionalities from various other back-ends created by ScenWise, which will open up new possibilities such as creating new tailored products for custom clients using the existing codebase. This maintainable approach will also improve the efficiency with which new and existing developers can work. Solving the performance issues will make SmartRoads interesting for customers anywhere located in the Netherlands. This could even create a snowball effect allowing ScenWise to extend to other countries as well. Some of the threats outside of the scope of the project would be the disability of ScenWise to follow-up on the proposed guidelines regarding software development lifecycles, like an improved workflow. Another threat would be an unsuccessful migration to SmartRoads 2.0. And at last, ScenWise has to keep in mind the threat that an enabling technology used for SmartRoads 2.0 may become unusable, for example, a large pricing change.

Internal Strengths:	Weaknesses:
Lots of functionalities available from the existing back-ends of SmartRoads 1.0 and Analytics.SmartRoads.	Workflow
ScenWise has a clear view of what features they want to add now and in the future	Code Quality → not modular, no extensive testing, missing documentations
	Periodical manual operations needed in order to keep the software up to date
	SmartRoads 1.0 performance issues: Replay Only able to display information about a small part of the roads of the Netherlands.
	Some unintuitive elements in front-end design
External Opportunities:	Threats:
Extending to nearby countries, by combining more data feeds	SmartRoads 2.0 will be just another Analytics.SmartRoads
Adding functionalities of the other existing application back-ends with a modular approach	Fundamental enabling technologies become unusable
Higher efficiency of new and existing developers	Stop focus on the more fundamental problems of ScenWise: By unsuccessfully remaining a healthy Workflow
Creating new tailored products for custom clients using the existing codebase	By unsuccessfully remaining the code quality, which gradually makes the code unmanageable
Being available for customers anywhere located in the Netherlands	

Table 2.1: SWOT-analysis

2.2. Limitations to overcome

In the SWOT analysis in figure 2.1 the current weaknesses are depicted. The weaknesses are indications of what is currently holding ScenWise and SmartRoads 1.0 back. These are the problems this project intends to fix. The strengths indicate aspects which can be utilized during the project. The threats indicate possible problems ScenWise may run into after the project has finished. Finding concrete solutions for the weaknesses will enable the listed opportunities for ScenWise. But, to find these solutions, their underlying problems have to be researched thoroughly. This research can be found in chapter 3.

Based on the current state of the product, it was clear that the new product should overcome the limitations of the current product by having the following features.

- The new product should have an overall improved performance compared to SmartRoads 1.0.
- The new product should be built in a modular way.
- The new product should be easily scalable.
- The new product should consist of high-quality code and documentation.
- The new product should be developed in an agile workflow.

As these only indicate a direction to follow and are not concrete requirements, it was necessary to research the root causes for these features to be limited or absent in the current system.

3 Research

To solve the limitations mentioned in chapter 2, it was essential to determine the root causes of the current problems. From these root causes, concrete requirements can be established that will solve the limitations encountered. Only a summary of the research is presented in this chapter, for more detailed and elaborate documentation of the research phase it urged to look at appendix H.

3.1. Research Approach

The main goal of this research was to determine the causes of the current problems and refine the requirements given by the client based on the found causes. First of all, the different types of requirements were defined by priority and category. These priorities would be determined by the MoSCoW methodology and the categories were determined to be:

- Requirements which aim to increase the performance of SmartRoads
- Requirements which are about adding new features to SmartRoads
- Requirements which aim to ease in-house development
- Requirements which aim to create more configurability to SmartRoads' users.

Finally, research questions were posed to find the root causes of the performance issues. These research questions were defined as:

- How can performance be improved?
  - Is the performance bottleneck located in the API?
  - Is the performance bottleneck located in the database?
  - Is the performance bottleneck located in the back end?
  - Is the performance bottleneck located in the front end?
- How can maintainability be improved?
  - Is the maintainability bottleneck located in the API?
  - Is the maintainability bottleneck located in the database?
  - Is the maintainability bottleneck located in the back end?
  - Is the maintainability bottleneck located in the front end?
- How can scalability be improved?
  - Is the scalability bottleneck located in the API?
  - Is the scalability bottleneck located in the database?
  - Is the scalability bottleneck located in the back end?

- Is the scalability bottleneck located in the front end?

An overview of the complete research approach can be found in fig. 3.1.

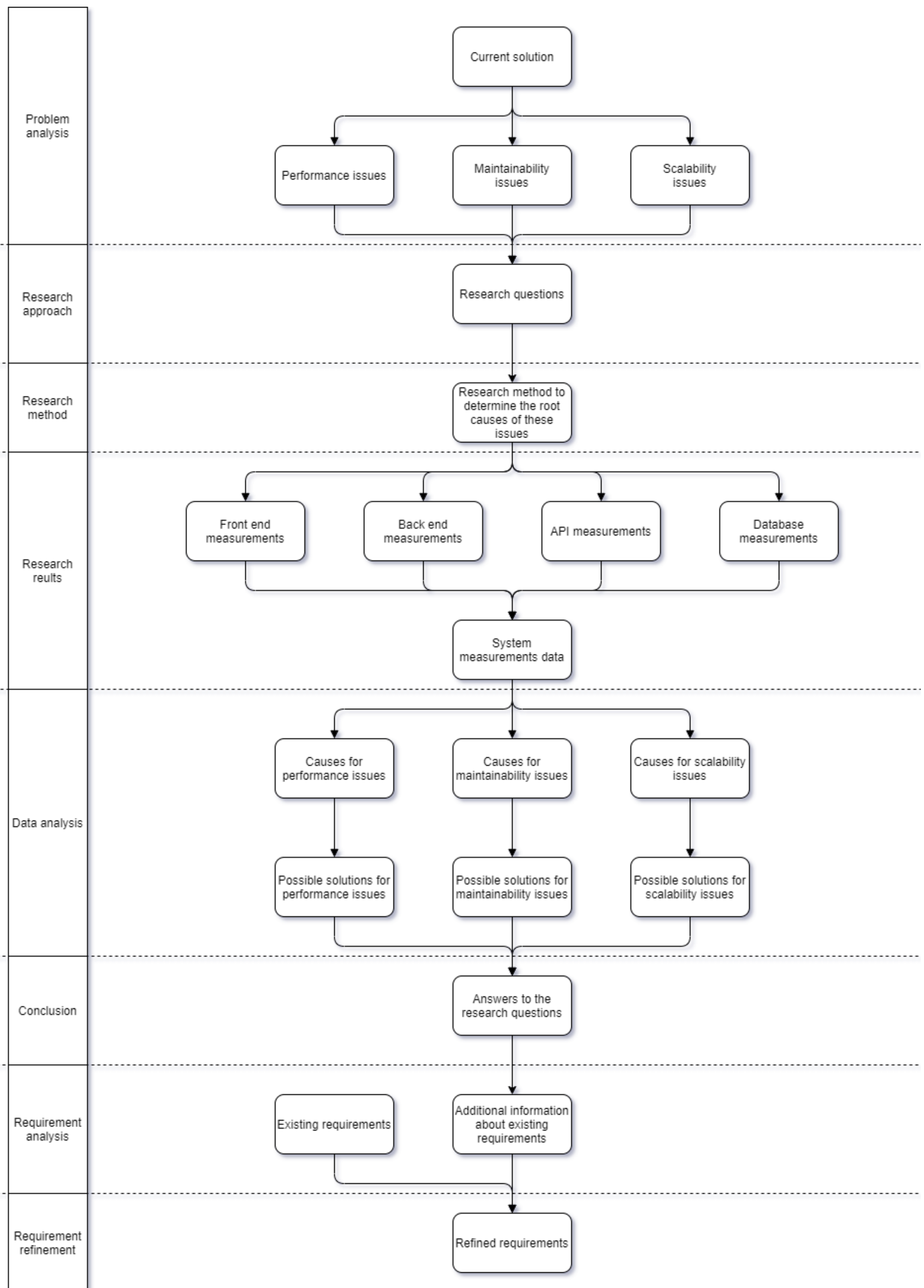


Figure 3.1: Overview of the research approach

3.2. Methodology

To answer the research questions, it was first required to find the proper tools necessary to measure the metrics related to the sub-questions. Java Mission Control (ref) was used to profile the application hosting the front end and the built-in profiler tool of Google Chrome (ref) was used for measuring the client-side. Java Mission Control was used for both components of the back end as well. The API was measured using Postman(ref). And finally, the database was measured by a combination of pgAdmin (ref) and pgBadger(ref). Additionally, multiple user stories were written based on how most users would utilize the system. These user stories can both be used for the research into the causes of the issues as well as the comparison between SmartRoads 1.0 and SmartRoads 2.0. Once this data has been collected, reasoning can be used to determine the causes of the issues. Based on these answers, possible solutions were proposed to solve the issues. Finally, based on the outcome of the research, the research questions were answered. After which the original requirements were analyzed again and refined using the information gathered in the research.

3.3. Results

As mentioned in the introduction of this chapter, there will only be a highlight of the most important findings in this section, more detailed results can be found in the research report H.

After measuring the API it showed that every API call, except one, was performed in less than 500ms. More interestingly was the maintainability of the API. After analyzing what was behind the API it became clear that there were no tests nor documentation available for the API.

Then the database was measured, this was done by performing user stories and to determine the most called queries so they can be measured. These measurements didn't show anything out of the ordinary except that some queries were called at high frequency. This led to some queries taking relatively long when looking at all calls made together. However, when inspecting the database schema itself, it became clear that the database is all but optimal. All in all, the performance of the database itself was reasonable, but the maintainability and scalability are incredibly low due to the absence of documentation and a proper schema.

After the database, the back end was measured. This back end consists of two components, one of which is responsible for everything except reading and writing of local data. Reading and writing of local data is done by the other component, which was analyzed first ("measurement repository"). As can be seen in the research report (appendix H), it shows the performance is drastically worse than a database as it reads fewer data in more time and that that is also its only task as demonstrated in the research report (appendix H). This means that this separate back end is not beneficial to the performance of the application. The other component was responsible for all other actions. Once this component had been profiled, it became clear that 90% of the calls made during profiling were made to the 'TimerThread'. This thread is used for all processes running on a time interval.

Finally, the front end was measured. When performing the user stories while profiling in chrome, it quickly showed clear bottlenecks. Mainly when replaying and rendering the map large spikes in resource usage could be seen for relatively small tasks. The time it takes for the replay with only the status messages and the relative speed around Rotterdam and Amsterdam takes 185.8 seconds to replay 1 hour.

3.4. Data analysis

Having done the research and obtained results from these measurements does not mean the root causes of the limitations have been found. The numbers provide the team with the facts about this system that have to be combined with the technical knowledge of the team to clarify the impact and consequences of the results. One scientific methodology that is suited for this goal is qualified reasoning, where all the information gathered about the architecture, the technologies and the measurements come together. Only then can conclusions be drawn about the root problems that have caused the limitations in the current solution.

3.4.1. API

In the overview of the response times of the API calls in the research report H are the API calls that were measured. From these response times, all calls take less than half a second to complete. This is very reasonable since the requirement is that all data should be retrieved and rendered in less than 1 minute.

Tests and documentation for the API are nearly non-existent and this is the most notable maintainability concern.

There is nothing remarkable concerning the scalability of the API.

3.4.2. database

The measurements of the database, number of queries, and the amount of time a query takes to complete are the metrics that have the most influence of the overall performance. From the overview in the research report H it is clear that there is no single slow query as these are at most a tenth of the slowest API call, which is also reasonable. Despite the absence of slow queries, some queries consume more total time, because they are being queried more often. Even then the most often queried queries have a total duration that is reasonable as can be seen in Research report H. The performance of the database is on par with other databases of this size since it utilizes PostgreSQL, which is one of the most popular database systems in the industry [1].

When the database was viewed from a maintainability perspective, the same issues as the API arose. The naming of the tables is confusing in addition to there not being any documentation.

The workflow around data gathering is where the most significant scalability issues appear. Half of the tables are empty and need manual actions to become populated. These manual actions are quite time-consuming since the data formats do not translate nicely to the tables that are used. Not does the data collection prevent scalability, but the data storage is also sub-optimally designed as can be seen in fig. 3.2. As the amount of data grows, the amount of duplicated data increases.

To conclude the database research, the database has reasonable performance, lacks maintainability, and could not be less scalable.

Figure 3.2: Entity Relationship Diagram (ERD) diagram of the database of SmartRoads 1.0

3.4.3. Back end

The backend op SmartRoads 1.0 consists of two different back ends, one main back end and the 'measurement repository'.

The back end that writes and reads one kind of data locally is called the 'measurement repository'. It stores the average speed that is measured on each road segment in specific regions. This separate back end is not beneficial to the performance of the application, but should not prove a bottleneck. When it is taken into consideration that the code for this system has a small README, no further comments and no tests, which means that it is not maintainable. The fact that the measurement repository stores data locally inherently means that this approach is not scalable.

Then there is the main back end where most of the parser for the NDW data is located. This is the part where the calculations are done and every other part of the application is connected to. The performance does not differ much from the performance of the back end of Analytics.SmartRoads since most of them go to the parser using the TimerThread which is similar to the one in Analytics.SmartRoads, which has reasonable performance. It adds some light computations, which do not negatively influence the performance. The performance of the main back end is sufficient. Maintainability wise the

code is not easily understandable since there are almost no documentation and comments. There are almost no tests and the code style is very inconsistent. As far as scalability is concerned, the whole service runs on a single server and increasing performance is done by upgrading this server. This so-called 'vertical-scaling' is not a scalable approach as the costs for upgrading a single server increases drastically and eventually there is a limit.

3.4.4. Front end

The bulk of the time is consumed by scripting followed by idle and then system. Most of the time there is nothing to do for the system as there is only new data once per minute, so it sits idle. Then once per minute there is new data and this data needs to be plotted and that is where a scripting spike can be seen. This includes everything from animations to function calls in AngularJS [2]. Then there is the 'system' category, but that is just the 'other' category. The whole front end is rendered when new data is coming in. Add the fact that all information is requested from the back end at startup and the whole road network is certainly updated every time even if it is not visible to the user. Then there is the fact that whenever the map is zoomed-in, zoomed-out, or panned, the whole front end renders again. When the performance of these fundamental operations is compared to the industry standard for map applications, Google Maps [3], it stands out that zooming in or out the response time of the SmartRoads 1.0 UI takes 1.5 seconds. Comparing this with Google Maps, which can zoom continuously without stuttering. Panning has a delay of 0.5 seconds with stutters where Google achieves completely smooth panning. These are all things that only make the UI feel unresponsive and are the root cause of the performance bottlenecks, since the front end should only be scripting a few seconds of each minute and not almost half the time.

Another big issue in the front-end was the combining of two of the biggest data-sets used in the application. These two data-sets are the coordinates of all road segments and the measured speed values. All road segments have an ID and these ID's match with the ID's of the measured speed values. The two arrays have the same size, and In the current implementation, for every road segment, the entire speed measurement array is traversed, even if the wanted ID is on the first index. This means that this combining process costs the same amount of time regardless of the order of both arrays, which is always the worst-case runtime $O(n^2)$. This can easily be reduced to $O(n)$ by just accessing the correct field.

The level of maintainability is the same as the other parts, there are no tests, almost no documentation or comments. This means that the code is hard to understand due to the lack of documentation and comments. If there would be any desire to change anything at all, a lot of code would likely break due to the lack of tests. These issues make it inefficient, time-wise, and cost-wise, to maintain this part of the application.

The front end scores low on scalability as well, as additional features can not be added as components.

3.5. Conclusion

In conclusion, the answer to the first research question: 'How can the performance be improved?', is that the application performance suffers mainly from the way the front end handles the rendering on the map and the retrieval of certain data. This contradicts the expectation of the client that the performance issues were caused by the back end. To improve the performance of the system, the focus should be to look at a new front end that only renders changed components and this should have the highest priority. To get a sense of the potential it is needed to look at enabling technologies. [4] comes to mind and is ideal for this case. Then there were several smaller improvements in the API calls and database structure.

The second research question is: 'How can the maintainability be improved?'. The maintainability of all components is very low due to a lack of documentation and tests. Naming is vague for most components and code style is inconsistent. These issues propagate throughout the entirety of the application. It seems easy to improve on these points, so why has this not already been done? For Scenwise it has more value to work on new features than to improve on what already works and since the company has to remain economically beneficial, this has simply been ignored. This is where this project can step in, as the team is not torn between meeting financial obligations and making a sound application, which would be financially more beneficial to the company in the long run.

The last research question is: 'How can scalability be improved?'. There are several big issues with scalability in some parts of the system. The database should not be filled manually and the measurement repository should not save measurements in a local file system. It is not easy to add features to the front end. It comes down to making different design choices for populating the database and using a different framework for the front end. To keep the service scalable in the future it is wise to start with a horizontally scalable architecture. This way processing power can be dynamically managed according to the current needs resulting in better performance as well as lower costs.

3.5.1. Requirements revised

Based on the results of this research, the root problems of the current Scenwise approach were identified and fitting solutions were devised. These problems and their solutions can be found in table 3.1. During the project, it came to the team's attention that the client placed much value in certain features that SmartRoads 1.0 had. It was therefore decided to add a must-have requirement, specifying that certain features of SmartRoads 1.0, as well as the solutions to the root causes, need to be implemented. The end product should have the structure proposed in fig. 3.3. The main idea behind this architecture is that all current features will be retained and features of Analytics.SmartRoads will even be incorporated, but any underperforming feature such as a replay function can easily be replaced by a new one. The details for the long term vision can be found in appendix I

For all new components that the team is going to build there has to be documentation and tests. A new front end should be built that requests and handles data more smartly, so as little data as possible and only what is needed is requested. Then it should adopt a new rendering method so that only changes will be re-rendered. This new front end should be better scalable by using components that can easily be reused and extended. The back end will be a combination of SmartRoads 1.0, Analytics.SmartRoads and optionally brand new back ends for specific features. This architecture does still have the problems found in the back end, but is a step in the right direction since this enables the possibility to rebuild features and then replace them until the old back ends are not needed anymore. A new database will be set up, but used besides the old databases until it is ready to take overall functionality. This new database will not be a local file system and will not require manual data collection and storage. The database schema will have to be thought through to minimize double data. As there are no significant performance issues caused by the database, the performance requirements under the database will be made a standalone requirement. These performance requirements were listed under database, because these were the suspicions of the developer of Scenwise.

These new insights were used to refine the requirements which can be found in D.

Problems in the current application	Proposed solution
Lack of documentation	All new code will have documentation
Lack of tests	New code will be tested
Data in database is stored double	Database will be redesigned
Data collection happens manually	Automated data collection into database
Use of local filesystem	Centralized database using engine
Front end always re-renders entire app	uses a virtual DOM
Front end requests all data at startup	Only request data for what can be seen

Table 3.1: Problems of the current and proposed solutions

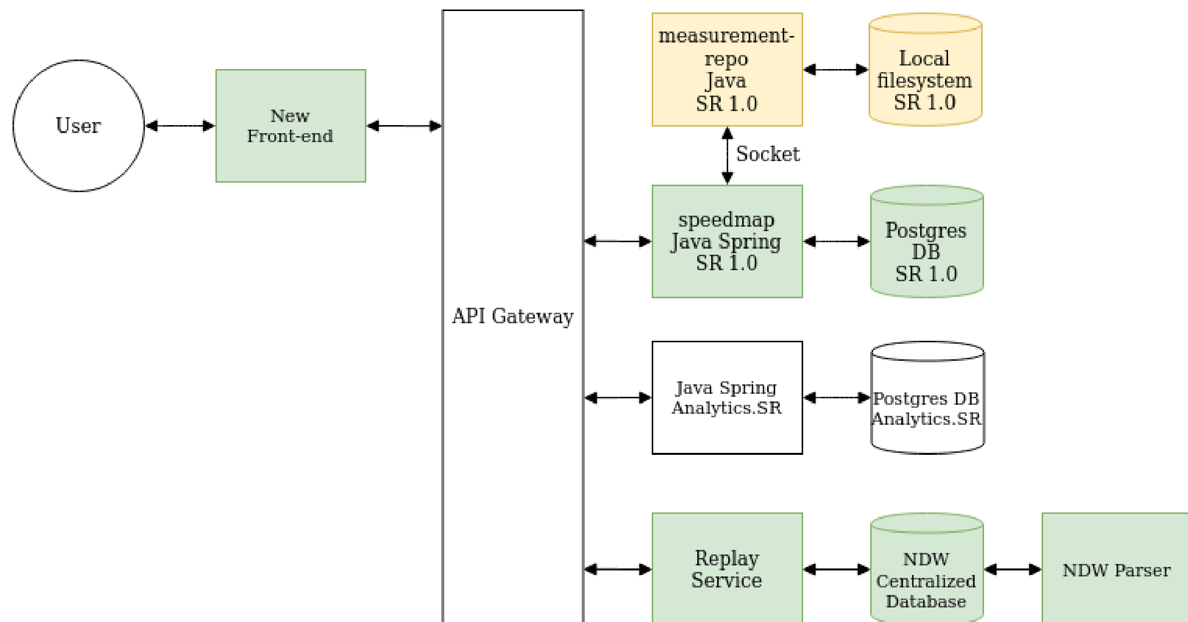


Figure 3.3: Overview of the proposed solution

3.6. Success Criteria

After finding out what the exact root causes are that lie at the root of the problems stated by the client, the requirements were revised to include solutions to these solutions. To determine whether the project is a success at the end of the project it is necessary to determine concrete success criteria. Based on the research and comparisons with industry standards a good understanding of the amount of potential improvement that is realistic in this time frame was developed. The success criteria are determined in table 10.1.

Category	Success criteria
Documentation	Front end, gateway, parser and database parts of the system should have a README
Documentation	At least 80% of the methods should have descriptive comments
Tests	All components should have at least 80% line coverage
Performance	Zooming on the map should take at least 80% less time
Performance	Replay of 1 hour[1] should take at least 60% less time
Modularity	Back end: gateway in place for all services to connect to
Modularity	Front end: components that can be reused

Table 3.2: Success criteria

[1]with the following layers on: accidents, maintenance, obstructions and relative speed in Rotterdam and Amsterdam

4 Design

In this chapter, the design of the solution is discussed. The solution consists of two parts, The LTE design Document, see I, and the codebase. This project needed to be complete within the course of 10 weeks. Because of this limited time, priorities had to be assigned. As stated in chapter 3, the front-end of SmartRoads 1.0 caused most performance issues.

4.1. Long-term evolution design

The LTE design document in appendix I is created to help solve two of the main problems mentioned in chapter 2 in the long-term. The first being the software integration problem, the second being the software development lifecycle problems. This document was written while keeping in mind that someone who has never worked with the system can continue working on it as smoothly as possible. This document is designed because this project should not end up unused like Analytics.SmartRoads, to ensure this, the following chapters are worked out in the LTE design document.

4.1.1. Architecture migration

To solve the integration problems of ScenWise mentioned in chapter 2, a design was created for a future software architecture for the company. A SOA approach was chosen. A detailed elaboration for this architecture can be found in the design decisions in appendix H, multiple architectures were compared. In chapter 2 of the LTE design the migration steps necessary to go from the current software architecture to this SOA have been mapped out.

4.1.2. Software Development Lifecycle Guidelines

To ensure that the problems regarding the software development lifecycle, for example, testing and documentation, mentioned in the chapter 2, will not reappear, clear guidelines have been described in chapter 3 of the LTE design. Should ScenWise find themselves relapsing on this area somewhere in the future, then this chapter can be used to establish what guidelines are failing, and can in turn be solved.

4.1.3. Communication protocols

A design has been created to explain how the communication between services has to be defined. This is regarding documentation of a service API. But also a detailed design is given for the implementation of event-driven communication [5].

4.1.4. Adding a new Service

The last chapter of the LTE design document contains the information necessary for a possible way the company could be using external workforces, like student groups for their software projects. It also contains a small implementation tutorial on how a service should be added to the API Gateway.

4.2. API Gateway

The API Gateway got its form based on the migration part of the design decisions from the research report H. It was designed to enable the first steps towards a SOA as described in the LTE design document found in appendix I. The API Gateway serves as an abstraction layer between the front end and the various back ends and possible new services. This is the place where front end requests are sent to, it redirects such a request to the appropriate back end or service. The decision had to be made to use an open-source project or create the API Gateway from scratch. Starting from scratch seemed like needing to develop functionalities which could be re-used from an existing project. And thus two open-source API Gateway projects were compared, namely Kong [6] and Spring Cloud Gateway [7]. As the developers at ScenWise are familiar with the Spring framework, and the members from the team with Java, and because Spring Cloud Gateway offered all necessary functionalities, the decision was made to set up the API Gateway with Spring Cloud Gateway. More detailed analysis can be found in the research report H.

Documentation of the API is made available with OpenAPI [8] specification files. Which are served via a Swagger UI service. Via this interface custom real-time requests can be made to the gateway in order to explore its capabilities.

The design decision was made to create integration tests in order to monitor the availability and correctness of the endpoints of the API Gateway.

4.3. Front end

Compared to SmartRoads 1.0, most changes were made in the front-end. That makes sense, considering performance issues of SmartRoads 1.0 were almost entirely caused by a poorly designed front end (see appendix H). The basic elements of the UI design of the front end are inspired by SmartRoads 1.0. However, it contains numerous new feature, like the offset slider. These features and designs are shown and described in chapter 7.

4.3.1. Framework

To select the framework best suited for the needs of the project, literature research was carried out. One of the sources reviewed was [9]. The entire research can be found in appendix H. After narrowing the possible options down to React [4] and Angular [2], it was decided to use React as the front-end framework. React is a JavaScript library which is primarily used for building fast and responsive user interfaces. Furthermore, it is component-based [4] which means it allows for modularity within the framework. Updating any component in the application is fast due to the use of virtual DOM. It is lightweight and allows for third party libraries to be used.

To create a good-looking and easily customizable application, React bootstrap [10] was used. React Bootstrap is recommended by the developers of React, and it is the most popular, complete and well-documented styling library currently available [11].

4.3.2. Map

A core component of the front-end is the Map. Therefore, the choice of map service is important. Due to the advantages Mapbox [12] has over Google Maps [3] and ESRI ArcGIS [13] in terms of possibilities, pricing and practicality in developing, it was decided to use Mapbox as the map service. The free trial itself allows to make up to 200.000 API calls and 50.000 map loads which should be enough for the current size of ScenWise. It has, like ArcGIS, library support for ReactJS and good documentation. It also provides good customization. The choice was made to use the standard Mapbox-gl library, in favour of one of the third party Mapbox libraries made for React. This choice was made because the team wanted direct control over the Mapbox functions, to ensure optimal performance and not depend on React.

Deck.gl

Deck.gl [14] is a framework developed by Uber to visualize data on a map, and works in conjunction with Mapbox. It can make defining layers to render on the map easier, but this can come at the cost of performance. This framework is used by SmartRoads 1.0. However, it was decided not to use this

Figure 4.1: ERD diagram of the database of SmartRoads 2.0

framework and instead use the MapBox API directly, to ensure the best possible performance.

However, if the client decides to prioritize switching between different map services, Deck.gl may still be useful. The architecture of the codebase of the front end of SmartRoads 2.0 is made in such a way that converting to Deck.gl layers requires minimal effort.

4.3.3. support for mobile browsers

Because React and React Bootstrap are used, SmartRoads 2.0 is inherently responsive and displays properly on mobile devices, such as smartphones. The performance increase is big enough to let the application run smoothly on a mobile phone. However, currently, the website does not work on an iPhone. This is not a problem as the client never specified that the site should work on mobile phones. The team thinks this issue can easily be resolved in the future, in case it is ever needed to use this application on an iPhone.

4.4. Parser and Central Database

To solve the flawed database structure as described in appendix H, a centralized database solution can be implemented as a separate service in the SOA. In the current SmartRoads 1.0 system there are multiple databases which varied in performance which can be seen in chapter 3. To solve the scalability issues of a local filesystem and the lacking performance it is clear that a database using a Database Management System (DBMS) such as PostgreSQL[15] would be the solution. However, this would imply that there would be data and code duplication. Since the data that is used is the same across all services, the data should be stored in a central location available for all services. If the goal is to implement a central database, a logical consequence would be that a parser is needed to retrieve, parse and store the data in this centralized database. Thus a centralized database and an accompanying parser became part of the design for the new SmartRoads 2.0.

The current database solution contains duplicate data due to tables having near-identical columns. This is improved by adopting and adapting the database schema that was designed by Analytics.SmartRoads. The parser that was designed by Analytics.SmartRoads eliminates the need for manual actions. By parsing every minute and inserting into the centralized database the system is guaranteed to stay up to date. This also solves the manual insertion of data, as this parser can do it completely automated.

The database is designed according to the schema found in fig. 4.1, which solves the data duplication problem.

5 Process

5.1. Workflow

For this group project scrum was used as an agile workflow. Scrum was chosen based on previous positive experiences the team members had with Scrum. More about scrum and agile workflow can be found in appendix appendix I. The reason for using scrum as an agile workflow was the experience the team had with this methodology. The team was also content with the previous experiences they had with scrum. Sprint meetings and daily scrum meetings were held. The Gitlab issue board was used to keep the team informed and updated on what everyone was working on. Gitlab was also a technology the team was familiar with. The general working hours of this project were on weekdays from 09:00 until 17:00. Working outside these working hours could be done on own volition.

5.1.1. Sprint meetings

Each sprint duration was one week and started on Monday. At the start of each sprint, a sprint planning and sprint review were held.

In these meetings, a list of issues for the coming week was created, also known as the backlog. The issues of the previous week were also evaluated and moved to the coming week if they had not yet been finished. These issues were kept up in a Google excel sheet [16]. The issues on the sheet were in turn added and updated on Gitlab issue board. Some of the issues were a bit general, resulting in the team splitting them up into smaller issues on the Gitlab issue board. This way the issues became more manageable and a clearer overview of the overall to-do's was achieved. Each of these issues was assigned to a member, contained a description explaining its purpose. A checklist was made within the issue to keep track of the progress made on an issue. The issues were also given labels to determine its priority, responsibility within the project and the estimated time to complete it.

At the end of the meeting a sprint retrospective was held, where each member stated the positives and negatives of the previous sprints. This way the team was aware of any personal issues or points of improvement, allowing the team to improve the overall workflow.

5.1.2. Daily scrum

The daily scrum meetings took place twice a day. One in the morning at 09:00 and one in the afternoon at around 16:45. In the morning meetings, points of attention, incoming e-mails and the to-do's for that day were discussed. In the afternoon meetings, the team members updated each other on their progress for that day and discussed other points like personal issues and incoming e-mails.

5.1.3. Gitlab workflow

Just as described in section 5.1.1 issues were created on the Gitlab issue board. A branch was made for each of the issues that were about code changes. The branch and the issue were directly linked with each other. When a team member felt like an issue was done a request to merge the branch, linked to the issue, into the develop branch, which, was the name indicates, the branch for the development stage. Other members would, in turn, review the code changes and give feedback or code changes if needed. When feedback or code changes were requested, they first needed to be resolved before

requesting another review. The Continuous Integration, which is an automated tool to check if the code quality is up to date, also played an important role in deciding whether a merge request was approved or not. More information about Continuous Integration can be found in section 6.1.6. If the Continuous Integration and two or more reviewers approved the request, then the branch was allowed to be merged into the develop branch.

Each sprint a merge request was made to merge the develop branch into the master branch which is the deployment branch, that is, it is the version of code that is deployed and live on a server.

5.2. Internal communication

5.2.1. Meetings

As stated in section 5.1, weekly sprint meetings and daily scrum meetings were held with the team. The weekly sprint meetings and the daily scrum meetings were both held using Discord [17]. A minute was taken of the daily scrum meetings in a Google docs [18] document and saved in a shared Google Drive [19] folder. WhatsApp [20] was used to communicate outside of the general project working hours.

5.2.2. Role division

The team roles are defined in table 5.1:

Role	Team member	Definition
Project-leader	Jasper	Responsible for meetings, proper discussions and team ambiance
Lead-communication	Kawin	Responsible for all communication within the team and with the client & coach
Scrum master	Chakir	Responsible for overall scrum process
Lead Testing	Daan	Responsible for test quality and coverage
Secretary	Jan	Responsible for the minutes of every meeting
Quality assurance responsible	Jasper	Responsible for quality of the product by making sure that every requirement is properly met without loss of quality
Front-end responsible	Chakir	Responsible for the front-end code quality
Back-end responsible	Jan	Responsible for the back-end code quality
CI & Git responsible	Daan	Responsible for Continuous Integration and proper GitHub usage

Table 5.1: Role division

5.3. External communication

5.3.1. Meetings client

The team had a meeting close to every week with the client. These meetings were usually on Thursday or Friday afternoon. The meetings were held using Skype [21]. Each meeting was, with the approval of the client, recorded so that the team was able to re-watch parts if needed. A minute was also made for each meeting and saved in a shared Google Drive [19]. In the beginning, the meetings revolved around the research phase and the team getting a better understanding of the problems at hand. In the later stages of the project, the meetings were more on feedback revolving around the creation of the product, where demos were given and progress was explained.

Some of the members also had separate meetings with the lead developer of the client to have their code explained to the team. The lead developer was also invited for a hands-on session, to get used to the code base, the introduced workflow and the good programming practices. This way the knowledge

is passed on more naturally. The lead developer even worked on implementing a new feature using SmartRoads 2.0. These meetings were held using Discord.

5.3.2. Meetings coach

The team also had weekly meetings with the TU coach, Bart Gerritsen, using Skype. Each meeting was also, with the approval of the coach, recorded so that that team was able to re-watch parts if needed. A minute was made for each of the meetings and saved in the shared Google Drive [19]. These meetings were usually held on Thursday. The coach provided the team with feedback and useful information regarding the research and the overall process of the project.

6 Code Quality and Testing

This chapter explains how the team ensured proper code quality both during the project and in the future.

6.1. Code quality

In this section, the different measures taken to ensure good code quality will be discussed.

6.1.1. Documentation

To keep the code clear and maintainable it must be well documented. Documentation is done on 3 levels; class, method/function and line level. Firstly each class should be documented on what it's overall function is and what responsibilities it holds. Secondly, all methods except for simple getters and setters should have a description of the functionality. Should the method contain parameters that are passed, then each of them is also clearly documented. Lastly, complex lines of code and lines of importance are also shortly documented to make the code even clearer.

6.1.2. Code size

The size of each class or component is kept at a minimum, namely 150 Lines of Code (LOC). Units like functions or methods are also kept at a minimal LOC length to ensure understandable and readable code. The line length also has a maximum of 80 in order to make them more readable. These rules are taken as general guidelines concerning the code size.

6.1.3. Front-end

To ensure the correct coding conventions are used for the front-end and for React [4] in particular, ESLint [22] is used to prevent any violations. ESLint runs static code analysis on all front-end code. Static code analysis is a method of examining code before a program is run. This makes the code more consistent, understandable, readable. It also helps in avoiding faults in the system, also known as bugs.

6.1.4. Back-end

In order for the backend code to comply to the coding rules of Java [23], Google Checkstyle [24], Spotbugs [25] and PMD [26] are used as static code analysis tools. With these tools, the chances of bugs occurring within the code are drastically reduced.

6.1.5. OpenAPI documentation

For the static-analysis of the OpenAPI [8] documentations files served by Swagger [27], spectral lint [28] is used. This static code analysis checks if the documentation files conform to the OpenAPI style guide.

6.1.6. Continuous Integration/Continuous Deployment

Software development is a continuous flow of code changes. Each single code change has a risk of introducing bugs or even breaking the code. Detecting this for each change can become quite cumbersome and time-consuming. That is why Continuous Integration is used for this project. In CI, each code change is checked on three different aspects: static analysis, tests and builds. If any of these aspects fail under a code change the CI detects it and warns the developers of failed aspects and the cause, making it easy for the developers to alter this code change for the better. This allows for a simple yet effective workflow in software development. In this project, the Gitlab CI [29] was used to improve the workflow and software development of SmartRoads 2.0.

To automate the deployment of the new code base, Continuous Deployment is used. This stage runs after a merge request to the development or master branch has been made, or when activated manually. The Continuous Deployment uses Docker [30] to easily deploy the system. Docker is explained in detail in section 7.5.4. An update to the development branch is deployed on a staging environment. A new version on the master branch should be deployed on the production servers. ScenWise currently does not use SmartRoads 2.0 in production yet, so this feature is currently not activated. However, all the necessary code is already created.

6.2. Testing

An important factor of attaining and maintaining high code quality is by testing the written code. Testing allows for code changes to happen without breaking the system unconsciously, ensuring high code quality and persistency.

6.2.1. Unit tests

Unit testing assesses the proper behaviour of individual modules/functions of an application in isolation, without any interaction with dependencies, or other modules/functions. If another module, function or dependency is needed, it is mocked. Each service the team implemented is unit tested using Continuous Integration & Continuous Deployment (CI/CD) (see section 6.1.6). Meaning all individual functions should be tested by one or more tests. For example, a function is responsible for the addition of two numbers, it should be assessed whether this function returns the correct value for 2 given numbers. May a developer accidentally change the addition character to a multiplication character this will be caught by a unit test. The benefit of this is that bugs can be found in a very early stage. And for example, when updating dependencies it will immediately be clear if functions do not behave as intended.

The team have set a minimal rule of unit test coverage of 80% for each independent service. This rule is enforced by the CI/CD pipeline, meaning it does not succeed if the minimal coverage is not met. Note that this is a bare minimum, the team aimed to maximize this coverage.

In the front end, it is tested whether each component renders correctly with snapshot tests using jest [31]. The team also make a snapshot of the HTML so that unwanted changes in the HTML can be detected. Additionally, state manipulation and each non-react function is assessed. The Jest [32] dependency was used as a testing framework, and Enzyme [33] to test React [4] components.

The latest coverage report is shown in fig. 6.1.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	90.36	79.68	86.94	92.06	
src	92.86	100	83.33	92.86	
App.js	92.86	100	83.33	92.86	86
src/components/assets/images	100	100	100	100	
Images.js	100	100	100	100	
src/components/assets/images/msi	100	100	100	100	
MSIImages.js	100	100	100	100	
src/components/assets/images/statusMessages	100	100	100	100	
StatusMessagesImages.js	100	100	100	100	
src/components/body	100	100	100	100	
ApiRequests.js	100	100	100	100	
src/components/body/map	81.37	65.48	77.78	84.38	
InitLayers.js	65.67	0	60	68.75	... 78,188,189,194
Map.js	85.44	78.57	74.19	89.36	... 15,149,164,182
UpdateLiveData.js	100	100	100	100	
src/components/body/map/Layers/HWNSpeedLayer	100	100	100	100	
HWNSpeed.js	100	100	100	100	
src/components/body/map/Layers/MSILayer	96.77	91.3	100	98.63	
MSI.js	100	100	100	100	
MSIConversion.js	96.39	91.3	100	98.44	15
src/components/body/map/Layers/OWNSpeedLayer	100	75	100	100	
OWNSpeed.js	100	75	100	100	10
src/components/body/map/Layers/SR1SpeedLayer	100	62.5	100	100	
SR1Speed.js	100	62.5	100	100	12,13,14
src/components/body/map/Layers/dripLayer	94.74	84.62	100	100	
Drip.js	94.74	84.62	100	100	13,33
src/components/body/map/Layers/statusMessagesLayer	100	88.89	100	100	
StatusMessages.js	100	88.89	100	100	55
src/components/body/map/Layers/telpuntLayer	77.78	66.67	100	80.77	
Telpunt.js	77.78	66.67	100	80.77	49,72,78,80,82
src/components/body/map/graph	93.55	100	88.24	93.55	
IntensityGraph.js	93.33	100	87.5	93.33	103
SpeedGraph.js	93.75	100	88.89	93.75	108
src/components/body/map/legend	88.89	50	85.71	88.89	
LegendItem.js	100	100	100	100	
MSILegend.js	100	100	100	100	
SpeedLegend.js	75	50	66.67	75	40
src/components/body/map/replay	91.46	92.98	81.48	92.55	
ReplayBuffer.js	87.8	50	87.5	87.5	47,51,52,67,107
ReplayData.js	89.74	100	80	89.74	49,64,79,94
ReplayInterrupt.js	100	100	100	100	
ReplayRunner.js	93.33	96.67	75	95.89	47,96,142
src/components/header	92.86	50	100	92.31	
Header.js	100	100	100	100	
Toggles.js	91.67	50	100	90.91	33
src/components/header/replay	97.87	90.91	96	97.87	
Controls.js	100	100	100	100	
PickDate.js	80	50	66.67	80	33
Replay.js	100	83.33	100	100	104
Time.js	100	100	100	100	
src/components/header/statusMessages	85.71	100	80.95	91.67	
Messages.js	80.95	100	73.33	88.24	48,70
StatusDropdown.js	100	100	100	100	

Figure 6.1: Coverage report of the front end

6.2.2. Integration tests

Integration tests are responsible for the testing of independently deployed services. Two types of integration tests can be specified, namely, the narrow integration tests and the broad integration tests

[34]. The narrow integration tests are similar to unit tests, except that they are responsible for asserting the correct behaviour of interactions of the tested service with external services. This type of integration tests doesn't require all services to be deployed. The broad integration tests are run on a system where all services are live. These test if multiple services work together as expected, instead of only testing the code responsible for the interactions.

Based on the API specifications broad integration tests were used. As explained these tests are introduced to make sure that if a service's endpoint mutates or stops working the developers are aware of it. This means that ScenWise should find the problem before a customer could. Therefore, each endpoint is tested. For the integration tests of the API Gateway Postman [35] is used. Postman can run collections of tests periodically on the deployed services. This gives a clear overview of the health of the system. These tests are integrated into the CI/CD pipeline.

These integration tests ensure that if something in the service-based system fails, ScenWise can deal with it before a customer has to deal with errors.

The difference with unit testing is that unit testing assesses the proper behaviour of functions in individual modules, whereas integration testing asseses if different modules are working properly together.

6.2.3. System testing

System testing includes testing the fully integrated system. The broad integration testing of the API Gateway can be seen as system testing the back-end. However, it is also desirable the fully integrated front-end is tested. This means executing the system, interacting with it and verifying that the behaviour is as expected. To achieve consistent system testing the CI/CD pipeline (section 6.1.6) automatically deploys the new version of the system to a staging server when a code change is proposed, where two developers (the developer that proposed the change is excluded) manually verify that the application is behaving properly. Until this is done, GitLab prevents the proposed changes from being merged. This approval rule ensures that at least three people agree with the proposed changes, which in turn minimizes the chance of a mistake slipping through.

6.3. SIG Feedback

Over the course of the project, it was required to make two submissions to the Software Improvement Group (SIG). This institution then reviewed the submissions and provided feedback on the maintainability of the system.

6.3.1. Initial feedback

The first submission was made on the 31st of May. This submission accidentally contained some generated code. Luckily, another submission could be made without the generated code of which the feedback was received on the 8th of June. For this feedback, 151 files were reviewed resulting in a total of 43 detected violations and a maintainability score of 4.0 stars. After analyzing these violations and several discussions on Mattermost [36], it was decided to exclude all files related to the parser. This was decided because the submission should only contain components made by the team, and pre-existing components should be excluded. After removing these classes from the review, the feedback contained 28 reviewed files. Of these 28 files, 10 files contained violations with a total of 15 violations.

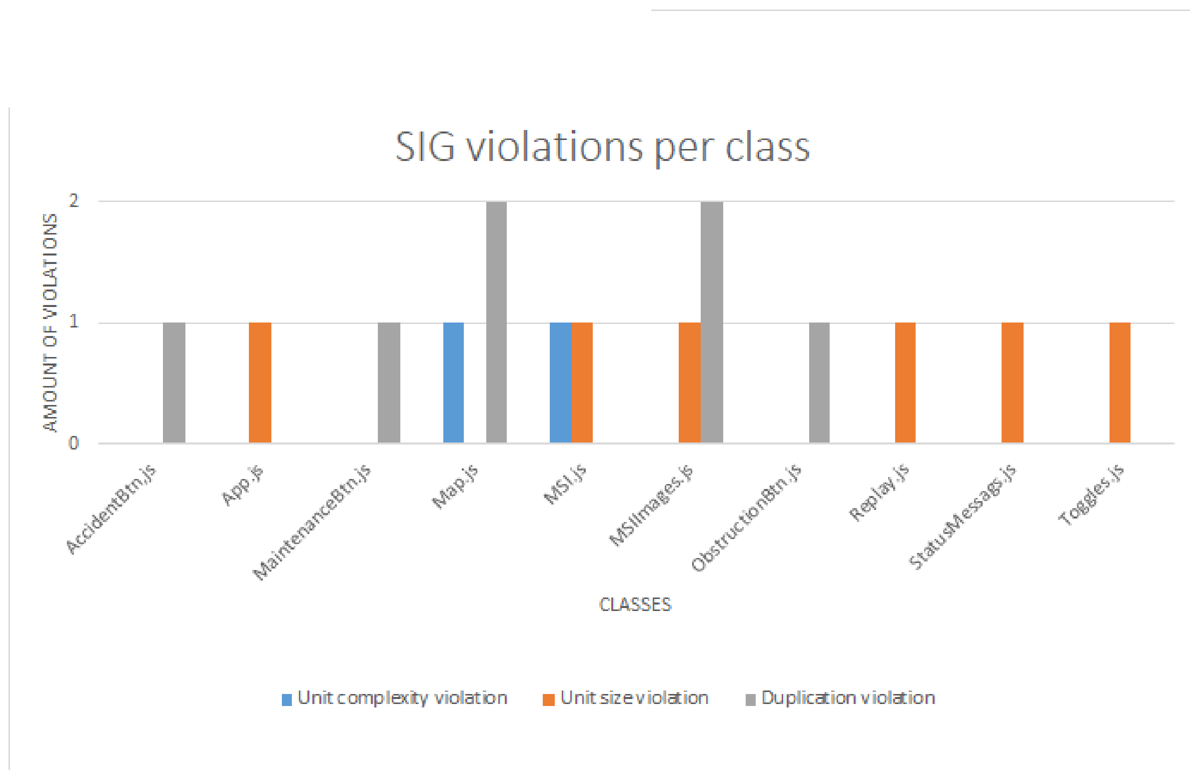


Figure 6.2: Initial feedback from the SIG.

As can be seen in figure 6.2, most of these violations were either related to the unit size or the amount of duplication. Also, 3 classes were responsible for 8 of the 15 violations. Based on this feedback, it was discussed on what is causing each violation and how can these violations be prevented in future code. Finally, for nearly all violations still present in the code, issues were made and solutions reviewed to ensure the violations had been solved. The following violations were deemed to be neglectable:

- MSIImages.js, this class contained a repeated function used to import the different MSI images into the application. The reason why this violation was deemed neglectable was since the 'require' calls in the class are done before runtime. Which means that every 'require' call has to be explicitly mentioned and cannot be made for every entry from a list.

- Map.js, unit size, this class is used to connect all front end components. Therefore, decoupling this class becomes very complex. Although a lot of the functionality has been divided over other classes, the class and its functions are still too large according to the criteria set by the SIG.

6.3.2. Second feedback

The second SIG feedback was received on the 22nd of June. This feedback had a maintainability score of 4.4 stars, which is a clear improvement over the first submission. The second submission included every component submitted for the first quality deadline except for the parser.

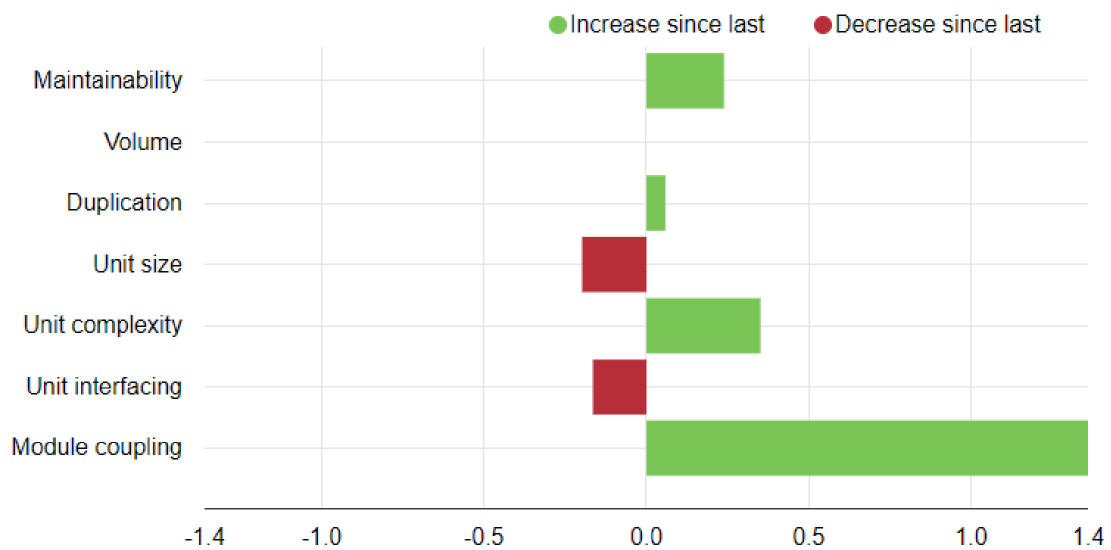


Figure 6.3: Second feedback from the SIG relative to the first feedback.

When compared to the first submission made, figure 6.3 clearly shows that the overall maintainability has improved in the second submission. However, figure 6.3 also shows that the system performed worse on ”Unit size and ”Unit interfacing”. The feedback detected 25 violations in 10 distinct classes in total which can be seen in figure 6.4.

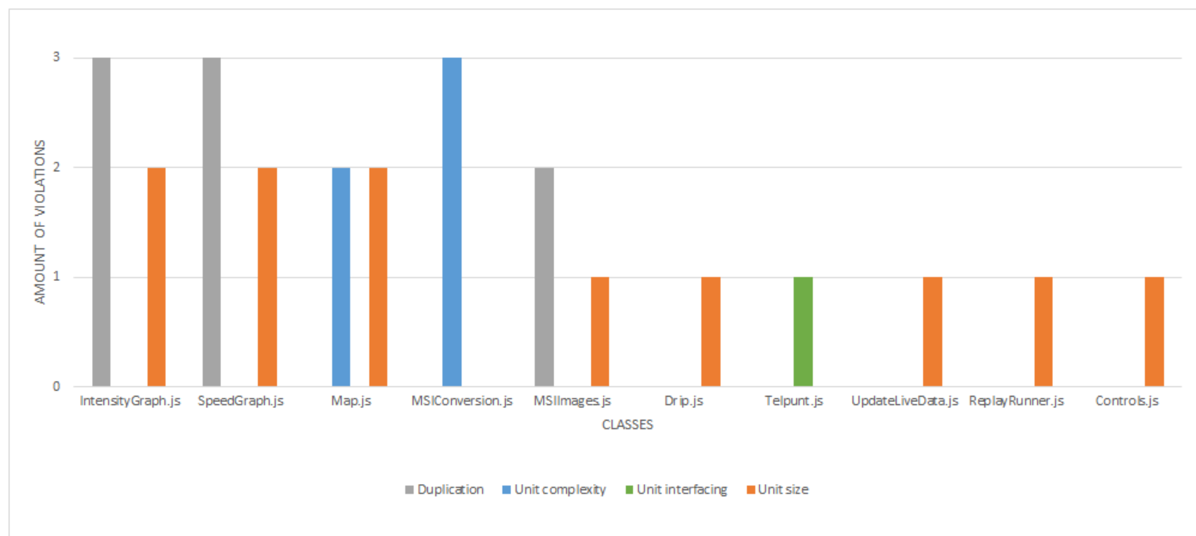


Figure 6.4: Second feedback from the SIG.

As seen in figure 6.4, all violations detected in the first submission have been resolved except for the violations deemed neglectable. The first area where the maintainability rating dropped was ”Unit size”. The violations introduced in the second submission should indeed be refactored into smaller components. However, due to the amount of time left between these implementations and the second submission deadline, other issues were prioritized over the refactorisation. The other area with a lower maintainability rating was the ”Unit interfacing” area. The second submission contains the first and only violation in this area, which was in a function that generates a new pop-up based on the given parameters. Since this function is only used to display the given parameters properly, the decision was made to neglect this violation. The function does not influence the inner workings of the front end and

is only used to generate the HTML content to be displayed into a popup object.

7 Deliverables

This chapter gives an overview of the end product that will be delivered to the client. The implementations as well as the LTE design document are presented. For the implementations, images are displayed to give an insight as to how it looks in the application. Technical details about parts of the implementation process are also elaborated on.

7.1. Features

This section provides the most notable features implemented in SmartRoads 2.0. For each feature, an image and description are provided to give more of an insight as to what it is and what it does.

7.1.1. Front-end design

The design of SmartRoads 2.0 was made to show all functionalities and features in one quick glance. The initial design was inspired by SmartRoads 1.0, but several changes have been made along the way. The colors and theme were chosen to have an ergonomic design. The colouring in the icons and markers on the map, as can be seen in the subsequent sections, are also chosen to be intuitive. Headers, subtexts and tooltips are also added to provide extra information about each of the components in the application.

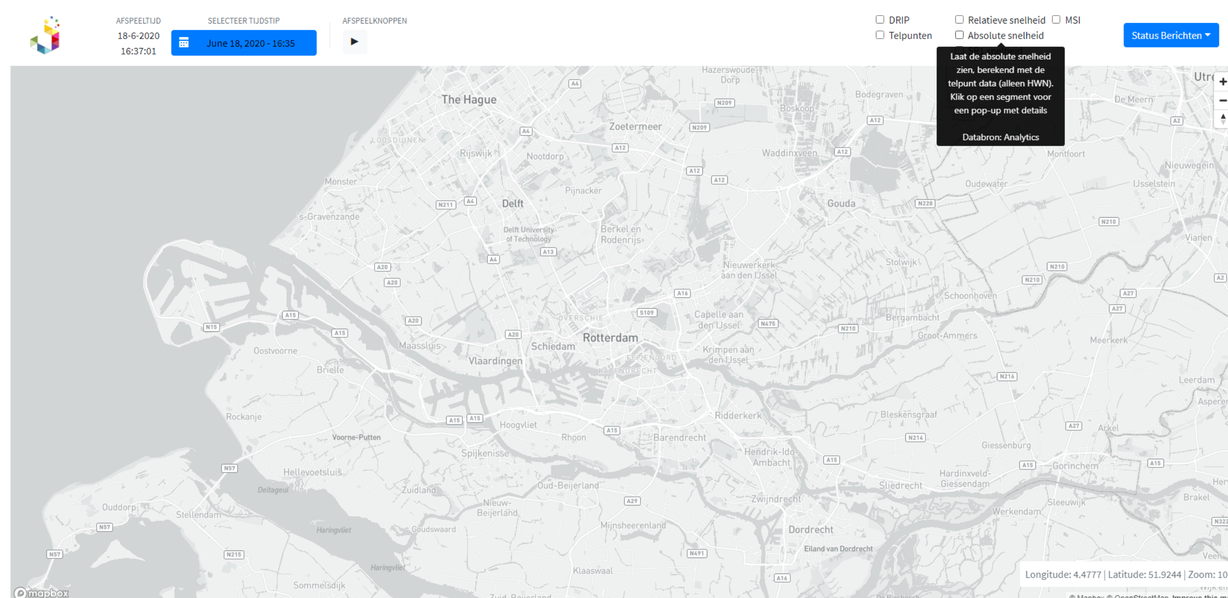


Figure 7.1: Overall design with hovering to show extra information

In figure fig. 7.1 a tooltip can be seen.

7.1.2. Layers

Everything that is displayed on the map is done in the form of a Mapbox layer. Each functionality has its own layer that can be shown or hidden. This is an integrated feature in Mapbox allowing developers to make their product more interactive. SmartRoads 2.0 contains all main layers which SmartRoads 1.0 supports, as described in the MosCow requirements. Additionally, it also contains two layers Analytics.SmartRoads supports. Table 7.1 shows a full overview of the supported layers as well as the areas they visualize data for and the data source that is used, the data that is used via SmartRoads 1.0 and Analytics.SmartRoads originally come from NDW. All layers can be replayed, as described in section 7.1.8. The *relative speeds in Rotterdam & Amsterdam* layer can even be replayed at 512 times speed. If

Layer	Active area	Data source
Dynamisch Route-informatiepaneel (DRIP)	The Netherlands	SmartRoads 1.0 (NDW)
Matrix Signaalgever (MSI)	The Netherlands	SmartRoads 1.0 (NDW)
Status messages	The Netherlands	SmartRoads 1.0 (NDW)
Relative speed	Rotterdam & Amsterdam	SmartRoads 1.0 (NDW)
Intensities (telpunten)	Rotterdam & Amsterdam	SmartRoads 1.0 (NDW)
Relative speed	The Netherlands	Analytics.SmartRoads (NDW)
Absolute speed	The Netherlands	Analytics.SmartRoads (NDW)

Table 7.1: Available layers in SmartRoads 2.0

7.1.3. DRIP

All the DRIP's of the Netherlands can also be displayed. They are initially displayed as dots, to avoid clutter when zooming out. A popup will appear when clicking one of the dots. An example can be seen in fig. 7.2 In the popup, the sign and the information about the sign are displayed. The information consists of the position and the exact time the DRIP was last changed.

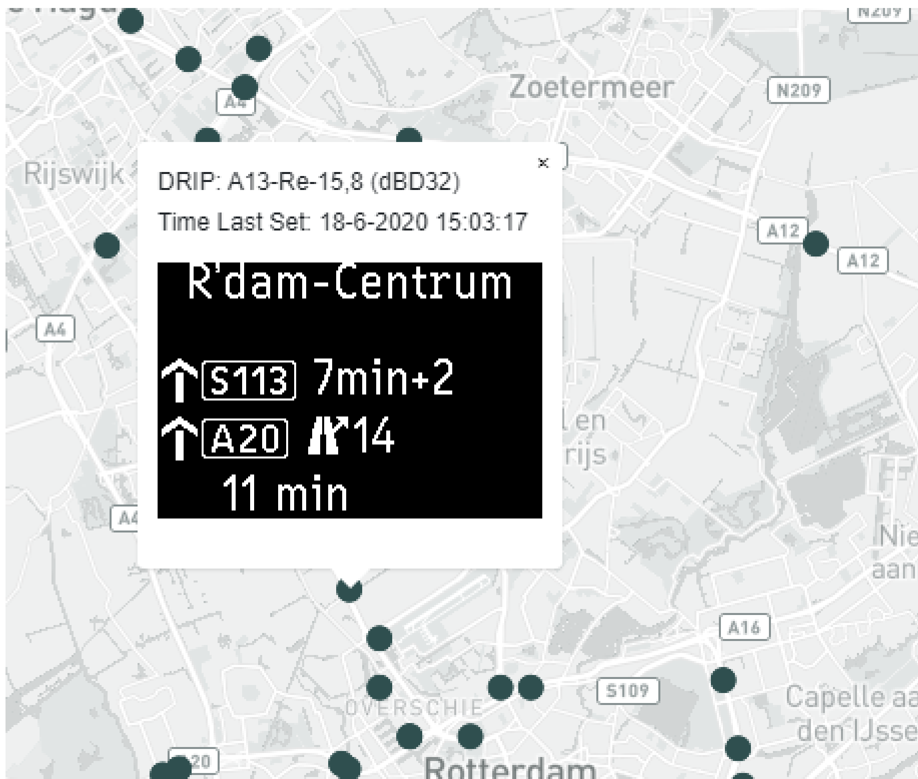


Figure 7.2: DRIP sign

7.1.4. Speed Map

The speed map is the heart of this application as it gives a good insight into the bottlenecks in traffic. It provides a clear overview for traffic analysts to base traffic decisions on.

Relative speed

The relative speed of road segments can be displayed on the entire road network of the Netherlands. The speed, as can be seen in fig. 7.3 is depicted as four different colors which represent the flow of traffic relative to what it could be. A legend is also displayed which provides information about what the colors represent, see fig. 7.4. The legend is also provided with a slider that defines the offset the colored lines have with the road they represent. The client requested this feature halfway through the project as this would give the product a competitive edge. There was no UI design for this feature yet, so the team created its own design. The slider gives an intuitive way to interact with the offset feature.

Figure 7.3: Relative speed

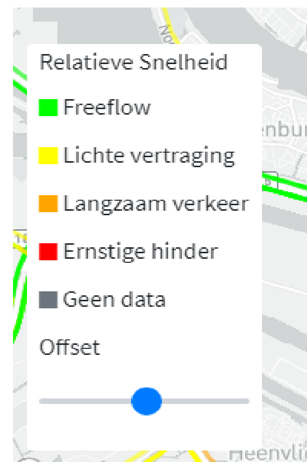


Figure 7.4: Corresponding legend

Absolute speed

The absolute speed can also be displayed on the entire road network of the Netherlands. The speed, as can be seen in fig. 7.5 is depicted as four different colors which represent the maximum allowed speed of vehicles on a road segment. A legend is also displayed which provides information about what the colors represent, see fig. 7.6. The legend is also provided with a slider that defines the offset the colored lines have with the road they represent.

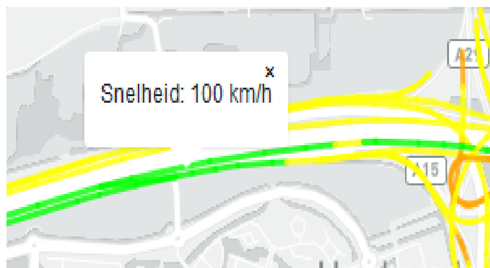


Figure 7.5: Absolute speed

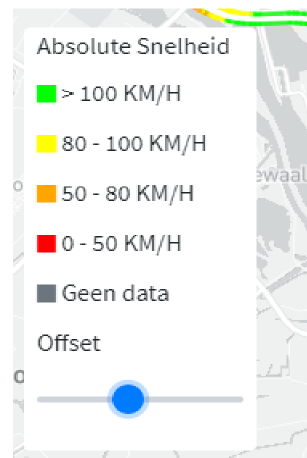


Figure 7.6: Corresponding legend

SR1 speed

As indicated in the name, SR1 speed represents the speed given by the backend of SmartRoads 1.0. It defines the speed of certain areas in the Netherlands. This seems insignificant considering it is only

a part of the Netherlands whereas the other speeds provide information about the entire road network, but it provides extra information about the road segments. Left-clicking a road segment shows, like the other speed implementations, the speed and free flow of a road segment. Right-clicking a road segment, however, now opens a popup that displays a graph, see fig. 7.7. This graph shows the historic data of the road segment concerning speed and free flow, see fig. 7.8. The modal allows for multiple options such as selecting a certain time frame to be analyzed, see fig. 7.10 and fig. 7.11. The graphs in the modal can also be exported as can be seen in fig. 7.9. The graph and its options are all provided by the apex charts [37] library.

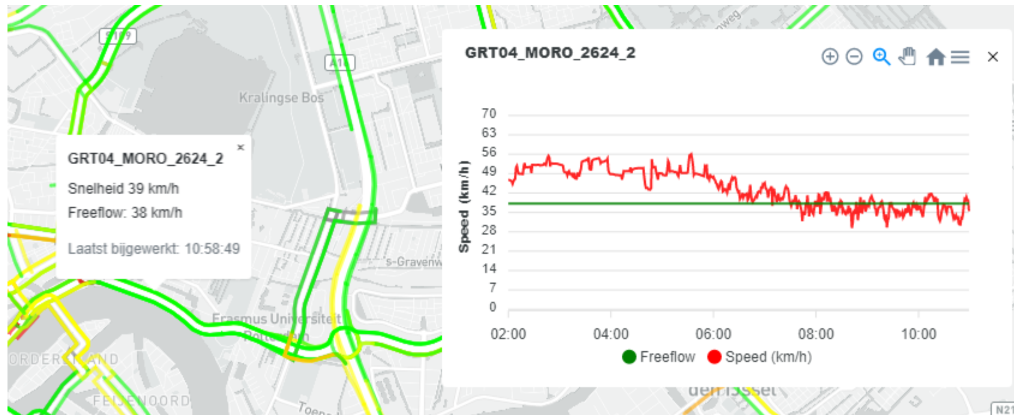


Figure 7.7: The intensity graph over a specific road segment

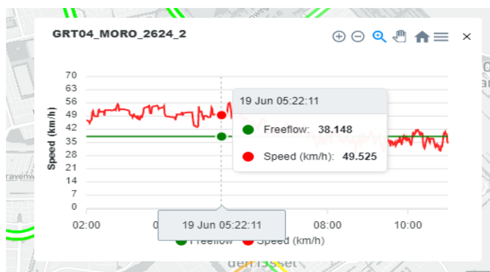


Figure 7.8: Intensity graph with hovering

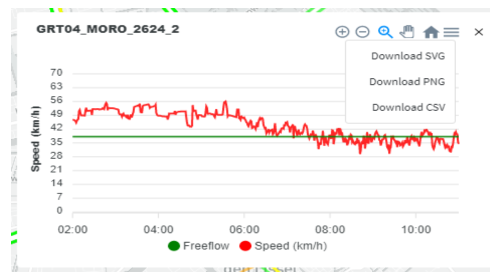


Figure 7.9: Exports for the intensity graph

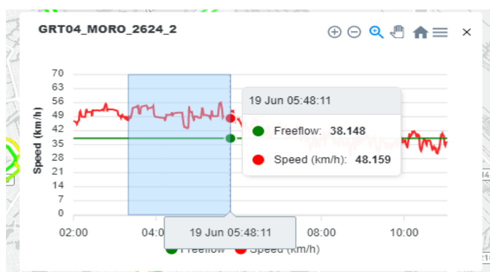


Figure 7.10: Intensity graph time frame selection

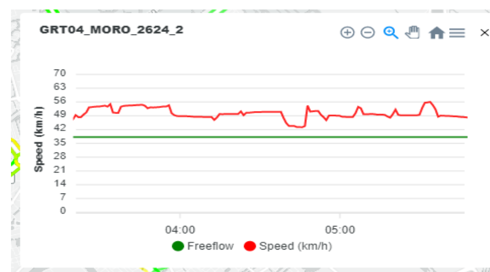


Figure 7.11: Intensity graph specific time frame

7.1.5. MSI

The matrix signs shown on the highways of the Netherlands can also be displayed. The way of displaying depends on the zoom-level the user is in. There are three ways of displaying the MSI layer, depending on the zoom level. When zoomed out the view would be similar to fig. 7.14. A legend is also displayed to indicate what the colors represent. This way of displaying is chosen to avoid too much clutter and give a clear overview of the situation with one glance. When looking for more detailed information about the matrix signs along a highway, the user can opt to zoom in. As the client desired, a middle zoom level was created to show the signs indicated as distinct colors, instead of the actual

sign images, see fig. 7.13. The actual matrix signs which are displayed are shown at an even higher zoom level, as can be seen in fig. 7.12. The option to either show or hide blank matrix signs is also given, as this distinction is quite important for traffic analysis.

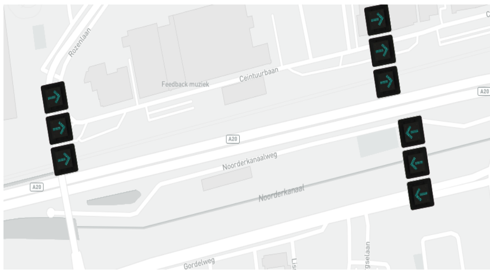


Figure 7.12: MSI zoomed in

Figure 7.13: MSI middle zoom

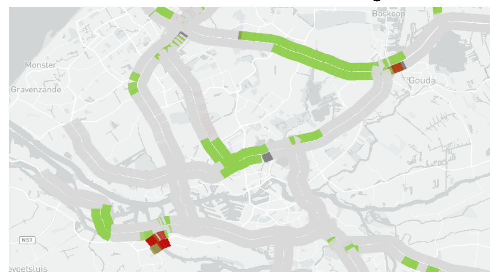


Figure 7.14: MSI zoomed out

7.1.6. Status messages

Situations that cause a hindrance to the natural flow of traffic are grouped into status messages. These messages can be categorized into three distinct types: accidents, (planned) maintenance and (vehicle) obstructions. In fig. 7.15 a list of one of these types is seen with detailed information about the message. These messages can also be plotted on the map as shown in fig. 7.16. The plotted messages also provide detailed information about the message to some extent. Clicking on one of the messages listed pans the map to the corresponding icon, creating a view similar to that of fig. 7.17.

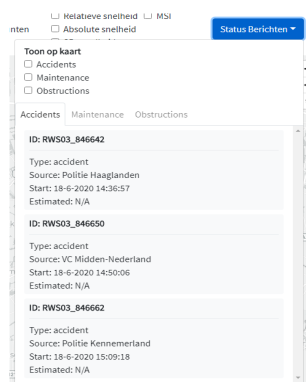


Figure 7.15: Status messages listed

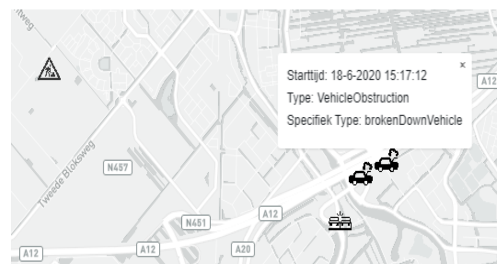


Figure 7.16: Status message icons with an information popup

Figure 7.17: Status message icon with the corresponding list item

7.1.7. Measurement points

The measurement points, called 'telpunten' in the front-end, are displayed as arrows on the map to show the direction of the road the data is about, as can be seen in fig. 7.18. The measurement point layer provides an extra toggle to switch between displaying all measurement points or only points that reached a critical value. By default, only the critical measurement points are displayed to reduce clutter. When left-clicking such an arrow, a popup appears which shows information about the traffic-intensity of that road segment. Information about the traffic-capacity, the number of lanes and current traffic-intensity are displayed. This gives a better understanding of how congested a road segment is. When right-clicking an arrow, a popup graph is shown displaying the traffic-intensity of a road segment, see fig. 7.19. These graphs share the same options as mentioned in section 7.1.4.

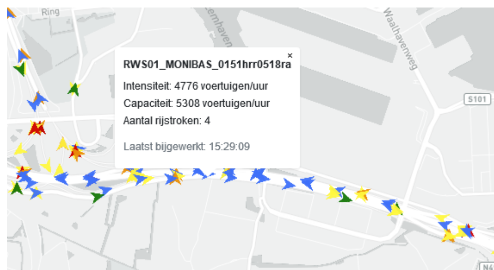


Figure 7.18: Measurement points

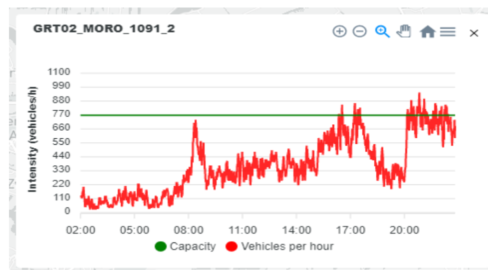


Figure 7.19: Measurement points intensity graph

7.1.8. Replay

The replay functionality allows the user to go back to a certain time to acquire information on the state the traffic was in that specific time period. The user selects a time moment and simply starts playing out the desired layers on the map. All of the layers can be replayed. All of this can be sped-up up to 512 times the normal time-lapse, allowing the user to get a good understanding of what happens throughout the day without having to wait too long. The replay can also be paused or completely stopped to reset to the current time, see fig. 7.21. The moment is chosen using a date and time picker, see fig. 7.20.

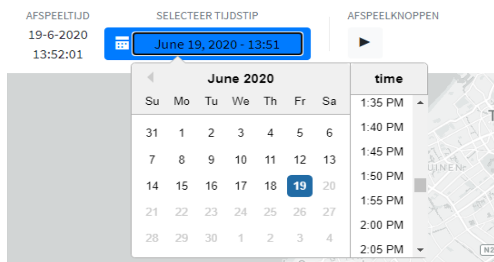


Figure 7.20: Selection of replay moment

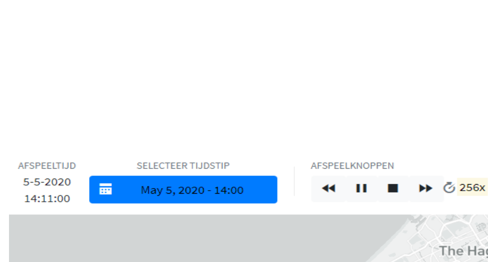


Figure 7.21: Controls when replaying

7.1.9. 3D view

The features can also be viewed in a three-dimensional view. This can give a better view of a certain situation. An example of MSI and a status message can be viewed in fig. 7.22.

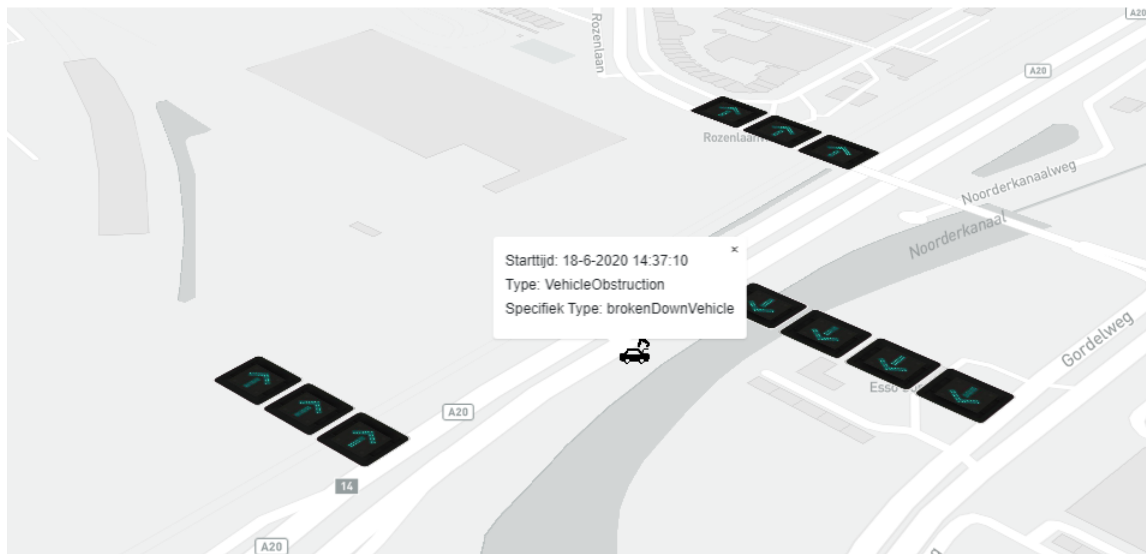


Figure 7.22: 3D view of MSI and a status message

7.2. Long-term evolution design

Next to the implementations for the application an LTE design document was delivered to the client. The document itself can be found in appendix I. It depicts a future architecture of the client's applications and a software development methodology the client should adhere to. This document was created to help the client beyond the scope of this project. This means that features that are not yet part of SmartRoads 2.0 can be added later on using this document.

7.3. API Gateway

As discussed in section 4.2 Spring Cloud Gateway [7] has been used for implementing the API Gateway. As the gateway in this stage is only used to connect a small amount of back ends/services. All configured routes and configurations are set-up in a configuration YAML file. Spring cloud gateway uses filters to mutate requests and responses, for the requirements the build-in filters sufficed. In the case that custom filters are needed these can be added with Java Spring [38]. At the start of the project a proxy was created to all endpoints of SmartRoads 1.0, these endpoints are configured under `http://gatewayhost/sr1/api/{sr1_endpoint}`. This enabled the use of all functionalities provided by SmartRoads 1.0 from the start via the gateway. This proxy is not meant for production deployment, each used endpoint via this proxy has gotten its endpoint configured via `http://gatewayhost/api/{specific_endpoint}`. These are the endpoints meant for production use, as these endpoints are documented and tested with integration tests, this is not done for all endpoints in the proxy as this would have taken a significant time investment for no added functionality to SmartRoads 2.0. It has been made clear to ScenWise's developers it is intended that all used endpoints should be documented and tested accordingly.

The documentation of the API Gateway is done via OpenAPI [8] specification files. These files are served via a swagger-ui [27] service which is accessible via the gateway's home URL `http://gatewayhost/`. More about the swagger documentation can be found in the LTE design document appendix I.

The integration tests are set up with Postman [35]. These integration tests check the availability of the endpoints and if the responses are valid by a given schema. This schema checks if e.g. a property name has not changed or an undocumented property has been added to the response. A more detailed explanation about the integration tests can be found in **??** and in the LTE design document, see appendix I.

7.4. Parser and Central Database

The parser and central database were implemented with the SOA in mind. Since the old databases is still needed for some features to function, implementing the parser and central database as its own separate service was ideal. The old system of SmartRoads 1.0 can be left intact and operational while flaws could get resolved. At this point, no new service is making use of the centralized database yet. Going forward all new features should be built as a separate service and connect via the gateway to utilize the parser and database. Eventually, all old features will have been rebuild and as the old database would no longer be needed, it can be detached from the system.

7.5. Technical details

7.5.1. React

React [4] is used to divide the front-end into different components, ensuring modularity. The components have a parent-child relation in the form of a tree structure. Arguments are passed through children by using props. Each of the aforementioned features that do not include the map, which is one of the components, has its own component. Separate classes are also used to support the components, e.g. make API requests for a layer.

7.5.2. Graphs

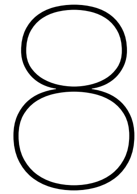
The speed and intensity graphs which provide additional options, such as zooming in on a specific time frame, was made using the apex charts library [37].

7.5.3. Replay

Since the replay functionality was made from scratch and focused on the specific purpose of replaying traffic data, several unique features have been added in its inner workings to ensure smooth and highly configurable replay. First of all, a buffer functionality was implemented. This allows SmartRoads 2.0 to separate the replay data fetching from the replay data rendering. If the replay ”player” plays faster than the replay ”buffer” can fetch the data, then the player keeps halting a second until the requested data has been added to the buffer. Secondly, the replay only fetches data from the enabled data feeds. This enables the player to playback in a speed that is limited by the performance of the worst performing enabled data feed. For example, replaying with only the ”MSI” feed enabled can still replay smoothly at over 100 times the normal playback speed. Finally, replay functionality has been designed to support a new timestamp while replaying. Although the implementation couldn’t be finished due to time constraints, the buffer does contain placeholders in its code where these functionalities should be implemented. This would allow the user to quickly traverse a moment back in time or allows the user to easily skip six hours ahead.

7.5.4. Docker

Docker [30] is used for each component in the system. Docker is a containerization program. Containers bundle all needed configuration files and dependencies required to run a component. This information is stored in so-called images. These images are build based on a specific configuration for each component, which is stored in a Dockerfile. A container ensures that a component can always run regardless of what software is available on the host machine. This is particularly useful for automatic deployment on machines with different operating systems. As SmartRoads 2.0 consists of multiple components, the docker-compose command is used which builds and starts all components with a single command.



Product evaluation

This chapter evaluates the overall product that is delivered to the client.

As can be seen in the previous chapters, SmartRoads 2.0 includes:

- Speed and Freeflow data of the entire dutch road network
- Offset slider for the speed and freeflow data
- Measurement points, with option to only display critical ones
- Live road accidents of the entire dutch road network
- Live road maintenance of the entire dutch road network
- Live road obstructions of the entire dutch road network
- A replay functionality wi
- up to 512 times replay functionality, supporting all layers
- High performance MSI layer
- Speed and freeflow history graph
- Intensity and capacity history graph
- New workflow
- Gateway
- 3D view
- DRIP data across the Netherlands
- Automated data collection from NDW through Parser
- Centralized data storage in PostgreSQL database[15]
- Gateway connecting multiple back ends
- LTE design - Architecture migration
- LTE design - Software Development Lifecycle Guidelines
- LTE design - Communication protocols
- LTE design - Adding a new Service

In order to assess whether these deliverables solve the problems posed by Scenwise in chapter 2, It is necessary to verify which requirements stated in section 3.5.1 have been met. Once these requirements have been verified, a conclusion will be made based on the MoSCoW label of these requirements.

8.1. Implementation challenges

8.1.1. Unique project

As mentioned in section 2.1, the problems tackled by the team during the project deviated from the initially proposed problem, thanks to the proposal of a different project approach. None of the team members had any experience with migrating an application making it hard to estimate the required workload and time needed to resolve the requirements. Although the problem clearly defines that the deliverable can also contain recommendations on features to add in the future (for example, features mentioned in the requirements), was it only possible to meet the requirements labeled as ”must have”. This resulted in a lot of requirements that the team was unable to meet.

8.1.2. Mapbox

As mentioned in appendix H, the downside of React [4] is that it relies on external libraries. One such library was the Mapbox library [12]. There were three choices: react-mapbox-gl, mapbox-gl and react-map-gl. The first choice was react-mapbox-gl, as this allowed for mapbox elements, like a marker, to be used in the form of React elements. As development progressed the drawbacks were starting to show. The library contained bugs and did not leave much of the customizability Mapbox is supposed to provide. That is why the team came to the conclusion to use mapbox-gl, as this was the main library of Mapbox. This change caused a small delay in development as the code used react-mapbox-gl needed to be converted to mapbox-gl code.

8.1.3. Docker

None of the team members had any experience with Docker [30] yet. Learning and setting up did require some investment. This was successfully done in a reasonable amount of time. There were some problems with serving the OpenAPI [8] specification files, as these were split up for better code quality. A solution for this issue was found within a few hours.

8.1.4. API Gateway

During the creation of the routes to the existing and hosted back ends of SmartRoads 1.0 and Analytics.SmartRoads some challenges arose. These challenges were subject to the fact that the deployed versions of the back ends were used.

One of which was Cross-Origin Resource Sharing (CORS) [39], this is an HTTP security protocol, as some of the routes from SmartRoads 1.0 did not allow all origins. This was fixed by setting the CORS to all origins within the gateway, this has been communicated to the client as this does bring some security risks. Setting the CORS in the API Gateway configuration to only the servers which serve the front ends will resolve these security risks.

Another challenge arose with the Analytics.SmartRoads back end. At the start of the development, the server which hosted this back end was very light-weight making the performance very low. This was upgraded to serve the needs of this project. However, as the implemented features grew and thus the number of requests grew the Analytics.SmartRoads server was causing some issues. These issues grew to the point where the server would respond with a server error on more than half of the requests. A quick fix was implemented in the gateway till the server problems will be resolved by ScenWise. This quick-fix will try to make a new request if the server responded with a server-error status code. It will retry the request a maximum of 7 times, this can be changed in the gateway configurations. This quick-fix resolves almost all server-errors. Noteworthy is that in these times in which the Analytics.SmartRoads server is under load the front-end of Analytics.SmartRoads is unusable.

8.2. Requirements assessment

In order to determine whether SmartRoads 2.0 can be labelled as a suitable solution to the problems posed in 2, all requirements posed in D first have to be assessed on whether they are met.

8.2.1. Requirements to increase performance

Must have

SmartRoads 2.0 must be able to retrieve the following datastreams from The NDW

As stated in section 7.1.2, all NDW data feeds corresponding to this requirement as mentioned

in appendix D, have been implemented. Thus this requirement has been met.

SmartRoads 2.0 must have at least the following features of SmartRoads 1.0
As mentioned in section section 7.1.2 all features mentioned in (ref to requirement) have been successfully implemented into SmartRoads 2.0.

The SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0
In order to assess whether this requirement was met, two test cases were written. The first test case involves the user to enable all live data feeds that are present in both SmartRoads 1.0 and SmartRoads 2.0 and traversing from Rotterdam to Amsterdam. The second test case involved doing almost the same as the first test case except that the drip data feed was disabled and that the actions were performed while replaying at 32 times speedup. Both applications were therefore profiled in chrome on a virtual machine, which clearly showed the performance of both SmartRoads 1.0 and SmartRoads 2.0 when performing similar operations.These results are visualized in fig. 8.1 and fig. 8.2. The responsiveness of the UI can be represented by the amount of time it takes to zoom in or out on the map, as this is a very basic and common use case. In SmartRoads 1.0 the amount of time it took to zoom was 1.5 seconds before the UI responded and rerendered. This has been decreased to around 100ms and now is on the same level as [3]. This means an improvement of 93.33% was achieved and can now compete with the industry standard.

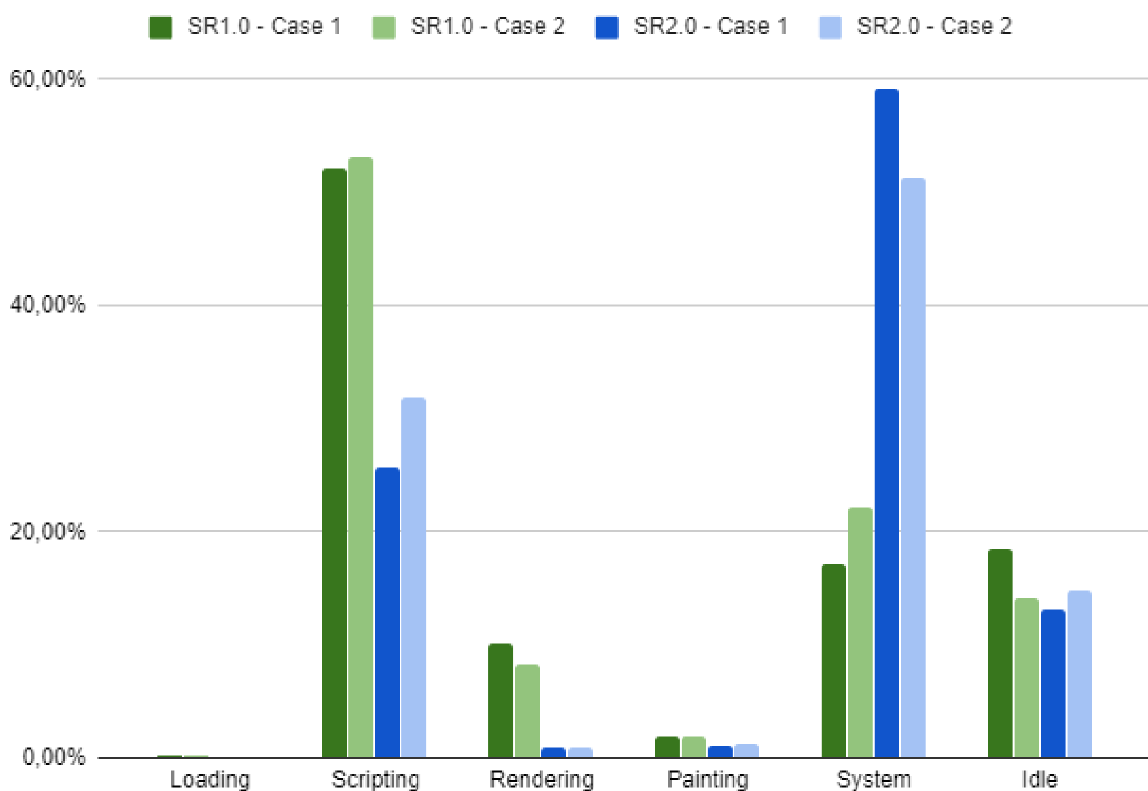


Figure 8.1: Graph overview of table C.3

[0]The Drip data feed was disabled due to the insufficient performance of the external service that provides us with this data.
[0]A virtual machine with a 2 core CPU and 4GB of RAM

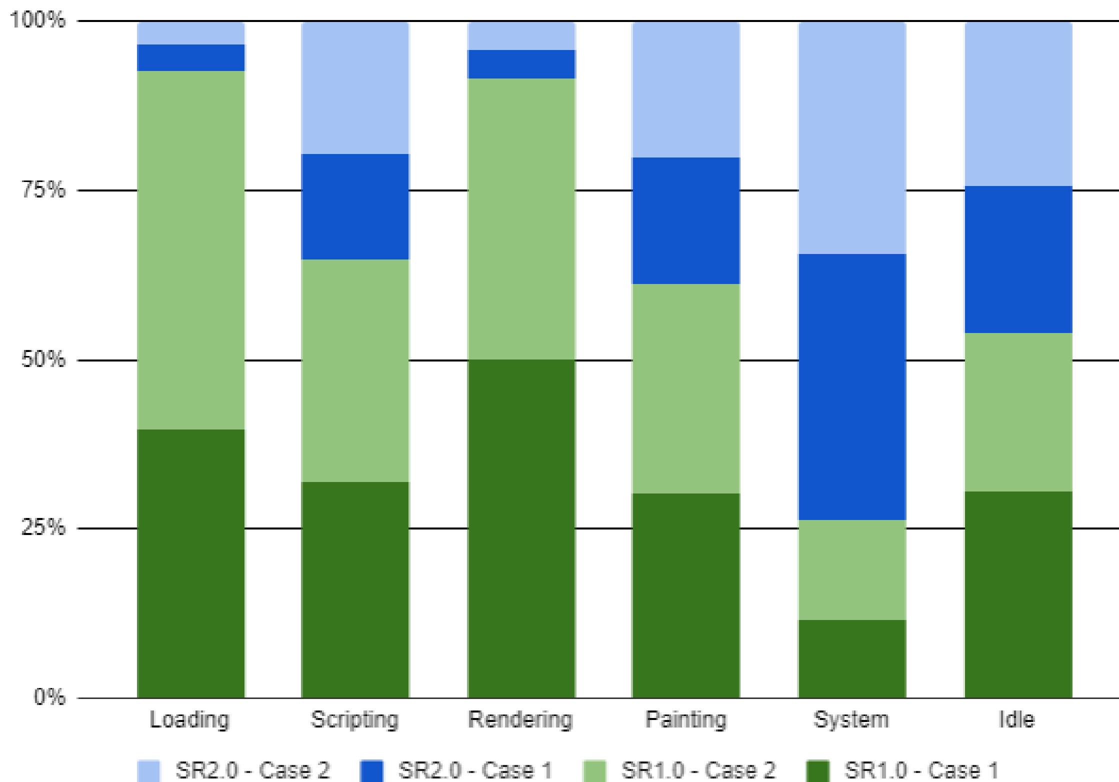


Figure 8.2: Stacked graph overview of table C.3

As seen in fig. 8.1 and fig. 8.2, SmartRoads 2.0 spends less time in every area except for "System". However, it appears that this area covers the time spent processing by chrome itself (including the profiler). After profiling, the console showed the number of samples it was unable to save during replay. The amounts of missed sampled were a lot higher in the profiling of SmartRoads 1.0, which would correspond to the browser freezing several times during these profilings. Since these samples are therefore written less frequently, it makes sense that SmartRoads 2.0 would spend more time writing sample data (causing the larger amount of time spent on "System" actions. All in all, this shows that the interface of SmartRoads 2.0 performs better on every aspect while also displaying more data.

The current replay function is too slow and performance replay[1] of 1 hour should increase by 60% The replay of 1 hour takes 52.93 seconds in SmartRoads 2.0 whereas SmartRoads 1.0 takes 185.8 seconds. This can be calculated to be an increase of 71.51%. As the new replay has a buffer at the start of the replay, the amount of time the buffer needs takes is relatively higher when the replay replays less time. This means that the longer the period of time that is replayed, the less the buffer time will be and thus the higher the percentage of improvement.

Database has multiple issues causing the retrieval of the needed traffic information being too slow. This requirement consisted of two sub-requirements. The first sub-requirement being: "The data must be stored more efficiently e.g. smaller tables, more foreign keys and less duplicate information." and the second being: "A lot of data collection and manipulation happens manually and must be automated.". For the first requirement, a new database schema was designed. This schema is shown in fig. 4.1. The second requirement requires SmartRoads 2.0 to automatically collect and manipulate data which is achieved by using functionalities from SmartRoads.analytics as shown in section 4.4. Therefore this requirement has been met as well.

This requirement consisted of four sub-requirements. The first requirement was "The current replay function is too slow and higher performance is required.". This requirement has been met as can be seen in figure 8.2, where "case 2" refers to a benchmark case focused on measuring the performance

[1]with the following layers on: accidents, maintenance, obstructions and relative speed in Rotterdam and Amsterdam

during replay. This figure clearly shows SmartRoads 2.0 spending less time performing every type of task related to the system performance. The other sub-requirements …

Should have

SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads

Of the three corresponding sub-requirements mentioned in appendix D, only ”Situational messages” was implemented. Due to the limited time and this requirement being labeled as a ”should have” it was decided to neglect the other sub-requirements mentioned.

8.2.2. Requirements for new features

Must have

SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands

Whereas SmartRoads 1.0 was only able to display these features for the areas Rotterdam en Amsterdam, SmartRoads 2.0 is able to display these features for the entire Netherlands. This implementation is described and shown in section 7.1.1.

Should have

A new design should make the updates of the road-network configuration file easier or even automatic. An improved design should be introduced to reduce manual handling and the chance of errors.

As mentioned earlier in section 8.1.1, is this one of the requirements that wasn't met. When prioritizing the ”should have” requirements, the decision was made to exclude this feature. Because this requirement has minimal influence on the actual functionality of SmartRoads 2.0 and solving this requirement would only automate a currently manual process. However, during design, this requirement was kept in mind. This combined with the overall modular design should make it relatively easy to implement a solution that satisfies this requirement.

SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands(e.g. Waze), next to the NDW data.

Although SmartRoads 2.0 currently doesn't ”retrieve, process, visualize and store” any other data feeds, the back end of SmartRoads 2.0 is designed to become a SOA. This means that any other feed can be added as a service via the gateway, more information about this can be found in chapter 2 form the LTE design document found in appendix I.

Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.

This requirement hasn't been met in SmartRoads 2.0 because of the challenge mentioned in section 8.1.1. Since this requirement is more of an extension to the main features than a unique feature on its own, it was decided to prioritize the requirements with a larger impact on the final product.

8.2.3. Requirements for in-house development

Must have

Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.

This architecture design has been made and has been added as the LTE design as appendix I. Therefore the requirement has been met.

Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.

As seen in I, a software development cycle has been defined for the future development of SmartRoads 2.0. Therefore the requirement has been met.

SmartRoads 2.0 must have good code quality

This requirement consists of two sub-requirements. The first requirement being ”The code written by us needs to have at least 80% test coverage.”. At the time of writing, is the line coverage as follows for the three distinct components:

- Front end: 92.06%
- Gateway: 100.0%
- Parser: 89.0% (Although the requirement states that only the code written by the development team needs at least 80% coverage, was the parser included as well, since more tests have been

implemented to create more trust in the external service.

Therefore, this sub-requirement has been met. The second sub-requirement was ”The code written by us needs to be fully and clearly documented.”. This requirement has also been met by making the CI pipeline check if all documentation is present and by failing the pipeline if documentation is missing. Since all sub-requirements have been met, is this requirement met as well.

Back end must be built in a modular way, such that new features can be added easily in the future.
The back end of SmartRoads 2.0 is designed to become a SOA. This means that any feature can be added as a service via the gateway, more information about this can be found in chapter 2 form the LTE design document found in appendix I. A SOA is inherently modular as each service is decoupled from another.

Front end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.
This requirement has been met by making use of React for the front end. React makes use of components that can be swapped or reused, making the front end a modular application and thus meeting this requirement.

Could have

As mentioned in 8.1.1, due to the uniqueness of the project, estimating the required time to resolve the issues corresponding to the requirements was more difficult than expected. The following requirements were labeled as ”could have”.

- *Implement Route planner as a module/service*
- *For the following future modules/services a custom integration plan should be created*
- *Implement the Authentication as a module/service.*

Because these requirements were marked as ”could have” is missing resulting features not critical to the main functionality of SmartRoads 2.0. Also, because the front end was designed modular, is the implementation of these features a minimal effort. Even though these requirements were not met, is this not considered to be of impact on whether SmartRoads 2.0 satisfies the requirements to be deemed a proper solution to the posed problems.

8.2.4. Requirements to increase configurability

Could have

Implement the User profiling as a module/service.
Although the implementation currently does not include an authentication service, is the implementation a minimal effort for the developers. This is due to the modular implementation of the front end in react, meaning the developers can use one of the many authentication modules available for react. All in all, the implementation is a minimal effort in future development but the requirement made was not met. Since this requirement was labeled as ”could have” has this minimal impact on the overall functionality of SmartRoads 2.0.

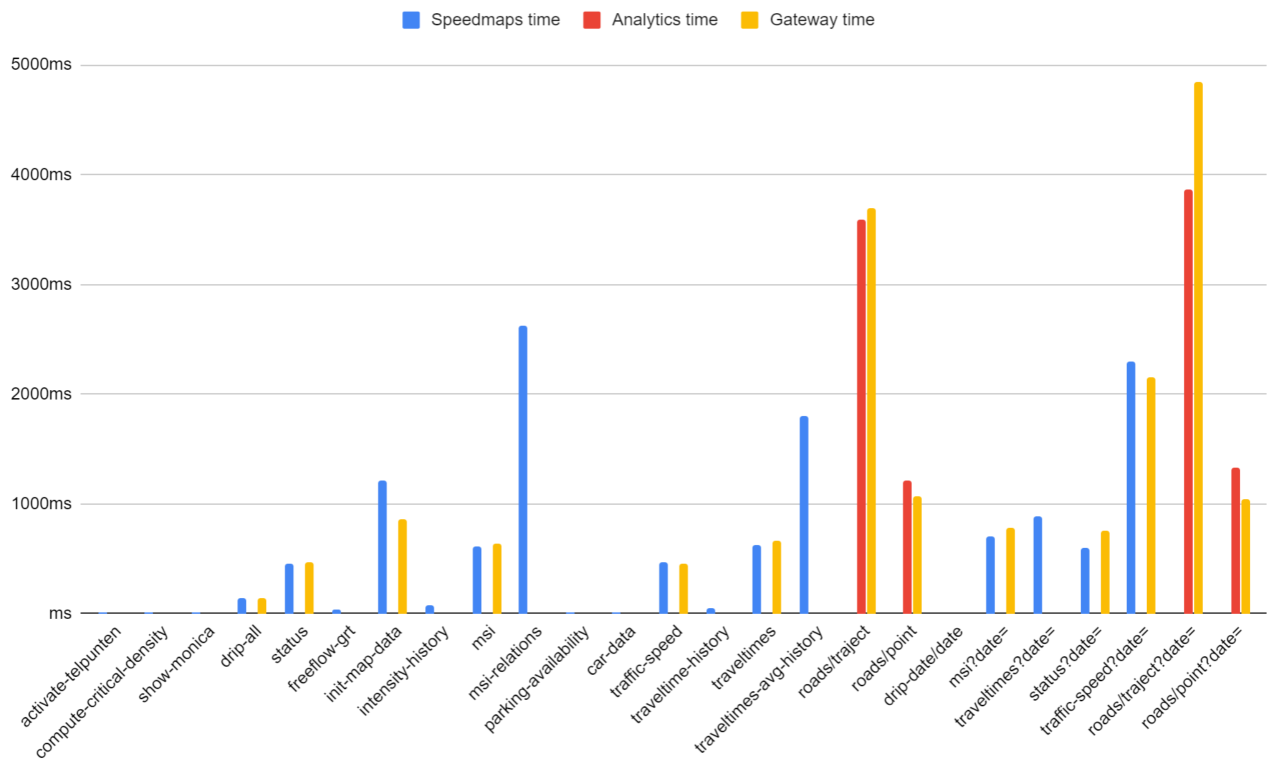


Figure 8.3: Graph overview[2] of table C.1

[2]drip-date is excluded to preserve the utility of the graph. The complete overview can be found in table C.1

9
Project Evaluation

This chapter will evaluate how the team carried out the project, which obstacles were overcome and the satisfaction of the client. In addition, the ethical implications of the project will be considered.

9.1. Workflow evaluation

In this section, the workflow of the project will be evaluated. This project was done amid the COVID-19 pandemic, and even in these exceptional times the project still went very smoothly. This can be explained by the clear and concise guidelines the team created, as described in chapter 5. The team needed to work completely autonomously, which means the team is not only responsible for delivering a final product that lives up to the client's expectations, but also for deciding on the process of achieving this. This includes making decisions on workflow and research methodologies. Using knowledge and experience gained in previous projects, the group chose to adopt the agile workflow SCRUM at the start of the project (see section 5.1.1). The assigned SCRUM roles can be found in section 5.2.2. The SCRUM approach combined with the Gitlab issue board turned out to be an excellent choice. With a minimum of two meetings a day, as described in section 5.2, progress was tracked. This strategy allowed every group member to actively stay informed of what each group member was doing, and the overall state of the project. This was especially important considering the group was forced to work from home due to the COVID-19 crisis. Another benefit of this strategy was that this enabled the team to assign tasks fairly, balancing the workload evenly across all team members. Overall, the group functioned efficiently and agrees that the project was successful in terms of the workflow process.

9.1.1. Inter group challenges

Due to the COVID-19 pandemic, the group was not able to meet in person. The challenge was to keep close contact and keep each other well informed during the project. Using the communication technologies described in section 5.2, the team was able to overcome this obstacle. Other than this, there were no challenges or internal conflicts. The team thinks that the absence of any internal conflicts is caused by the multiple daily meetings as described in section 5.2. Especially the final two points on the agenda of the daily meeting, namely whether someone wanted to raise an additional point of attention, and whether anyone wanted to share a personal matter.

9.2. Client satisfaction

The team beliefs that communication is key. How the team communicated with the client is described in section 5.3.1. These guidelines made sure that every time an impact-full decision needed to be made, the team contacted the client to inform him and ask his opinion on the matter. In close collaboration with the client, the product requirements were reevaluated and prioritized again. The team placed great value in keeping the client up to date and making sure the client always knew what was going on. Keeping the client in the loop prevented any negative surprises for both parties. The client clearly expressed that the team exceeded his expectations, even offering all the team members positions at his company. Furthermore, he explicitly complemented the team's communication skills.

9.3. Course of the project

In this section, the actual course of events during the project will be compared to the original planning made by the team in the project plan. The original planning can be seen in the project planning, see appendix G. In appendix F an overview of the actual course of events is shown.

In week one the team focused on understanding the project and what is expected of the team, establishing the workflow and meeting times, made agreements and solved many practical issues. This week went according to the planning.

In the next week, the team focused on successfully running SmartRoads 1.0 and Analytics.SmartRoads locally, and determining how to make the project scientific. This week mostly went according to the planning. However, the project plan was handed in later than expected. This was done after consulting the BEP coach, so even though this was a diversion of our initial planning, it did not cause any problems. The BEP coach let the team know this was fairly normal.

In the following weeks, the research report was handed in later than expected. The team decided to not just start implementing the features proposed by the client but to take a step back and research if the BEP assignment given by the client was the best way to help ScenWise as a company. This caused the research phase to be longer than originally planned. This did not cause any problems because the team could successfully convince the client that the research stage is an essential part of this project, even though it would take more time than the client initially expected before the first features were implemented. Not following the initial planning created in week one turned out to be a very good decision, because the extra research done caused the team to make the unique shift in approach as described in section 2.1.

Due to this extended research phase, the team started programming in week 4, which was later than expected. However, as stated above, this was not a problem because the client was always kept in the loop. The team could also provide the client with new insights and previously unknown causes of the problems ScenWise was facing. This new information showed the client the value of the research phase, and gave the client the trust that the team was putting in enough work in the project.

The final phase of the project was implementing the requirements as specified in section 3.5.1. This part of the project went according to the planning. The client expressed he was content with the pace of which new features were added.

Apart from the research phase (and the research report as a consequence), the overall planning was executed according to plan. The only difference was the moving of a SIG submission deadline by the BEP coordinators, but these changes did not affect the development considerably.

9.4. Evaluation of Ethical implications

SmartRoads 2.0 only uses publicly available data. The source of this data is NDW. All NDW data is anonymized. This is ensured by the privacy statement of the NDW [40]. Since all data is anonymized, it is impossible to track a vehicle (or person).

Subjects that often bring about ethical concerns are prediction, classification or decision-making mechanisms. However, none of these subjects concern SmartRoads 2.0. SmartRoads 2.0 does calculate freeflow values. The freeflow value is the speed a car can drive at a particular road segment if there is no traffic congestion. The team does not see any way this data could be harmful in any way. Because the rest of SmartRoads 2.0 only visualizes data that is already publicly available, this makes the risk of any negative ethical implications very small.

10 Conclusion

In order to assess whether SmartRoads 2.0 overcomes the limitations SmartRoads 1.0 faces, it is essential to assess whether SmartRoads 2.0 does provide a solution to the problems posed in 2.2.

10.1. Success criteria assessment

Before a conclusion about the results of the project can be drawn, an assessment has to be made whether all the success criteria stated in section 3.6 have been satisfied.

10.1.1. Documentation

In section 3.6, two success criteria related to the documentation are stated.

Front end, gateway, parser and database parts of the system should have a README
As can be seen in the Gitlab repository, all the mentioned components have a README included in their root folder, thus this criteria has been met.

At least 80% of the methods should have descriptive comments
As mentioned in section 8.2.3, the pipeline used during development checks if every method, written by the team, contains a descriptive documentation comment. Since the pipeline for the latest version SmartRoads 2.0 has succeeded, is at least 80% of the methods commented. Thus this criteria has been met.

10.1.2. Tests

All components should have at least 80% line coverage.
As shown in section 8.2.3, has every component at least 80% line coverage, thus this criteria has been met as well.

10.1.3. Performance

Zooming on the map should take at least 80% less time
As mentioned in section 8.2.1 the responsiveness of the zoom functionality in the front end has increased to the point it is the same as [3]. An improvement of 93.33% was achieved which satisfies the success criteria. **Replay of 1 hour[1]should take at least 60% less time**
As mentioned in section 8.2.1 the performance of the replay has increased with 71.51% which satisfies the success criteria.

10.1.4. Modularity

Front end: components that can be reused
By making use of React we satisfy this criteria.
Backend: gateway in place for all services to connect to

The designed back end SOA which can be found in the LTE design document in appendix I is inherently modular. And the implemented gateway is implemented to connect to all the future services.

Category	Success criteria	Achieved
Documentation	Front end, gateway, parser and database parts of the system should have a README	Yes
Documentation	At least 80% of the methods should have descriptive comments	Yes
Tests	All components should have at least 80% statement and 80% line coverage	Yes
Performance	Zooming on the map should take at least 80% less time	Yes
Performance	Replay of 1 hour[2] should take at least 60% less time	Yes
Modularity	Back end: gateway in place for all services to connect to	Yes
Modularity	Front end: components that can be reused	Yes

Table 10.1: Success criteria

10.2. Conclusion

All in all, SmartRoads 2.0 is a valid solution for the problems posed in 2. This is based on the evaluations in chapter 8 and chapter 9 and the assessment of the success criteria in section 10.1. Because all criteria and all "must have" requirements are met, all root causes are resolved (based on the research performed in appendix H) and the verification that the problems have been resolved as well, is done in chapter 8 and chapter 9. Thus SmartRoads 2.0 is the solution to the limitations and problems of SmartRoads 1.0.

11 Recommendations

Thanks to the unique take on the project the group proposed during the problem analysis in chapter 2, they were able to find solutions for not only problems concerning SmartRoads 1.0, but also for the initial integration problems and software development lifecycle problems ScenWise had. This enables the group to give a very broad scope of recommendation. This chapter discusses these recommendations for both the future of the company and the created product.

11.1. Requirements

First and foremost, the requirements stated in section 8.2 that have not been finished should be focused on in the future. These requirements should be re-evaluated by ScenWise and prioritized as the situation is subject to change.

11.2. Long-term evolution design

Should ScenWise adhere to the architecture of the LTE design document, then the possibilities become endless. The mapped out architecture allows for the aforementioned requirements to be implemented as well as extra features.

11.3. Improvements

11.3.1. Moving conversions from the front end to the backend

The performance of the front end can be optimized further by moving even more functionality to the backend. To display a layer on the Mapbox map, the data that the layer uses needs to be converted to the format supported by the Mapbox API. This format is called GeoJSON [41]. In the current version of SmartRoads 2.0, these conversions are made in the front end. However, these conversions could also be made in the backend. This would improve the performance of the front end, leading to shorter loading times and thus making it more responsive.

11.3.2. Separate data in the Analytics.SmartRoads backend

In the current version of the Analytics.SmartRoads backend, the data needed to display the speedmap (see section 7.1.4) is sent as one big response to the front end. This response contains both the coordinates of all road segments, the calculated flow values, and the measured speed values. The latter is updated every minute, but the coordinates of the road segments are static. The coordinate dataset takes up the lion share of the response size.

In the backend of SmartRoads 1.0, both the measurements point layer (see section 7.1.7) and the SR1 speed layer (see section 7.1.4) datasets have endpoints for requesting the needed coordinates and measured live data separately. This enables SmartRoads 2.0 to only fetch the coordinates when the front end is loaded and to update the layer every minute only the measurement need to be fetched again every minute. This reduces loading time significantly and also provides a significant boost in the

replay performance of said layers.

This same principle can easily be added to the backend of Analytics.SmartRoads, however, this was out of the scope of this project.

11.3.3. Support iPhones

Currently, SmartRoads 2.0 supports android devices but does not work properly on iPhones. The team expects this to be simple and easy to fix the platform-specific issue. This was never solved because, at the time of writing, the client is not interested in supporting mobile devices. However, when the client realizes that the application can run smoothly on mobile devices, it is recommended to fix this issue.

11.4. Extra features

These sections describe extra features that are out of the scope of this project but could still be valuable to the company. Some suggestions to possibly valuable features are also made.

11.4.1. New services

The created software architecture provides the option to make use of services. Such services, like the created parser service, can be added in a modular way. This means a separate feature can be added through the backend without any complications. These new services add a lot of variety to the product and make the product easily maintainable, scalable and customizable.

11.4.2. User accounts

ScenWise is ever-expanding and needs to tailor to more and more clients. These clients, however, can vary a lot which could form a problem. User profiling is a way to counter this problem by giving each specific client their own set of tools out of the whole application to work with. This allows for more customization and a broader scope of potential clients. There should also be a login service, to distinguish the clients and provide security, which can be created using the newly created software architecture.

11.4.3. Event-driven communication

Event-driven communication is ideal for live-data purposes. It is recommended such a system is in place for communication between the services. This is explained in more detail in in chapter 4 of the LTE design document in appendix I.

11.4.4. Server-sent events

With the implementation of event-driven communication between the services in the back end. It would be of great value to transform the connection between back end and front end to server-sent events. This will ensure the most recent available data is as fast as possible available to the users. These events can be expanded to only send the deltas in the data to minimize the amount of data that has to be transferred.

11.4.5. Kubernetes, Docker Swarm

To utilize the SOA to its full potential a container orchestra software should be implemented. A container orchestra enables automatic scaling based on the load the system receives. Also, the automatic deployment of new versions of services will be provided by this software. This could be a BEP project on its own.

Bibliography

[1] W. J. Gilmore and R. H. Treat, “Introducing postgresql”, *Beginning PHP and PostgreSQL 8: From Novice to Professional*, pp. 573–577, 2006.

[2] AngularJs, *Angularjs api*, Accessed: 2020-5-5, 2020. [Online]. Available: https://angularjs.org/.

[3] *The google-maps platform*, Accessed: 2020-5-8, 2020. [Online]. Available: https://developers.google.com/maps/documentation.

[4] ReactJS, *A javascript library for building user interfaces*, Accessed: 2020-5-5, 2020. [Online]. Available: https://reactjs.org/.

[5] M. Richards, *Software architecture patterns by mark richards*, Feb. 2015. [Online]. Available: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/.

[6] *Kong gateway*, Accessed: 2020-5-8. [Online]. Available: https://konghq.com/kong/.

[7] *Spring cloud gateway*, Accessed: 2020-5-8. [Online]. Available: https://spring.io/projects/spring-cloud-gateway.

[8] *Openapi*, Accessed: 2020-6-24. [Online]. Available: https://swagger.io/specification/.

[9] Anonymous, *Stack overflow developer survey 2019*, Accessed: 2020-5-5, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology.

[10] R. Bootstrap, *React bootstrap*, Accessed: 2020-5-5, 2020. [Online]. Available: https://react-bootstrap.github.io/.

[11] React, *Adding bootstrap*. [Online]. Available: https://create-react-app.dev/docs/adding-bootstrap/ (visited on 06/17/2020).

[12] *Mapbox gl js api*, Accessed: 2020-5-8, 2020.

[13] *Mapping and analysis: Location intelligence for everyone*, Accessed: 2020-5-8, 2020.

[14] Deck.gl, *Deck.gl*, Accessed: 2020-5-6, 2020. [Online]. Available: https://deck.gl/.

[15] *Postgres documentation*, Accessed: 2020-4-30. [Online]. Available: https://www.postgresql.org/docs/.

[16] *Google sheets*, Accessed: 2020-6-24. [Online]. Available: https://www.google.nl/intl/nl/sheets/about/.

[17] *Discord*, Accessed: 2020-6-24. [Online]. Available: https://discord.com.

[18] *Google docs*, Accessed: 2020-6-24. [Online]. Available: https://www.google.nl/intl/nl/docs/about/.

[19] *Google drive*, Accessed: 2020-6-24. [Online]. Available: https://www.google.com/intl/nl_ALL/drive/.

[20] *Whatsapp*, Accessed: 2020-6-24. [Online]. Available: https://www.whatsapp.com/.

[21] *Skype*, Accessed: 2020-6-24. [Online]. Available: https://www.skype.com/.

[22] *Eslint*, Accessed: 2020-6-24. [Online]. Available: https://eslint.org/.

[23] *Java*, Accessed: 2020-6-24. [Online]. Available: https://www.java.com/nl/about/whatis_java.jsp.

[24] *Google checkstyle*, Accessed: 2020-6-24. [Online]. Available: http://google.github.io/styleguide/javaguide.html.

[25] *Spotbugs*, Accessed: 2020-6-24. [Online]. Available: https://spotbugs.github.io/.

[26] *Pmd*, Accessed: 2020-6-24. [Online]. Available: https://pmd.github.io/.

[27] *Swagger*, Accessed: 2020-6-24. [Online]. Available: https://swagger.io.

[28] Stoplight, *Spectral*, Accessed: 2020-6-22, 2020. [Online]. Available: https://stoplight.io/open-source/spectral/.

[29] *Gitlab*, Accessed: 2020-6-24. [Online]. Available: https://about.gitlab.com/.

[30] Docker, *Docker*. [Online]. Available: https://www.docker.com/ (visited on 06/18/2020).

[31] *Snapshot testing - jest*, Accessed: 2020-5-5, 2020. [Online]. Available: https://react-bootstrap.github.io/.

[32] *Jest*, Accessed: 2020-5-5, 2020. [Online]. Available: https://jestjs.io/.

[33] *Enzyme*, Accessed: 2020-5-5, 2020. [Online]. Available: https://jestjs.io/docs/en/snapshot-testing.

[34] M. Fowler, *Integrationtest*, martinFowler.com, Jan. 2018. [Online]. Available: https://martinfowler.com/bliki/IntegrationTest.html (visited on 06/22/2020).

[35] Postman, *Postman*. [Online]. Available: https://www.postman.com/ (visited on 06/17/2020).

[36] MatterMost, *Mattermost*, Accessed: 2020-6-21, 2020. [Online]. Available: https://mattermost.com/product/.

[37] *Apex charts documentation*, Accessed: 2020-6-23. [Online]. Available: https://apexcharts.com/docs/options/annotations/#.

[38] *Spring framework documentation*, Accessed: 2020-4-30. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html.

[39] MDN, *Cross-origin resource sharing*, Accessed: 2020-6-24, 2019. [Online]. Available: https://developer.mozilla.org/nl/docs/Web/HTTP/CORS.

[40] NDW, *Privacyreglement ndw*, Accessed: 2020-5-19, 2018. [Online]. Available: http://www.ndw.nu/downloaddocument/834642063e0ab6c3bc55ffed54e494c4/Privacyreglement%5C%5C%5C%20NDW.pdf.

[41] *Geojson*, Accessed: 2020-5-5, 2020. [Online]. Available: https://geojson.org/.

[42] Google, *Waze*, Accessed: 2020-5-5, 2020. [Online]. Available: https://developers.google.com/waze.

Glossary

agile Software development methodology centered around the idea of iterative development. 16

Analytics.SmartRoads The application developed by a BEP group[1] in 2019 to improve on SmartRoads 1.0. vi, 9, 11, 13, 15, 27, 35, 42, 45, 46

back end Relating to or denoting the part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data. iv, 2, 9–12, 14, 32, 34, 35, 38, 39, 44, 46, 62, 66

Continuous Deployment Continuous deployment refers to automatically releasing a developer's changes from the repository to production, where it is usable by customers. It addresses the problem of overloading operations teams with manual processes that slow down app delivery. It builds on the benefits of continuous delivery by automating the next stage in the pipeline.. 20

Continuous Integration Continuous integration is an automation process for developers. Successful CI means new code changes to an app are regularly built, tested, and merged to a shared repository. It's a solution to the problem of having too many branches of an app in development at once that might conflict with each other. . 17, 20

Continuous Integration & Continuous Deployment The CI in CI/CD always refers to continuous integration, which is an automation process for developers. Successful CI means new code changes to an app are regularly built, tested, and merged to a shared repository. It's a solution to the problem of having too many branches of an app in development at once that might conflict with each other. Continuous deployment refers to automatically releasing a developer's changes from the repository to production, where it is usable by customers. It addresses the problem of overloading operations teams with manual processes that slow down app delivery. It builds on the benefits of continuous delivery by automating the next stage in the pipeline.. 20, 51

database A structured set of data held in a computer, especially one that is accessible in various ways. iv, 9–12, 15, 33, 34, 43, 44

develop branch The branch for the development stage. 16

Discord Software application for video and audio communication. 17

Entity Relationship Diagram Entity Relationship Diagram, also known as ERD, ER Diagram or ER model, is a type of structural diagram for use in database design. An ERD contains different symbols and connectors that visualize two important information: The major entities within the system scope, and the inter-relationships among these entities. . 9, 51

Feyenoord Dutch football club located in Rotterdam. 1

framework An abstraction in which software providing generic functionality can be selectively changed by additional user-written code. 11, 14

front end Relating to or denoting the part of a computer system or application with which the user interacts directly. iv, vi, 2, 10–12, 14, 20, 21, 24, 35, 43–46, 58, 60, 62, 64, 66

Gitlab issue board A board containing the issues on the backlog provided by Gitlab [29]. 16, 41

[1] `https://repository.tudelft.nl/islandora/object/uuid%3A1f39c7d1-7cbe-4985-a032-5ea71d5daa49?collection=education`

Google Drive A shared file storage and synchronization service. 17, 18

master branch The branch for the deployement of the code. 17

merge request Merge requests allow you to visualize and collaborate on the proposed changes to source code that exist as commits on a given Git branch. A Merge Request (MR) is the basis of GitLab as a code collaboration and version control platform. It's exactly as the name implies: a request to merge one branch into another . 17, 20

PostgreSQL PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads . 9

repository A central location in which data is stored and managed. 11

Rijkswaterstaat The Dutch Highway Agency. 1

service Software functionality with the purpose of clients being able to reuse it for different purposes.. 13, 21, 22

SIG Stands for Software Improvement Group, an institute focused on analysing software maintainability. 22, 23, 42

SmartRoads 1.0 An application developed by Scenwise for traffic engineers to monitor the traffic situation. ii, iii, 2, 4, 8–11, 14, 15, 26, 27, 32, 33, 35–37, 42, 45, 49, 50

SmartRoads 2.0 The developed product of this BEP, a newer version of SmartRoads 1.0. iii, 2, 3, 8, 18, 20, 32, 34–36, 38, 39, 45, 46

Acronyms

API Application Programming Interface. iii–v, 9, 13–15, 22, 32, 33, 35, 57, 67

CI/CD Continuous Integration & Continuous Deployment. 20, 22

CORS Cross-Origin Resource Sharing. 35

DRIP Dynamisch Route-informatiepaneel. 27

ERD Entity Relationship Diagram. 9, 15

LOC Lines of Code. 19

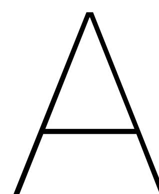
LTE Long-term evolution. iii, 13, 14, 26, 32, 34, 38, 39, 44–46

MSI Matrix Signaalgever. 27

NDW Nationale Databank Wegverkeersgegevens. 1, 9, 27, 34, 35, 42

SOA Service-oriented architecture. iii, 13–15, 33, 38, 39, 44, 46

SWOT-analysis Strengths Weaknesses Opportunities and Threats analysis. 2, 3



Info Sheet

General Information
Title of the project: SmartRoads 2.0
Name of the client organization: Scenwise
Date of final presentation : 1 July 2020

Problem description Scenwise is a company specialized in traffic data science and the smart mobility domain. One of their products is SmartRoads, an application to view and replay traffic situations. However, this application is very limited due to existing performance issues.
Challenge The original challenge was to build a new version of SmartRoads, however after research the challenge became to improve the inefficient software development process and build a modular and integratable architecture. All while still developing a working end product.
Research During the research phase it was discovered that the initial project and its tasks did not cover and solve the most fundamental issues that were actually present. This meant the team had to come up with a brand new project to solve these newfound problems.
Process The software development process followed the [42] framework and was devided into weekly sprints. Each sprint started with the product backlog and ended with the retrospective. This way the team remained flexible and agile. When the client requested a new feature, the road offset in fig. 7.4, during the project it could be included in the next sprint.
Product The product that was created, SmartRoads 2.0, has overcome the performance issues completely and can now display traffic information over the whole of the Netherlands. The architecture has been designed from the ground up with modularity and integratability in mind. SmartRoads 2.0 was developed using an agile workflow and coding best practices which improve the software development process. Part of the software development process was CI/CD, which made sure our code was tested continuously.
Outlook At the end of the project multiple deliverables will be presented to the client: the end product, the improved software development process and the LTE plan. The LTE plan contains our vision on the future architecture of SmartRoads 2.0 and will ensure that the client has the knowledge to continue developing software the right way. More suggestions to improve the product by making the application event-driven or implementing further optimizations in the front end can be found in chapter 11

Members of the project team

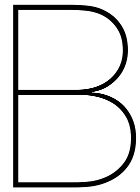
- Name: Jan Buijnsters
  Interests: Back-end development, Machine learning, Automation, Entrepreneurship
  Project role(s): Back-end responsible, Secretary
  Project contributions: LTE Design, API Gateway, Minutes
- Name: Daan Hofman
  Interests: Data Science, Artificial Intelligence, Full-stack development
  Project role(s): CI/CD & Git responsible, Lead Testing
  Project contributions: Research, Front end Mapbox integration, front end features and design, all front end layer functionality
- Name: Jasper Klein Kranenbarg
  Interests: Algorithm design, Big data, Machine learning, Artificial intelligence and optimization.
  Project role(s): Project-leader, Quality assurance responsible.
  Project contributions: Research, Replay feature, Front end features, Benchmarking, Data feed optimization.
- Name: Chakir el Moussaoui
  Interests: Web development, Big data, Block-chain technology
  Project role(s): Front-end code quality, Scrum master
  Project contributions: Front-end features and design
- Name: Kawin Zheng
  Interests: Data analytics, Machine learning

  Project role(s): Lead communication, Lead research
  Project contributions: Research, Replay feature, Parser & database, front end features, Benchmarking

Contributions made by the entire team: The entire team worked together on the reports and final presentation.

Contact

K.F. Chan	Founder Scenwise	kinfai.chan@scenwise.com
B. Gerritsen	Department of Software Technology at TU-Delft	B.H.M.Gerritsen@tudelft.nl
J.L. Buijnsters	Team member	buijnstersjan@gmail.com
D. Hofman	Team member	daanhofman@outlook.com
J.G.P. Klein Kranenbarg	Team member	japperkk@hotmail.com
C. El Moussaoui	Team member	chakir.emoussaoui@gmail.com
K. Zheng	Team member	kawin.zheng@gmail.com



ProjectForum Description

Company Background
Scenwise B.V. is company with a lots of experience in Data Science & 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 (e.g. traffic monitoring, data fusion, Big Data, Machine Learning)

Our customers are the Dutch Highway Agency (Rijkswaterstaat), Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, ITS system suppliers, Feyenoord stadium and recently also the city of Edmonton in Canada.

Project description
Scenwise has developed the SmartRoads platform. SmartRoads is used by our customers for monitoring and analyse traffic situations. For example:

- Feyenoord Stadium uses SmartRoads to monitor the traffic during a day with football match;
- Rijkswaterstaat uses SmartRoads for road-tests of Driving Automation.

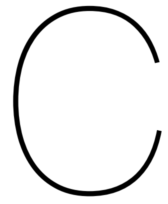
Scenwise intends to extend the functionalities of SmartRoads to support a wide range of customers. For this reason, we start a new project to rebuild SmartRoads. The development will include:

1. Flexible web-interface supporting different maps layers (e.g. Google Maps, ESRI, OpenStreetmaps, Mapbox, etc.);
2. Big Data back-end incorporating a wide range of Open Data and proprietary data;
3. Real-time data analyse functionalities;
4. High performance replay-function;
5. Interface with Decision Support Applications (e.g. Automatic Accident Warning);
6. Interface with mobile apps to provide traveller advises.

Within this project, a team of students will work together to conduct research, make design decisions, develop the application inclusive testing. The back-end should be a new big data platform using different data sources. The front-end should be web-based using modern graphical interfaces which different real-time graphics to provide adequate information to the end-users. This is an innovative project with a lots of technical challenges.

We are looking for enthusiastic students with skills and interest in database techniques, data processing, geographical information systems (GIS), visualization techniques and algorithms who are willing to take the challenge of developing a new platform to meet the upcoming requirements of our customers.

Scenwise will provide detailed information for making design decisions and guidance in making both the technical designs and the graphical interfaces designs. Setting up test scenario's for software acceptance is also a part of the project. Since this is a new development, there are enough rooms for the project team to make design decisions. If the project team is familiar with Java Spring (back-end), PostgreSQL, Python, React or Angular (front-end), then they may re-use part of the current source code.



Evaluation Success Criteria

HTTP method	API Call	Response size	Speedmaps time	Analytics time	Gateway time
POST	activate-telpunten	206B	15ms		
POST	compute-critical-density	205B	16ms		
PUT	show-monica	177B	19ms		
GET	drip-all	568.23KB	139ms		143ms
GET	status	268.72KB	458ms		465ms
GET	freeflow-grt	138.53KB	43ms		
GET	init-map-data	60.2MB	1217ms		868ms
GET	intensity-history	13.06KB	81ms		
GET	msi	3.19MB	609ms		636ms
GET	msi-relations	13.22MB	2630ms		
GET	parking-availability	13.78KB	16ms		
GET	car-data	260B	19ms		
GET	traffic-speed	139.47KB	465ms		462ms
GET	traveltime-history	20.55KB	57ms		
GET	traveltimes	136.15KB	632ms		663ms
GET	traveltimes-avg-history	38.79KB	1804ms		
POST	roads/traject	37.01MB		3.59s	3.7s
POST	roads/point	16.79MB		1215ms	1071ms
GET	drip-date/date	575.21KB	46s		
GET	msi?date=	3.41MB	706ms		777ms
GET	traveltimes?date=	137.03KB	886ms		
GET	status?date=	351.51 KB	601ms		752ms
GET	traffic-speed?date=	139.47KB	2.3s		2.16s
POST	roads/traject?date=	36.97MB		3.87s	4.85s
POST	roads/point?date=	16.82MB		1331ms	1050ms

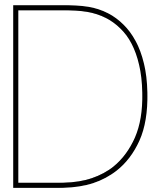
Table C.1: API calls and their response times

Profiler Category	SR1.0 - Case 1	SR1.0 - Case 2	SR2.0 - Case 1	SR2.0 - Case 2
Loading	158ms	222ms	14ms	22ms
Scripting	34182ms	36832ms	15742ms	32824ms
Rendering	6618ms	5791ms	548ms	858ms
Painting	1223ms	1316ms	708ms	1272ms
System	11256ms	15337ms	36314ms	52755ms
Idle	12168ms	9830ms	8088ms	15238ms
Total	65605ms	69328ms	61414ms	102969ms

Table C.2: Front end absolute performance

Profiler Category	SR1.0 - Case 1	SR1.0 - Case 2	SR2.0 - Case 1	SR2.0 - Case 2
Loading	0,24%	0,32%	0,02%	0,02%
Scripting	52,10%	53,13%	25,63%	31,88%
Rendering	10,09%	8,35%	0,89%	0,83%
Painting	1,86%	1,90%	1,16%	1,24%
System	17,16%	22,12%	59,13%	51,23%
Idle	18,55%	14,18%	13,17%	14,80%
Total	100,00%	100,00%	100,00%	100,00%

Table C.3: Front end relative performance



MoSCoW Requirements

Requirements to increase performance	**Design Goal**	**MoSCoW classification**
SmartRoads 2.0 must be able to retrieve the following datastreams from The Nationale Databank Wegverkeersgegevens (NDW): • The NDW ”incidents” stream [1] • The NDW ”measurement” stream [2] • The NDW ”Wegwerkzaamheden” stream [3] • The NDW ”trafficspeed” stream [4] • The NDW ”traveltime” stream [5] • The NDW ”Matrixsignaalinformatie” stream [6]		*Must have*
The SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0. • Rendering of data on the map needs better performance e.g. retrieved and rendered within 1 minute. • To achieve this we need to apply component rendering instead of rendering complete front end • To achieve this the front end must only request what can be seen by the user	*Performance*	*Must have*
The current replay function is too slow and performance replay[7] of 1 hour should increase by 60%	*Performance*	*Must have*
Database has multiple issues causing the retrieval of the needed traffic information being too slow. The following requirements will solve it: • The data must be stored more efficiently e.g. smaller tables, more foreign keys and less duplicate information. • A lot of data collection and manipulation happens manually and must be automated.	*Performance*	*Must have*

Table D.1: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance

Requirements to increase performance	**Design Goal**	**MoSCoW classification**
SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads features: • Situational messages. • Implement analytics contour plot. • Implement analytics cloud plot.		*Should have*
SmartRoads 2.0 must have at least the following features of SmartRoads 1.0: • The DRIP layer • The MSI layer, with different zoom levels • The measurement point layer, with option to display only critical points. • The relative speed layer (Rotterdam and Amsterdam) • The measurement speed history graph for every road segment • The measurement point history graph for every measurement point		*Must have*

Table D.2: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance (part 2)

Requirements for new features	**Design Goal**	**MoSCoW classification**
A new design should make the updates of the road-network configuration file easier or even automatic. An improved design should be introduced to reduce manual handling and the chance of errors.		*Should have*
SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands	*Performance & Scalability*	*Must have*
SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands (e.g. Waze [42]), next to the NDW data.		*Should have*
Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.		*Should have*

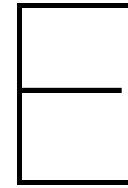
Table D.3: List of requirements which are about implementing new features, along with their corresponding design goal and importance

Requirements for in-house development	Design Goal	MoSCoW classification
Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.	*Maintainability*	*Must have*
Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.	*Maintainability*	*Must have*
SmartRoads 2.0 must have good code quality. • The code written by us needs to have at least an 80% test coverage. • The code written by us needs to be fully and clearly documented.	*Maintainability*	*Must have*
Back end must be built in a modular way, such that new features can be added easily in the future.	*Maintainability*	*Must have*
Front end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.	*Maintainability*	*Must have*
Implement Route planner as a module/service.	*Maintainability*	*Could have*
For the following future modules/services a custom integration plan should be created. • Authentication. • User Profiling. • Route Planner. • Automatic Accident Detection. • Scenario Designer. • Traffic Light Analytics.		*Could have*
Implement the Authentication as a module/service.	*Maintainability*	*Could have*

Table D.4: List of requirements which aim to ease in-house development, along with their corresponding design goal and importance

Requirements to make it more configurable to customers	Design Goal	MoSCoW classification
Implement the User profiling as a module/service. • Add user profiles, with settings per profile. • Users should be able to temporarily change the settings of his/her user profile.	*Maintainability*	*Could have*

Table D.5: List of requirements which aim to make SmartRoads more configurable to customers of Scenwise, along with their corresponding design goal and importance



MoSCoW Evaluation

Requirements for new features	MoSCoW classification	Fulfilled?
A new design should make the updates of the road-network configuration file easier or even automatic. An improved design should be introduced to reduce manual handling and the chance of errors.	*Should have*	No
SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands	*Must have*	Yes
SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands (e.g. Waze [42]), next to the NDW data.	*Should have*	No
Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.	*Should have*	No

Table E.1: List of requirements which are about implementing new features, along with their corresponding design goal and importance

[7] with the following layers on: accidents, maintenance, obstructions and relative speed in Rotterdam and Amsterdam

Requirements to increase performance	MoSCoW classification	Fulfilled?
SmartRoads 2.0 must be able to retrieve the following datastreams from The Nationale Databank Wegverkeersgegevens (NDW): • The NDW ”incidents” stream [1] • The NDW ”measurement” stream [2] • The NDW ”Wegwerkzaamheden” stream [3] • The NDW ”trafficspeed” stream [4] • The NDW ”traveltime” stream [5] • The NDW ”Matrixsignaalinformatie” stream [6]	*Must have*	Yes
SmartRoads 2.0 must have at least the following features of SmartRoads 1.0: • The DRIP layer • The MSI layer, with different zoom levels • The measurement point layer, with option to display only critical points. • The relative speed layer (Rotterdam and Amsterdam) • The measurement speed history graph for every road segment • The measurement point history graph for every measurement point	*Must have*	Yes
SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads features: • Situational messages. • Implement analytics contour plot. • Implement analytics cloud plot.	*Should have*	Partly, only the situational messages
The SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0. • Rendering of data on the map needs better performance e.g. retrieved and rendered within 1 minute. • To achieve this we need to apply component rendering instead of rendering complete front end • To achieve this the front end must only request what can be seen by the user	*Must have*	Yes

Table E.2: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance (part 1)

Requirements to increase performance	**MoSCoW classification**	**Fulfilled?**
The current replay function is too slow and performance replay[7] of 1 hour should increase by 60%	*Must have*	Yes
Database has multiple issues causing the retrieval of the needed traffic information being too slow. The following requirements will solve it: • The data must be stored more efficiently e.g. smaller tables, more foreign keys and less duplicate information. • A lot of data collection and manipulation happens manually and must be automated.	*Must have*	Yes

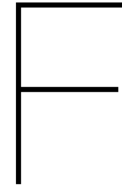
Table E.3: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance (part 2)

Requirements for in-house development and maintainability	MoSCoW classification	Fulfilled?
Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.	*Must have*	Yes
Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.	*Must have*	Yes
SmartRoads 2.0 must have good code quality. • The code written by us needs to have at least an 80% test coverage. • The code written by us needs to be fully and clearly documented.	*Must have*	Yes
Back end must be built in a modular way, such that new features can be added easily in the future.	*Must have*	Yes (Gateway)
Front end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.	*Must have*	Yes
Implement Route planner as a module/service.	*Could have*	No
For the following future modules/services a custom integration plan should be created. • Authentication. • User Profiling. • Route Planner. • Automatic Accident Detection. • Scenario Designer. • Traffic Light Analytics.	*Could have*	No
Implement the Authentication as a module/service.	*Could have*	No

Table E.4: List of requirements which aim to ease in-house development, along with their corresponding design goal and importance

Requirements to make it more configurable to customers	MoSCoW classification	Fulfilled?
Implement the User profiling as a module/service. • Add user profiles, with settings per profile. • Users should be able to temporarily change the settings of his/her user profile.	*Could have*	No

Table E.5: List of requirements which aim to make SmartRoads more configurable to customers of Scenwise, along with their corresponding design goal and importance



Course of events

Week	Date	Worked on	Deliverable
4.1	20-04 - 26-04	Meeting with client, orientation on technologies, code & possible solutions	
4.2	27-04 - 03-05	Setup project, research possible solutions in detail and start research report	Project plan
4.3	04-05 - 10-05	Perform measurement research & Set-up new architecture design	
4.4	11-05 - 17-05	Parser, set-up database, development map view, prototype first steps of the new architecture design	Research report
4.5	18-05 - 24-05	Develop API and start visualizing data feeds on map view	Decide which parts of the new architecture design we are going to implement.
4.6	25-06 - 31-05	Development faster replay functionality, visualize the entire road network of the Netherlands (speed and accidents)	First SIG code submission
4.7	01-06 - 07-06	further development	
4.8	08-06 - 14-06	Incorporate feedback from SIG & Finish faster replay functionality	
4.9	15-06 - 21-06	Start final report & Finish Architecture design	Second SIG code submission
4.10	22-06 - 28-06	Prepare presentation & Finish final report & Finishing up development	Final report & Architecture design
4.11	29-06 - 01-07	Finishing up development & Prepare presentation	Final presentation

Table F.1: The actual course of events, as opposed to the planning

G

Project Plan

SmartRoads 2.0

Project Plan

by

J.L. Buijnsters	4598733
D. Hofman	4688988
J.G.P. Klein Kranenbarg	4586387
C. el Moussaoui	4609395
K. Zheng	4698290

Project duration:	April 20, 2020 – June 28, 2020	
Supervisors:	Dr.ir. H. Wang,	TU Delft, BEP Coordinator
	Ir. O.W. Visser,	TU Delft, BEP Coordinator
	Dr.ir. B.H.M. Gerritsen	Coach
	Ir. K. F. Chan	Client

TU Delft

Contents

1 Company background **1**

2 Problem analysis **2**
2.1 Problem definition 2
2.2 Change of project definition 2
2.3 Requirements 2
2.4 Research plan 5

3 Approach **7**
3.1 Planning 7
3.2 Meetings 7
3.3 Work-style 7
3.3.1 Roles in the team 7
3.3.2 Communication types 7
3.3.3 Other services 8
3.4 Quality control 8
3.5 Client and TU coach 8
3.6 Agreements 8

Bibliography **11**

A ProjectForum Description **12**

B Internship Agreement **14**

1 Company background

Scenwise[1] is a small company with big dreams. Their mission is: Organizing all available data within the reach of the traffic management domain. They want to make this data uniform, accessible and usable. With this ambitious and inspiring goal, they help their clients to develop high-quality applications and services. They do this on behalf of their clients, through the secondment of their software development tracks. Scenwise has a large network of experienced professionals. They work together with their partners to carry out large assignments. Scenwise also develops innovative applications and services in collaboration with its business partners. An example of this is the ScenarioVerkenner® for the automation of control scenarios. This algorithm will soon be rolled out to the road authorities in North Holland.

[1] https://www.scenwise.com/

2 Problem analysis

2.1. Problem definition

Scenwise has developed the SmartRoads platform. This platform is used by its customers for monitoring and analyzing traffic situations. Scenwise intends to both solve currently existing performance issues, add more functionalities to SmartRoads to support a wider range of customers. Multiple similar web applications are currently being sold by Scenwise, with different extra features depending on the demands of the customer.

After the initial meeting with the client and subsequent meetings with both client and their developer, we were able to get a more detailed insight into the company. All of their solutions share the same base functionality, but all of them have been built from the ground up. This approach to software development is inefficient and indicates a lack of modularity in their software. This means they would greatly benefit from making their code-base modular and thus easily reusable. Currently, it is hard for them to reuse code because their applications are not modular, have little to no tests and are not properly documented.

The goal of this project is to create SmartRoads 2.0. This new platform has to have all features of SmartRoads 1.0, plus some extra features specified by Scenwise. More importantly, SmartRoads 2.0 needs to solve the more fundamental problems of SmartRoads 1.0. This means that the platform is designed with modularity as a core value, the code is tested and properly documented. This ensures that the code can easily be reused in the numerous web applications used by Scenwise and further improved by other developers.

2.2. Change of project definition

The original problem description on the ProjectForum (see appendix A) was already altered in an early stage of the project. At the start of the project, Scenwise supplied a list of features SmartRoads 2.0 should include. In the first week, it was already clear that simply adding these features to SmartRoads 1.0 lacked a research component, which is required by the TU Delft. The team then noticed more fundamental issues within Scenwise and started researching these issues. Furthermore, Scenwise turned out to already have a new version of SmartRoads 1.0 made by a BEP carried out last year. However, this version is not being used because some features are missing and the layout did not correspond with the previous application. The team intends to research the possibility of building further on top of this platform instead of SmartRoads 1.0, as it might turn out to have a superior code quality. After a Skype meeting, the client was convinced of the added value of this research.

2.3. Requirements

Requirements from Scenwise

Scenwise wants us to develop a new platform that incorporates both the features of the old SmartRoads and various new features.

SmartRoads has the following features:

- Display data from various sources concerning road usage.

- Perform calculations on the data and display these.
- Generate a replay.

The following new features have to be added:

- The new application has to have better performance in all aspects.
- Make updating the road configuration easier or automated, this includes calculating free-flow and capacity for the new road segments.
- User accounts with different access rights.
- An improved overview of the Situation messages.
- Add a tab Exceptional Situations that only displays abnormal data, that requires action from traffic regulators.
- Automatic accident detection not based on users reporting implemented into the Exceptional Situations tab.
- Implement GPS tracking of a phone app, which can continue when the GPS signal is lost temporarily.
- Implement the innovative visualization features found in Analytics.SmartRoads made by a previous BEP group.

Scenwise stresses that they want to see a working product at the end of the project, otherwise they consider it a failed project. This working product does not have to have all the aforementioned features, but at least the following features:

- Display data from various sources concerning road usage for the whole of the Netherlands within 1 minute.
- Perform calculations on the data and display these for the whole of the Netherlands within 1 minute.
- Generate a faster replay.

Requirements from TU Delft

The TU Delft requires a certain degree of Software engineering and a research component. The degree of software engineering is measured in four different areas according to the general guide [2], namely: Software development cycle execution, Proper use of development strategies, Quality assurance and justification and explanation of process and results.

To comply with the research component, we intend to not only research the provided requirements but also how we can develop our solution in such a way that it can contribute to more efficient software development at Scenwise as a whole. We chose to broaden the scope of our project in this way, because there is room for improvement in the said area, and the team agreed with Scenwise that this way the team can provide the most value for the company.

Final product requirements

After orientating we have concluded that the client needs a modular application, which can be easily extended. To achieve this we should prioritize some features over others and add other requirements besides the client his requirements that are needed to bring this project to a satisfying end. It also became apparent that some of the features requested by the client would not contribute to a modular and scalable application and thus will not be in the scope of this project. If we happen to complete our plan and still have time left, these features can be added. From this, we have compiled the final list of requirements for this project. We will use the MoSCoW-Method for clarity (must, should, could and won't-haves).

Must-haves

- SmartRoads 2.0 must be able to retrieve the following datastreams from The Nationale Databank Wegverkeersgegevens (NDW):
  - The NDW ”incidents” stream [1]
  - The NDW ”measurement” stream [2]
  - The NDW ”Wegwerkzaamheden” stream [3]
  - The NDW ”trafficspeed” stream [4]
  - The NDW ”traveltime” stream [5]
  - The NDW ”Matrixsignaalinformatie” stream [6]
- SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands.
- SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0.
  - Rendering of data on the map needs better performance e.g. retrieved and rendered within 1 minute.
- SmartRoads 2.0 must have good code quality.
  - The code written by us needs to have at least an 80% test coverage.
  - The code written by us needs to be clearly documented.
- Back-end must be built in a modular way, such that new features can be added easily in the future.
- Front-end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.
- Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.
- Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.
- Database retrieval of the needed traffic information is too slow. Resulting in:
  - The current replay function is too slow and higher performance is required.
  - The retrieval and processing of a specific amount of traffic data regarding road length and duration must be under a specific amount of time.

Should haves

- SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands (e.g. Waze[1]), next to the NDW data.
- SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads features:
  - Situational messages.
  - Implement analytics contour plot.
  - Implement analytics cloud plot.

[1] http://opendata.ndw.nu/incidents.xml.gz
[2] http://opendata.ndw.nu/measurement.xml.gz
[3] http://opendata.ndw.nu/wegwerkzaamheden.xml.gz
[4] http://opendata.ndw.nu/trafficspeed.xml.gz
[5] http://opendata.ndw.nu/traveltime.xml.gz
[6] http://opendata.ndw.nu/Matrixsignaalinformatie.xml.gz

- A new design should make the updates of configuration easier or even automatically. An improved design should be introduced to reduce manual handling and the chance of errors.
- For the following future modules/services a custom integration plan should be created.
  - Authentication.
  - User Profiling.
  - Route Planner.
  - Automatic Accident Detection.
  - Scenario Designer.
  - Traffic Light Analytics.

Could haves

- Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.
- Extra features situational messages.
  - Group and filter the situation messages per region, road and direction
  - Add the possibility to sort the messages in consecutive order w.r.t. the hectometer positions.
- Implement the Authentication module/service.
- Implement the User profiling module/service.
  - Add user profiles, with settings per profile.
  - Users should be able to temporarily change the settings of his/her user profile.
- Implement Route planner module/service.
- Show the number of free parking spots for parking garages in Amsterdam.

Won't-haves

- Tracking app
  - Users may accidentally stop the sharing of locations, solve this problem.
  - The system may lose track of the vehicle when GPS signal is lost (e.g. in a cell where no GPS coverage is, or in a tunnel). Create a solution for this.
  - Find out what is the maximal numbers of vehicle tracking that can be supported.
- Modules/services that won't be implemented.
  - Automatic Accident Detection.
  - Scenario Designer.
  - Traffic Light Analytics.

2.4. Research plan

Most of the requirements are related to the improvement of existing functionalities. To do so, we need to know the current causes of the need for improvement. Therefore we want to research the current systems in order to define a metric that should be able to determine whether we have met the requirements.

First, we want to identify the causes of the problems we want to solve. To do so we have to research the current state of the systems. Most of these problems consist of functionalities not meeting the expected performance. The main functionalities experiencing these problems are the map plotting and the replay functionality. We want to start by deciding on what tools to use for tracing the systems to find the causes for these insufficient performances.

Based on our meetings with both the client and the developer, we derived the front-end, the back-end and the database to be the possible causes for the performance issues the client is experiencing in his systems. Therefore, we want to profile these areas using the decided metrics and tools and then analyze the acquired data to identify the causes. Once these causes have been determined, we will decide on what feasible improvement rates for the performances of the different systems could be in our new design. Finally, based on these improvement rates, we will sharpen the related requirements to correspond to these rates and making it more clear on whether requirements are met at the end of the project.

The other problems we want to solve are the scalability and modularity of the systems. To solve these problems we want to start with determining metrics on how to measure these aspects. Once these metrics are determined, we will apply these metrics on the systems to know the scalability and modularity of the current systems. We will then apply a feasibility study on both of these aspects to determine if these aspects can be improved and to what extent. Finally, we will sharpen our requirements to correspond to the feasible improvement rate found in our research and use this data to propose a new architecture design for the current systems. A part of this architecture design is meant for clarification on how the client can migrate from their old product to their new product, but also to migrate to a more efficient project workflow.

3

Approach

3.1. Planning

Based on the requirements of the project we have made an estimation of the workload and made a planning accordingly in table 3.1. As it is still unclear in which week the final presentations will be, we added them to both weeks 4.10 and 4.11. Note that the deadline for the final report is dependent on the presentation, which means this deadline also can not be determined.

3.2. Meetings

Every day from Monday till Friday we have two fixed meetings with the group. The first one is at 9:00 clock in the morning. This meeting is meant to discuss the activities of everybody for the rest of the day. The second one is at the end of the project day at 16:45, here we can discuss the progress of the day and any questions which arise during the day. The product owner tries to meet with us every Friday, but because of time constraints, it could be necessary for this meeting to be skipped. Thus we have agreed that we will have this Friday meeting with the product owner at least once every two weeks. We also have weekly meetings with our TU coach on Thursdays.

3.3. Work-style

In any project, it is very important to agree on a work-style beforehand. The agile scrum methodology is used for the workflow. Our client Kin Fai Chan is the Product owner in this project. Daily scrums are done in every morning meeting. We have weekly sprints on Monday mornings, where we review our previous sprint, create a retrospective and make a sprint backlog/planning for the next sprint. The tool used for the scrum board is the GitLab issue board. In the scrum board we have the lists mentioned in table 3.2:

3.3.1. Roles in the team

The team roles are defined in table 3.3:

3.3.2. Communication types

- **Group**: the main way of communicating within the group is through Discord, where we have our aforementioned meetings. WhatsApp will be used for extra communication when members are not in the discord.

- **Client/Product Owner**: Email is used to formally communicate with our client and Skype is used for our meetings.

- **TU Coach**: Email and Mattermost are used to formally communicate with the coach and Skype is used for our meetings.

3.3.3. Other services

To ensure a smooth workflow GitHub is used as a development platform. All documents of the meetings and reports are recorded in a Google Drive folder.

3.4. Quality control

It is our goal to complete the project to the best of our ability and with the satisfaction of both the client and the TU Delft. To ensure we achieve this we have documented the project and our approach in detail in this project plan along with the agreements and conditions. To prevent conflict and ensure smooth collaboration in the future we will present this project plan to both the client and the coach. The coach will then approve of this plan. In this manner, there can be no doubt whether this project fulfills the expectations of both parties. Expectations and requirements can change, which is why we will need written confirmation of any future changes to our plan after it has been finalized.

To keep the client and coach updated with our progress, we will plan weekly meetings with both. The client and coach can let us know if anything is not according to the plan or their liking. If needed more meetings can be scheduled.

Quality will also be maintained within the team. This means we will be honest and transparent in our communication towards one another. Problems will be solved immediately as a group. Apart from team dynamics, there is also code quality. Our standard will be checked by each other before any code can be merged by adopting the agile way of working. Not only code style is important, but to be certain code does not break after future changes we have to implement automated tests. As a team, we have agreed that tests are just as important as code and thus we should aim for code coverage of at least 80% for branch, class and line coverage.

3.5. Client and TU coach

This project has two external parties involved, the TU Delft and Scenwise, both parties have supplied us with an advisor. The TU Delft provides us with a coach, Bart Gerritsen, with whom we will meet every week and who will offer us support in making sure the project follows the designated timeline. Scenwise provides us with Kin Fai Chan, who provides us with an open problem to which we have to provide a solution by scientific research. The team and Kin Fai Chan have signed an Internship Agreement, included in appendix B.

3.6. Agreements

We have set up some agreements within the team. We will work from Monday till Friday from 9:00 - 17:00. Everybody has to be online on our discord channel at the beginning of the day. And at the end of the day, we have a last meeting of the day at 16:45 also via discord. We will have a weekly scrum meeting on Monday. The meetings with the project owner will take place via Skype. We will take minutes of all the meetings, Jan is responsible for the minutes. All members of the group have signed a non-disclosure agreement to ensure confidential information of Scenwise. We agreed with the client that we don't have to deal with any privacy-related data. All members receive internship compensation and travel allowance conform to market prices. Therefore all members have filled in the payroll tax form. Additional to these documents we also have established an internship agreement with the client, included in appendix B.

Week	Date	Tasks	Deadline
4.1	20-04 - 26-04	Meeting with client, orientation on technologies, code & possible solutions	Deliverable: Project plan
4.2	27-04 - 03-05	Setup project, research possible solutions in detail and write research report	
4.3	04-05 - 10-05	Perform measurement research regarding front-end, back-end and database; Set-up new architecture design to a modular- or service-based architecture	Deliverable: Research report
4.4	11-05 - 17-05	NDW Parser, set-up database, development map view, prototype first steps of the new architecture design	
4.5	18-05 - 24-05	Develop API, start visualizing data feeds on map view, development faster replay functionality	Decide which parts of the new architecture design we are going to implement.
4.6	25-06 - 31-05	Continue development faster replay functionality, visualize the entire road network of the Netherlands (free-flow, speed and accidents)	Deliverable: First SIG code submission; Must-have functionalities should be implemented.
4.7	01-06 - 07-06	Incorporate feedback from SIG & further development	
4.8	08-06 - 14-06	Incorporate feedback from SIG & start final report	Deliverable: Second SIG code submission; Should-have functionalities should be implemented
4.9	15-06 - 21-06	Wrap up final product & finish report	Deliverable: Final report seven days before the presentation
4.10	22-06 - 28-06	Prepare presentation	Final presentation
4.11	29-06 - 05-07	Prepare presentation	Final presentation

Table 3.1: Planning per week

List name	Description
Product backlog	The entire backlog containing all items to be done
Sprint backlog	The backlog for the current sprint containing the items to be done for this sprint
In-progress	The items that are in progress
In-review	The items that need to be reviewed
Revision required	The items/pull requests that have been reviewed and should be revised
Approved	The items that have been approved and are ready to have a final Quality Assurance check after which they can be properly rounded up and merged
Finished	The finished items

Table 3.2: Scrum board lists

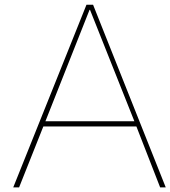
Role	Team member	Definition
Project-leader	Jasper	Responsible for meetings, proper discussions and team ambiance
Lead-communication	Kawin	Responsible for all communication within the team and with the client & coach
Scrum master	Chakir	Responsible for overall scrum process
Lead Testing	Daan	Responsible for test quality and coverage
Secretary	Jan	Responsible for the minutes of every meeting
Quality assurance responsible	Jasper	Responsible for quality of the product by making sure that every requirement is properly met without loss of quality
Front-end responsible	Chakir	Responsible for the front-end code quality
Back-end responsible	Jan	Responsible for the back-end code quality
CI & Git responsible	Daan	Responsible for Continuous Integration and proper GitHub usage

Table 3.3: Role division

Bibliography

[1] Google. Waze, 2020. URL `https://developers.google.com/waze`.

[2] *General Guide for TU Delft TI3806 Computer Science Bachelor Project (2020)*. TU Delft, Delft, 3 2020.



ProjectForum Description

Company Background
Scenwise B.V. is a company with a lot of experience in Data Science & 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 (e.g. traffic monitoring, data fusion, Big Data, Machine Learning)

Our customers are the Dutch Highway Agency (Rijkswaterstaat), Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, ITS system suppliers, Feyenoord Stadium and recently also the city of Edmonton in Canada.

Project description
Scenwise has developed the SmartRoads platform. SmartRoads is used by our customers to monitor and analyze traffic situations. For example:

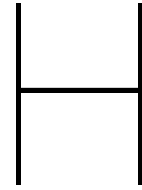
- Feyenoord Stadium uses SmartRoads to monitor the traffic during a day with a football match;
- Rijkswaterstaat uses SmartRoads for road-tests of Driving Automation.

Scenwise intends to extend the functionalities of SmartRoads to support a wide range of customers. For this reason, we start a new project to rebuild SmartRoads. The development will include:

1. Flexible web-interface supporting different maps layers (e.g. Google Maps, ESRI, OpenStreetmaps, Mapbox, etc.);
2. Big Data back-end incorporating a wide range of Open Data and proprietary data;
3. Real-time data analyze functionalities;
4. High-performance replay-function;
5. Interface with Decision Support Applications (e.g. Automatic Accident Warning);
6. Interface with mobile apps to provide traveler advice.

Within this project, a team of students will work together to conduct research, make design decisions, develop the application inclusive testing. The back-end should be a new big data platform using different data sources. The front-end should be web-based using modern graphical interfaces which different real-time graphics to provide adequate information to the end-users. This is an innovative project with a lot of technical challenges.

We are looking for enthusiastic students with skills and interest in database techniques, data processing, geographical information systems (GIS), visualization techniques and algorithms who are willing to take the challenge of developing a new platform to meet the upcoming requirements of our customers.



Research Report

SmartRoads 2.0

Research Report

by

J.L. Buijnsters
D. Hofman
J.G.P. Klein Kranenbarg
C. el Moussaoui
K. Zheng

Project duration:	April 20, 2020 – June 28, 2020	
Supervisors:	Dr.ir. H. Wang,	TU Delft, BEP Coordinator
	Ir. O.W. Visser,	TU Delft, BEP Coordinator
	Dr.ir. B.H.M. Gerritsen	Coach
	Ir. K. F. Chan	Client

TU Delft

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Shifting the focus . . . 1

2 Problem Analysis **3**
2.1 Scenwise . . . 3
2.2 The problems . . . 3
2.2.1 Scenwise problems. . . . 3
2.2.2 Customer problems . . . 4
2.2.3 Developer problems . . . 4
2.2.4 Maintenance problems . . . 4
2.3 Current solutions . . . 4
2.3.1 SmartRoads 1.0 . . . 5
2.3.2 Analytics.SmartRoads . . . 6
2.4 Integration problems . . . 7
2.5 SWOT-analysis . . . 7
2.6 Conclusion . . . 9

3 Research Approach **10**
3.1 Research questions . . . 10
3.2 Research method. . . . 11

4 Research Results **13**
4.1 API calls. . . . 13
4.2 Database . . . 13
4.3 Back end . . . 14
4.4 Front end . . . 14
4.5 Conclusion . . . 15
4.6 Requirements revised . . . 15

5 Design Goals **16**
5.1 Performance . . . 16
5.2 Maintainability . . . 16
5.2.1 Testing . . . 16
5.2.2 Documentation . . . 16
5.2.3 Modularity. . . . 16
5.3 Scalability . . . 16

6 Design Decisions **18**
6.1 Front end . . . 18
6.1.1 React . . . 18
6.1.2 AngularJS. . . . 18
6.1.3 Angular . . . 19
6.1.4 Conclusion . . . 19
6.2 Map . . . 19
6.2.1 Google Maps . . . 19
6.2.2 ArcGIS . . . 19
6.2.3 Mapbox . . . 19
6.3 Back end . . . 19
6.3.1 Migration . . . 20
6.3.2 Long-Term evolution design . . . 21

6.4 Data Storage . . . 22
6.4.1 Relational databases . . . 23
6.4.2 Non relational databases . . . 23
6.4.3 Using the file system . . . 23
6.4.4 Conclusion . . . 23

7 Success Criteria **24**
7.1 Performance . . . 24
7.2 Maintainability . . . 24
7.3 Scalability . . . 24

Glossary **27**

Acronyms **28**

A ProjectForum Description **29**

B Requirements **31**

C Research Results **35**
C.1 User Stories. . . . 35
C.1.1 Story 1 . . . 35
C.1.2 Story 2 . . . 35
C.1.3 Story 3 . . . 35
C.1.4 Story 4 . . . 35
C.1.5 Story 5 . . . 35
C.1.6 Story 6 . . . 35
C.2 API Research . . . 36
C.3 Database Research . . . 36
C.4 Back end Research . . . 36
C.5 Front end Research . . . 38
C.6 Current application vs Proposed solution . . . 39

D Revised Requirements **41**

1 Introduction

1.1. Overview

The first part of any new software project is the research phase. In this report, we will discuss the research we conducted regarding the problems Scenwise B.V is dealing with and the way we will try to solve them. We will first discuss the problem analysis in chapter 2. Then chapter 3 will categorize our requirements in specific research questions, and how we are going to research these requirements. The results of this research can be found in chapter 4, based on these results we refined our requirements. Next in chapter 5 the outline of the design goals we aim to achieve within this project are discussed. Based on the design goals we give detailed design decisions in chapter 6 as well as their justifications. And finally, we will set our success criteria for the requirements and design decisions in chapter 7 to be able to evaluate the success of this project.

1.2. Shifting the focus

Within the first days of the project, we found some larger, more fundamental issues about the way the company handled its software products. We decided that it would be more valuable for the company if we first tackle these bigger, more fundamental problems.

When we reviewed the original requirements our client gave us, we missed the scientific research part. We have scanned through the code bases in order to form an initial idea on how we want to create the newer version of SmartRoads 1.0. First of all we noticed a few flaws. An example would be the use of the GitHub repository. It is missing the proper tools in order to ensure a healthy workflow, e.g. continuous integration. When going through the back end code, we noticed an almost complete absence of tests and documentation. This is a big problem, because the software at Scenwise is being written by many different developers and or student teams. These developers and teams currently lose a considerable amount of time trying to understand the current code, let alone verifying the correctness.

Scenwise currently has multiple tools that share the same base functionality, but have some small unique features. For every new customer they essentially create a whole new application from scratch, even though the functionality overlaps significantly. Because their entire business is based on selling these applications, we think the company can greatly benefit from applying concepts like modularity, proper documentation and adding tests.

We are going to create a detailed analysis of the good things and the bad things, as we learned that another Bachelor End Projects (BEP's) from the past also worked an application for Scenwise, but was ultimately incompatible with SmartRoads 1.0. We want to make a detailed plan to ensure the good parts of the current projects can be migrated to the new system we will design. To ensure future projects can also be integrated with our new framework and to make it future proof, we will focus on scalability, modularity and quality of code. We think it is very important for the long term value that these fundamentals are thoroughly researched and implemented from the beginning of a software project.

With this in mind, we switched the focus of our project from just trying to finish as many tasks from our clients task list as possible, to finding a future proof solution to improve the workflow and efficiency of Scenwise's entire software stack.

2 Problem Analysis

2.1. Scenwise

Scenwise B.V.[1] is a company with experience in the data science and smart mobility domain. They work together with partners to develop software for domain of Traffic management (e.g. automatic incident detection, response plans, etc.), and Data science (e.g. traffic monitoring, data fusion, Big Data, Machine Learning). Their customers include: Rijkswaterstaat, Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, IT system suppliers, Feyenoord stadium and recently also the city of Edmonton in Canada.

They created a collection of tools, which are mainly meant for traffic managers or large venues (like Feyenoord stadium) that want to monitor traffic on the most important roads nearby. The main functions these tools offer are:

- Monitoring traffic density
- accident detection
- displaying matrix sign information
- scenario planning and response plans
- Visualizing traffic data in clear graphs
- Replay functionality: Looking at historical data

These functions are spread over multiple different applications, which are listed in section 2.3. Their main data-source is the publicly available data of the NDW.

2.2. The problems

In this section we will take a better look at the problems and requirements from the client. We will place the problems into four categories, depending on where the problems originate. The four categories are: Problems Scenwise is facing, Problems customers of Scenwise (e.g. Feyenoord, Rijkswaterstaat) are facing, Problems developers at Scenwise are facing and Problems with the software itself.

2.2.1. Scenwise problems

Scenwise has provided us a list with problems they are currently dealing with and features they would like to have implemented. This is thoroughly mapped in B. Besides the software problems, the client has given us a detailed explanation of how the company has developed from the beginning until now. The client has also explained the future they have in mind. At this moment in time the company is hosting multiple applications which are sold independently. Each application has also been built independently.

[1] https://www.scenwise.com/

Some big functionalities have been rebuild multiple times and are all in production at the same time in different code bases. The Scenario tool, for example, is such a tool with much potential and competitors are way behind. Next to the Scenario tool there is a tool in development by another student group which is about accident detection before they are reported by any road user. The future vision of the client is one application which combines each previously independently developed application, but one of the problems is that they all have a different back end. This architectural problem arose because of funding problems. Each developed tool creates funding and enables development for new tools, meaning there is little room for restructuring the architecture. In our meeting at the end of week 1 we explained to the client what was needed in order to enable such future sights. At this point the client is convinced that this is the moment to invest in a better architectural approach.

2.2.2. Customer problems

The customers of Scenwise have the problem of missing features and a poor performance of the overall system, but especially of the replay functionality in SmartRoads 1.0. We can also imagine that having to use multiple different applications for each task is far from ideal. A problem for a new customer from a new region is that there is not yet any historical data stored about that region. This means they will be missing some key information for traffic management.

2.2.3. Developer problems

The software of the client is currently developed by two part-time software developers. They primarily work on the SmartRoads 1.0 application, but also have separate research projects in development. One key problem is the lack of documentation and automated tests. The main reason of missing these are a lack of time and them having a low priority. The employers functional demands are mainly focused on the graphical usability, the performance of the front end and manually checking and calculating if the information shown in the application is correct. Besides the part-time developers, the client often uses students projects to develop new applications and functionalities. An example is later described in section 2.3. This workforce has to deliver a working product in a development time of around 10 weeks. The main problem of these work forces is that the decision between continuing development on one of the products of the client or starting a new project from scratch is quickly made, as it is easier to start from scratch instead of working on a code base without any documentation. For a new developer it is hard to know where to start developing in the software of our client.

2.2.4. Maintenance problems

A well maintained piece of software includes clear documentation and at least some degree of testing to make sure no code breaking changes are introduced when pushing to deployment [1]. The code of the tools created by other student projects are well tested and the documentation is clear. But, as stated before, the tests and documentations for SmartRoads 1.0 are lacking. This deteriorates the maintainability. Another problem of the software is the performance. The client is considering to upgrade the hardware of the server implying vertical scaling [2], which would not be beneficial. Vertical scaling can only be done to a certain extend, adding more power to one machine will eventually become too expensive.

2.3. Current solutions

Scenwise currently has multiple separate systems for traffic data analysis and traffic management scenarios. Despite all sharing the same base functionality, they have been implemented in a different manner. These systems are:

- SmartRoads 1.0
- Analytics.SmartRoads (made by a previous BEP group [3])
- Scenario Designer 1.0
- Scenario Designer 2.0 (made by a previous BEP group [4])
- Incident Detection

- SmartRoads 3D
- Vlog viewer
- Regelaanpak GUI
- VRI-PZH & VRI routes-pzh
- IWAP/RRIP
- Datafusie Amsterdam

Scenwise never made use of Analytics.SmartRoads and its features. Instead, Scenwise decided to built further upon their own version, adapting a small number of features from Analytics.SmartRoads. We will look at their respective design decisions to have a better understanding as to what their fundamental differences are.

2.3.1. SmartRoads 1.0

SmartRoads 1.0 is where the company started four years ago. This product has been updated ever since, however, adding more and more features over the years has not been favourable to the documentation, performance and maintainability. Despite all this it is still the solution that is being used and sold due to the amount of features that attract customers.

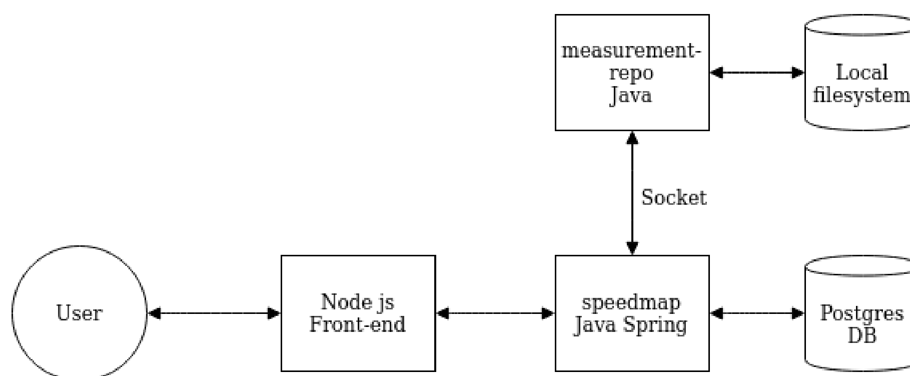


Figure 2.1: SmartRoads 1.0 current architecture

Used frameworks

To provide an overview of the system it is important to know what frameworks are used. The frameworks can be divided into the front end, back end, Application Programming Interface (API) and database. The front end is built using AngularJS [5]. The back end and the API utilise Java Spring and the database is a Postgres database [6], [7]. However an effort has been made to enhance the performance of the replay functionality. This is done in a separate repository, the measurement-repository. This server is used by the back end to get historical data. This data is not stored in a database, but on the local filesytem. The communication between these two servers is established with a socket [8].

Data structure

Another important aspect in understanding the architecture is uncovering the data structure of the application. This can be seen when we look at the different tables in the database:

- drip_information, where information about Dynamisch Route-informatiepanelen (DRIP's) is saved.
- drip_information_latest, where the latest DRIP information is stored.
- drip_locations, the locations where the DRIP's are located.
- hectopunten, information about mile markers are saved.
- measurement_characteristic, has information about measurements.

- measurement_site, with information about the site where measurements are made.
- measurement_site_critical_density, stores the critical density of measurement sites.
- measurement_site_group, contains descriptions of measurement sites.
- measurement_site_group_sites, assigns measurement sites to groups.
- msi, has information on matrix signs.
- status_construction, contains the construction status messages.
- status_construction_text, contains the texts for the construction status messages.
- status_coords, contains the coordinates of status messages.
- status_messages, contains the status messages.

Features
With this software architecture SmartRoads 1.0 is able to provide the following features, but only with so much lag it becomes almost unusable.

- Retrieve, process, visualize and store data from different data feeds for the regions of Rotterdam and Amsterdam. The following data is supported:
  - Free flow
  - Traffic intensity
  - Parking spots only supported in Amsterdam
  - Accidents
  - Matrix signs
  - Construction sites
- Replay traffic situations in the past for Rotterdam and Amsterdam.

2.3.2. Analytics.SmartRoads
As stated before, just a small number of features of Analytics.SmartRoads were adopted by Scenwise due to fundamental differences. As Analytics.SmartRoads has better performance, we will look at their design decisions to have an insight of their approach in contrast to SmartRoads 1.0.

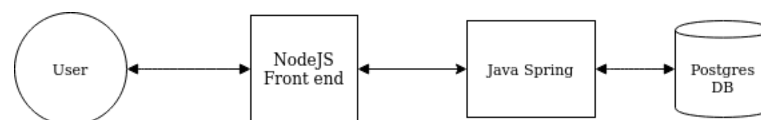


Figure 2.2: Analytics.SmartRoads current architecture

Used frameworks
As for the frameworks and API's they use we will separate them into three categories; front end, back end and communication between them. In the front end no actual framework is used since the User Interface (UI) is relatively small. Instead of native JavaScript however, they have chosen to use TypeScript for added type safety. For the back end they decided to go with Java Spring [6] as this made it easier to make use of legacy code. The connection between these two was done using the RESTful API to ensure separation between them.

Data structure
PostgreSQL is used as a database [7]. The data structure itself is kept relatively small and simple, as can be seen in the following tables :

- measurement
- measurement_sites
- msi_locations
- msi
- status_messages

Features
Features Analytics.SmartRoads provides are :

- Creating routes
- Countour plot
- Point cloud
- Cover entirety of the Netherlands
- Matrix signs
- Status messages
- Live traffic intensity

2.4. Integration problems

As can be seen in table 2.1 there are differences in some aspects of the software. Since there is a notable difference in the size of functionality between the applications, there also be a difference in front end. Due to the smaller scope of Analytics.SmartRoads they decided to work without a framework, but this would less favourable to do for an application as big as SmartRoads 1.0. There is also a notable difference in the size of data that is stored. This resulted in SmartRoads 1.0 opting for a local file system whereas Analytics.SmartRoads did not need to, thanks to the smaller size of data stored. Finally, Analytics.SmartRoads makes use of docker which is by no means implemented by SmartRoads 1.0.

	SmartRoads 1.0	**Analytics.SmartRoads**
Back end framework	Java Spring	Java Spring
Front end framework	AngularJs 1.6	No framework
Front end hosted by	Node.JS	Node.JS
Data storage	PostgreSQL and local files system	PostgreSQL
Communication back- and front end	RESTful API	RESTful API
Replay functionally	Yes, using local file system for data storage	No
Uses docker	No	Yes

Table 2.1: Comparison between SmartRoads 1.0 and Analytics.SmartRoads

2.5. SWOT-analysis

To get a better understanding of what lays in the scope of this project, a Strengths Weaknesses Opportunities and Threats analysis (SWOT-analysis) of Scenwise was made (see figure 2.2). ScenWise has two functional back-ends with functionalities applicable to SmartRoads 2.0. They have a clear view of what features need to be implemented in the near future and lots of possible features for the future. However, there is room for improvement in the workflow at Scenwise. SmartRoads 1.0 has an unintuitive front-end design. Their current solution has a large technical debt, caused mostly by the lack

of tests, the lack of documentation and the lack of a modular structure. This makes their applications difficult to maintain. We aim to solve these weaknesses by creating a modular and easily maintainable structure. This will allow combining functionalities from various other back-ends created by ScenWise opening up new possibilities such as creating new tailored products for custom clients using the existing codebase. This maintainable approach will also improve the efficiency with which new and existing developers can work. Being able to solve the performance issues will make SmartRoads available for customers anywhere located in the Netherlands. This could even create a snowball effect allowing ScenWise to extend to other countries as well. Some of the threats outside of the scope of the project would be the disability of ScenWise to follow-up on our proposed guidelines, like an improved workflow, taken to the extreme would be even an unsuccessful migration to SmartRoads 2.0. ScenWise has to keep in mind the threat that an enabling technology becomes unusable, for example, a pricing change, this is the case with Google Maps in SmartRoads 1.0.

Internal Strengths:	Weaknesses:
Lots of functionalities available from the existing back-ends of SmartRoads 1.0 and Analytics.SmartRoads. ScenWise has a clear view of what features they want to add now and in the future	Workflow Code Quality → not modular, no extensive testing, missing documentations Periodical manual operations needed in order to keep the software up to date SmartRoads 1.0 performance issues: Replay Only able to display information about a small part of the roads of the Netherlands. Some unintuitive elements in front-end design
External Opportunities:	Threats:
Extending to nearby countries, by combining more data feeds Adding functionalities of the other existing application back-ends with a modular approach Higher efficiency of new and existing developers Creating new tailored products for custom clients using the existing codebase Being available for customers anywhere located in the Netherlands	SmartRoads 2.0 will be just another Analytics.SmartRoads Fundamental enabling technologies become unusable Stop focus on the more fundamental problems of ScenWise: By unsuccessfully remaining a healthy Workflow By unsuccessfully remaining the code quality, which gradually makes the code unmanageable

Table 2.2: SWOT-analysis

2.6. Conclusion

After performing the problem analysis the team has concluded that software at Scenwise can be improved. Lack of performance, maintainability and scalability are the main problems we identified within existing code at Scenwise.

3 Research Approach

These requirements can be seen in appendix B as follows:

- Requirements which aim to increase the performance of SmartRoads (table D.1)
- Requirements which are about adding new features to SmartRoads (table D.2)
- Requirements which aim to ease in-house development (table D.3)
- Requirements which aim to create more configurability to customers (table D.4)

The requirements made at the beginning of the project were based on meetings with the client and their developers. They had suspicions about most of the limitations of SmartRoads 1.0, however, these were just suspicions and based on assumptions. In the section 2.6 the problem was broken down into three main problems. To find the root causes of these problems we decided to conduct our research with three main research questions and subquestions that are given in section 3.1. If we can find the answers to these questions and subquestions, we have found the root causes. These root causes can then be categorized for each of the main problems and solutions can be found accordingly. With the solutions, we intend to further refine our requirements so we can be sure that we will solve the problems that exist in SmartRoads 1.0.

An overview of our research approach will be given in fig. 3.1.

3.1. Research questions

- How can performance be improved?
  - Is the performance bottleneck located in the API?
  - Is the performance bottleneck located in the database?
  - Is the performance bottleneck located in the back end?
  - Is the performance bottleneck located in the front end?
- How can maintainability be improved?
  - Is the maintainability bottleneck located in the API?
  - Is the maintainability bottleneck located in the database?
  - Is the maintainability bottleneck located in the back end?
  - Is the maintainability bottleneck located in the front end?
- How can scalability be improved?
  - Is the scalability bottleneck located in the API?
  - Is the scalability bottleneck located in the database?
  - Is the scalability bottleneck located in the back end?
  - Is the scalability bottleneck located in the front end?

3.2. Research method

To conduct the research scientifically and systematically we have defined user stories of the most basic functionalities of the application. The idea was to measure and analyze the performance, scalability and maintainability of these functionalities since the performance was one of the main problems in SmartRoads 1.0. These measurements could then be divided into different categories so it would become clear what the root causes were. These categories are: API, database, back end and front end. To measure the API performance we used Postman, for database performance we used PgBadger, a log analyzer for Postgres databases [7] and the back end was measured using The API yaml fileJava's built-in Java mission control. Lastly, we used the Google Chrome Runtime Performance Analyzer to analyze the front end.

However, it quickly turned out that these user stories did not matter, as all the data is loaded as soon as the application is opened and that the only action triggering any data retrieval is the replay function. These user stories can still be of use when validating our end product.

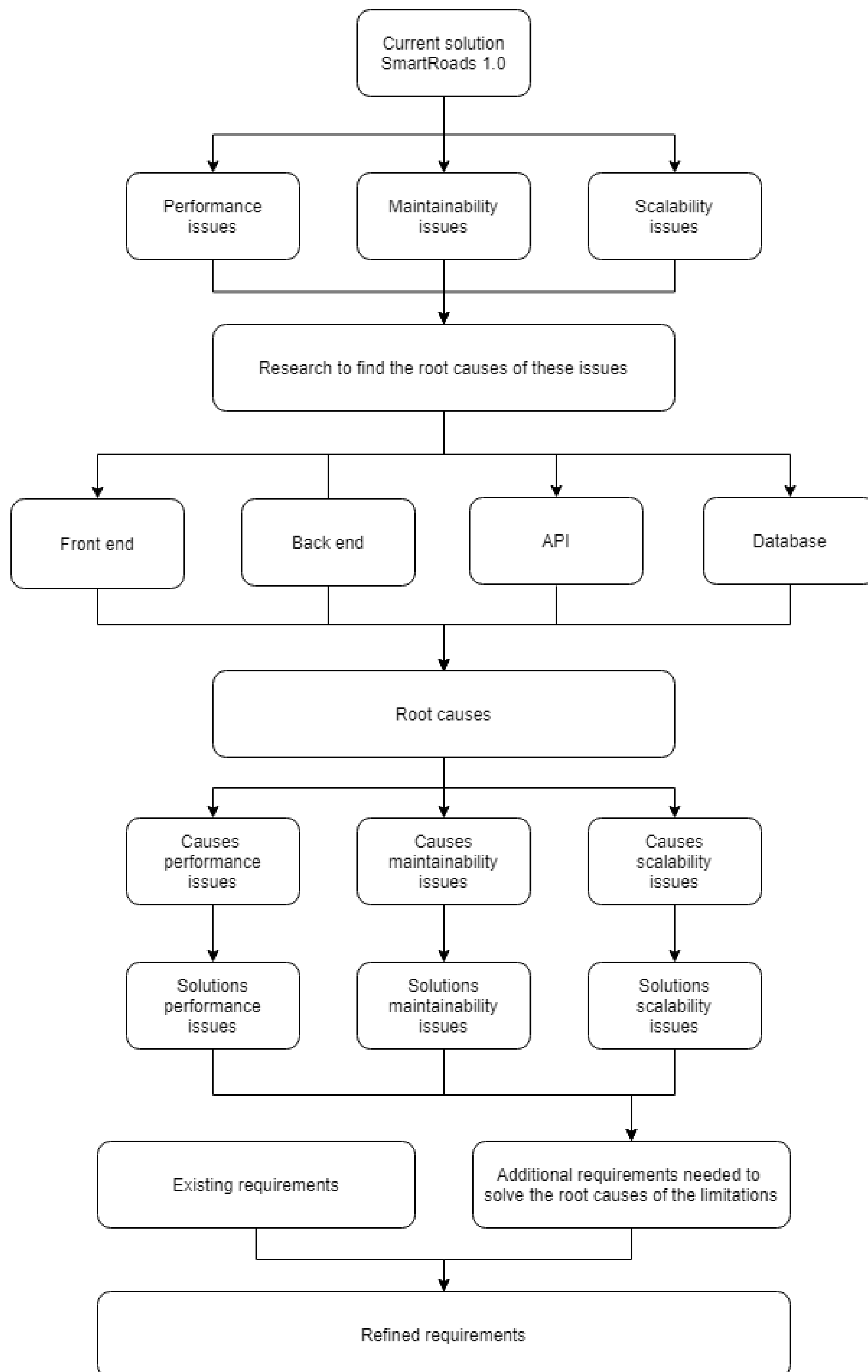
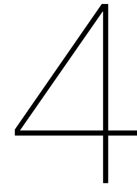


Figure 3.1: Research approach



Research Results

The research yielded too much data to be included and talked about. The research can be broken down into different parts such that each parts can be analyzed and a conclusion can be reached. These conclusions are supported by visuals obtained using the various tools used in the research.

4.1. API calls

In table C.1 we have an overview of the different API calls made by SmartRoads 1.0 complete with the response times and sizes. From this table we can conclude that the response times for most API calls are less than half a second. Only one API call really stands out and could possible cause a performance bottleneck. There were several issues concerning maintainability of the API. The most important being the complete lack of tests and documentation. As for the scalability aspect, there was nothing remarkable.

4.2. Database

4.3. Back end

The measurements of the back end were obtained during the execution of the user stories in appendix C.1. The backend op SmartRoads 1.0 consists of two different back ends, one main back end and one to write and read only one kind of data locally.

We will start with the second one, the 'measurement repository' since it is the simple one and it stores one kind of measurement, the average speed that is measured on that road segment. It does not need to be said that storing measurements locally is a bad design choice and if we look at the numbers in fig. C.1 we see that it performs worse than a database as it reads less data in more time and that is also it's only task as demonstrated in fig. C.2. This means that this separate back end is not beneficial to the performance of the application. Then we have to take into account that the code for this system has a small README, no further comments and no tests, which means that it is absolutely not maintainable. The fact that this stores data locally inherently means that this approach is not scalable.

Then there is the main backend where most of the parser for the NDW data is located, the other parsers are not included in this repository. This is the part where the calculations are done and every other part of the application is connected to. When we look at the method profiler in fig. C.3 we can see that almost 90% of the code leads back to the TimerThread. After closer inspection we saw that this is the parser which runs on a timer. As this parser is largely the same as the one built by a previous BEP group and their overhead was 600ms and 2000 ms on startup. Since there is not much other computationally heavy functions except some average computing, we can conclude that the performance of the back end is sufficient. Maintainability wise we can say that the code is not easily understandable, since there is almost no documentation and comments. There are almost no tests and the code style is very inconsistent. As far as scalability is concerned there is nothing notable.

4.4. Front end

The front end performance was measured during the execution of the user stories in appendix C.1. They were then analyzed using Chrome's performance analyzer. This allowed us to obtain the diagram in fig. C.4 when starting the frontend and logging in, fig. C.5 when we do user story 4 and fig. C.6 when the replay function is tested. We see that the bulk of the time is consumed by scripting followed by idle and then system. The same goes for all user stories and this can be explained. Most of the time there is nothing to do for the system as there is only new data once per minute, so it sits idle. Then once per minute there is new data and this data needs to be plotted and that is where we see scripting spike. This includes everything from animations to function calls in AngularJS [5]. Then there is 'system' category, but that is just the 'other' category. The difference between fig. C.6 and the other two is that the replay is longer and thus idles more of the duration of the recording and scipts less.

Combine this knowledge with the observation that the clock begins to stutter and skip seconds whenever a new minute hits and thus new data is coming in. This means that the whole front end is rerendered when new data is coming in. Add the fact that all information is requested from the back end at startup and it is certain that the whole road network is updated every time even if it is not visible to the user. Then there is the fact that whenever the map is zoomed in, out or dragged, the whole frontend renders again. These are all things that only make the UI feel unresponsive and seem to be the cause of the performance bottlenecks, since the front end should only be scripting 1 second of the minute and not almost half the time.

The level of maintainability is the same as the other parts, there are no tests, almost no documentation or comments. This means that the code is hard to understand due to the lack of documentation and comments. If there would be any desire to change anything at all, it is likely that a lot of code would break due to the lack of tests. These issues make if inefficient, time wise and cost wise, to maintain this part of the application.

The front end scores low on scalability as well, as additional features can not be added as components.

4.5. Conclusion

In conclusion, the answer to the first research question: 'How can the performance be improved?', is that the application performance suffers mainly from the way the front end handles the rendering on the map and the retrieval of certain data. To improve this the focus should be to look at a new front end that only renders changed components and this should have the highest priority. Then there were several smaller improvements in the API calls and database structure.

The second research question is: 'How can the maintainability be improved?'. The maintainability of all components is very low due to lack of documentation and tests. Naming is vague for most components and code style is inconsistent. These issues propagate throughout the entirety of the application. It seems easy to improve on these points, so why has this not already been done? For Scenwise it has more value to work on new features than to improve on what already works and since the company has to remain economically beneficial, this has simply been ignored. This is where this project can step in, as the team is not torn between meeting financial obligations and making a sound application, which would be financially more beneficial to the company in the long run.

The last research question is: 'How can scalability be improved?'. There are several big issues with scalability in some parts of the system. The database should not be filled manually and the measurement repository should not save measurements in a local filesystem. It is not easy to add features to the front end. It comes down to making different design choices for populating the database and using a different framework for the front end.

4.6. Requirements revised

Based on the results of our research we have identified the root problems of the current Scenwise approach as can be found in fig. C.7. For each problem we have devised a solution in table C.2 and are going to revise our requirements accordingly. In the end our product should have the structure proposed in fig. C.8. The main idea behind this architecture is that we will retain all current features and even incorporate the features of Analytics.SmartRoads, but can easily replace any under performing feature such as a replay function by a new one.

For all new components that the team is going to build there has to be documentation and tests. A new front end should be built that requests data in a smarter way, so we should request as little data as possible and only what is actually needed. Then it should adopt a new rendering method so that we only re-render what has changed. This new front end should be better scalable by using components that can be easily reused. The back end will be a combination of SmartRoads 1.0, Analytics.SmartRoads and brand new back ends for specific functions. This architecture does still have the problems found in the back end, but is a step in the right direction since this enables us to rebuild features and then replace them until the old back ends is not needed anymore. A new database will be setup, but used besides the old databases until it is ready to take over all functionality. This new database will not be a local file system and will not require manual data collection and storage. The database schema will have to be thought through to minimize double data.

These new insights will be used to refine the requirements which can be found in appendix D. That will be our final list of requirements.

5 Design Goals

In this section, we will elaborate on the main design goals of this project. After performing extensive research we have come to the conclusion that software at Scenwise can be improved, by having clear goals and focus. Lack thereof is the main problem we identified within existing code at Scenwise. During the development of our application, these are the main values that we will always keep in mind while making decisions. The main design goals are: performance, maintainability and scalability.

5.1. Performance

Performance is an important design goal for this project. Our solution must provide mostly the same features as the current solution, but work in a different, more efficient way. The main reason our client has requested a new system is the decreasing performance of the current system, which at this point has declined so much it has become a substantial problem. This means that to make this project a success, the performance has to be substantially better than the current application. The way we will measure this performance increase is mentioned in chapter 7.

5.2. Maintainability

An important aspect of any software engineering system is for it to be maintainable, that is, it is relatively easy to replace parts and update pieces of software in order to keep the performance at an all time high. It also reduces a big portion of the life cycle cost of a system [9]. This is achieved by testing, documenting and keeping it all modular.

5.2.1. Testing

Testing allows for code changes to happen without breaking the system unconsciously, ensuring high code quality and persistency.

5.2.2. Documentation

Documentation allows for new developers to know what functionality a piece of code brings. It is also for clarity of functionality of code when it is changed.

5.2.3. Modularity

In order to keep up with this change, the concept of modularity has been introduced. Modularity enables a project to safely replace a part of a whole to enhance its performance, and all that without too much complication. That is why we will make the base of this project a modular core so that it can benefit from the ever increasing amount of changes in technology.

5.3. Scalability

Scenwise wants to expand its customer base, and for this they want to be able to offer their service for any customer based anywhere in the Netherlands. In the current solution there is only one version

of the application. Due to the fact that displaying traffic information of the entire Netherlands does not have the desired performance, they only implemented two regions, Amsterdam and Rotterdam. The regions for which traffic information is displayed are hard coded. This is not a scalable approach as it makes it very tedious to expand beyond the regions of Rotterdam and Amsterdam, because new functionality needs to be added if a new customer wants to see a new region. Furthermore, every time a new region is implemented on the SmartRoads 1.0 map, the performance declines. This means that it is very hard for Scenwise to scale up their application for new customers.

The current design of SmartRoads 1.0 does not scale with the amount of users that use the solution at the same time. This is currently not yet a problem, because there are not many users. The performance is affected by the amount of processing needed to serve multiple users simultaneously, which means that it will decline even further when there are more users. We want to enable Scenwise to grow, so the application should be able to handle an increase in the numbers of concurrent users.

These reasons have led to the conclusion that we need to build smartroads 2.0 with scalability as one of our design goals.

6 Design Decisions

In this chapter the design decisions made for the new solution (SmartRoads 2.0) are explained.

6.1. Front end

According to stackoverflow.com [10] the most used front end tools for user interfaces are ReactJS, Angular and AngularJS. The team looked at the most popular frameworks, because they have a bigger community and more resources to use for development.They were researched separately and decided on which one will be used depending on several criteria. These criteria include development speed, performance, testability etc.

6.1.1. React

React [11] is a JavaScript library which is specific to building user interfaces. It is also component based so it allows for any modularity within the framework. Updating any component in the application is fast due to the use of virtual DOM by React. It is lightweight and allows for third party libraries to be used. However, there are drawbacks to it. Finding the right library and continuously updating them can turn out to be quite an obstacle and time consuming. Potential future mobile application development is supported by React.

Documentation

Being lightweight and depending on third party libraries, the documentation of React is minimal, but does not imply that there is no documentation as the libraries used are sure to have documentation.

Testing

Testing is be done by testing different components with different tools, as there is no single way to test everything.

6.1.2. AngularJS

AngularJS [5] is already used for the SmartRoads 1.0 application, so if desirable it is possible to reuse some of the soon to be legacy code. It also makes use of dependency injection which allows for services to be used within the front end components.

Documentation

AngularJS has everything well documented and explained.

Testing

Testing can be done using Karma [12] or Jasmine [13] as a tool for unit and end-to-end testing purposes.

6.1.3. Angular

Angular [14] provides TypeScript instead of plain JavaScript that AngularJS uses. TypeScript allows for cleaner code as it eases the process of finding common errors made in programming. It also ensures high code quality which is important for maintainability. In contrast to AngularJs, Angular makes use of Angular CLI, which is used for generating components and services, ultimately enhancing development speed. Like AngularJS, Angular also makes use of dependency injection but does in a more clear way. It also has a component-based architecture to ensure modularity within the front end. A drawback to developing with Angular is that it has a steep learning curve.

Documentation

Contrary to React, Angular has a lot of documentation which makes development easier, but it is under development, so it might not be up to date at all times.

Testing

Like in AngularJS, testing tools like Karma and Jasmine can be used. Like, React, Angular also supports mobile application development.

6.1.4. Conclusion

Since Angular is the more improved version of AngularJS and adds extra possibilities it comes down to Angular and React. To be more specific it comes down to the development speed of Angular vs the performance speed of React. Performance is very important to the new application and having a steep learning curve takes away development speed for Angular. Therefore, it was decided to use ReactJS for the front end.

6.2. Map

In this section the choice for the map that is going to be used is discussed. The following three mapping platforms were considered: Google Maps [15], ArcGIS [16] and Mapbox [17].

6.2.1. Google Maps

The Google Maps does not give much breathing space in terms of free usage of the API. That is why it was quickly dropped.

6.2.2. ArcGIS

ArcGIS was, next to Mapbox, one of the API's preferred by the client, but for different reasons. ArcGIS is seen as a more strategic choice as they are potential to the company. ArcGIS is also somewhat vague about their pricing, as they do give out free credits in the beginning, but do not specify what a credit entails. This does not give a good impression as to how long it is possible to use it without having to pay for it.

6.2.3. Mapbox

Mapbox is preferred by the client, because of the possibilities is provides. It has a more clear definition of a free trial. The free trial itself allows us to make up to 200.000 API calls and 50.000 map loads which should be enough for our development. It has, like ArcGIS, library support for ReactJS and good documentation. It also adds customizability to the maps used which could prove very useful for our use case.

Due to the of advantages Mapbox has over Google Maps and ArcGIS in terms of possibilities, pricing and practicality in developing, it was decided to go with Mapbox as our mapping platform.

6.3. Back end

We want to create the back end in a way such that future functionalities can be added separately from existing code. We, however, have the constraint of wanting to ensure that SmartRoads 1.0 will be able to systematically migrate to our new version. To do so it is important for the back end to choose a software design which allows for this new system to meet these requirements.

6.3.1. Migration

The following points are important when migrating to the new system:

There is currently a lot of functionality in the back end of SmartRoads 1.0 and Analytics.SmartRoads. It would not only be inefficient to start over from scratch again, but also put ScenWise again in a dire situation yet again, where they should invest in order to migrate.

The API's of the two back ends are not uniform, hence one of the reasons why they never migrated Analytics.SmartRoads in their current system. This could be due to the fact that there currently are no specific rules on how the API should look and act. In order to set these rules and thus create a uniform API to communicate with the front end and possibly a future mobile application, an API Gateway will be created.

This API Gateway should serve all data endpoints which the back end of SmartRoads 1.0 supports. It would only be a gateway to the already existing end-points. This means that the newly developed front end will have the needed data available to support the already existing functionality of SmartRoads 1.0.

Likewise the current deployed SmartRoads 1.0 front end should be able to function via this control API as well. The endpoints of Analytics.SmartRoads should also be reachable from the control API. Some endpoints, which have better performance and the same functionality, might even replace the old ones. Such a gateway is shown in figure 6.1.

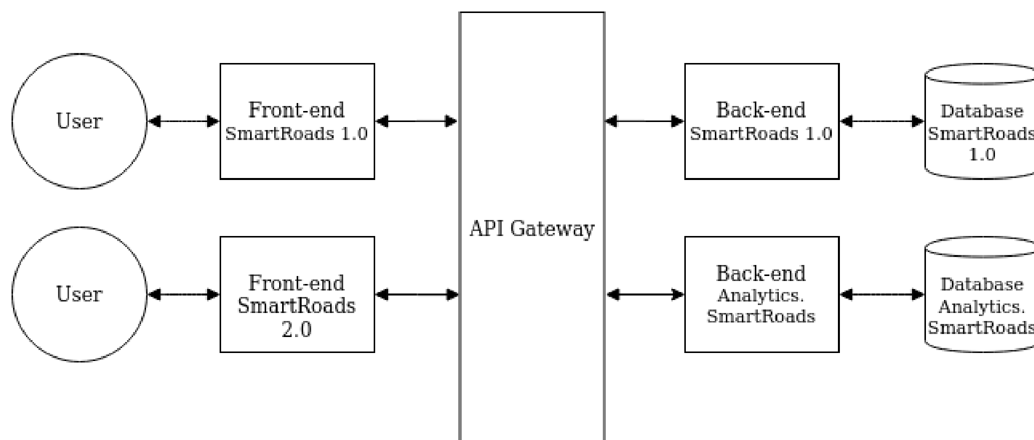


Figure 6.1: API Gateway set-up

After this setup, the strangler pattern [18] can be used. Which means gradually implementing functionality from the ”legacy code” into independent new services. Once such functionality is done developing, the control API can update it's public endpoint accordingly if necessary. This means that endpoints in our API Gateway will not be using the endpoints of the ”old back end”, but of the new services. Meaning ScenWise can gradually migrate to a new back end, with the back end complexity abstracted away from the front ends. This process would be similar for new functionalities, but the control API will then need new endpoints for them.

Data is currently stored multiple times on different databases, the new architecture should not be that redundant. This issue will be solved by having centralized databases. One such database would for example be the NDW database. A NDW parser service would fill this database, and all services which need NDW data can get it from that centralized NDW database. It could be beneficial to give future new implemented data streams their own centralized database, as this keeps the system decoupled. In figure 6.2 you can see how the replay function would be separated from the back end of SmartRoads 1.0 into its own service and using the new centralized NDW database.

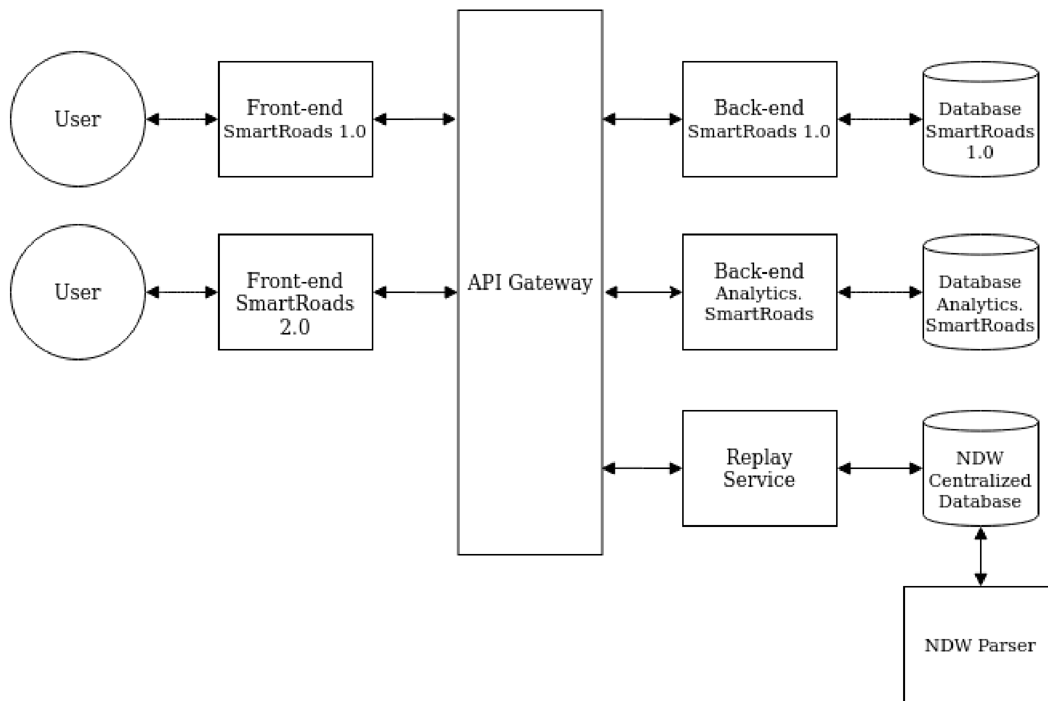


Figure 6.2: Replay service with centralized database

The migration of the front end should be pretty straightforward. First the new SmartRoads 2.0 front end should be developed to a point where all the minimum functionalities for a viable replacement product are adhered. At that point the new front-end can be served to the user and the front end migration is complete.

To make sure ScenWise will have the knowledge to follow all these migration steps a Long-term evolution (LTE) design document will be created as a deliverable for ScenWise. This document will, next to the aforementioned steps, also include guidelines regarding code quality and testing, version control, an explanation regarding the communication between the components in the system and an explanation for new developers on how they can add their own services to the system.

6.3.2. Long-Term evolution design

After the steps in section 6.3.1 are taken, the team will should focus on a future architecture design for the back end.

Layered Architecture

An layered architectural [19] approach was considerd, but as this architecture is in place right now and it does not seem to work, this is not a good direction to go in for the future.

Microkernel Architecture

A Microkernel Architecture [19] has also been considered. As the concept of this plug-in module could be a good way for new developers to add to the system. However, enabling the migration from the current situation into a Microkernel Architecture is probably very hard, as the core functionality should be in the core system. This would mean it needs to be decided what the core functionality is going to be, most likely this will be a lot of functionality from the current codebase. As discussed this code base lacks documentation and testing, forcing us to begin from scratch. Still, the idea of minimum communication between plug-ins is something to keep in mind.

Service Orientated Architecture

A Service Orientated Architecture [20] seems to fall right in place. With the separately created services for improving the performance, the first steps towards such an architecture have already been taken. Next to that, it is mostly in harmony with our design goals. Compared to the monolith, maintainability

of the services is high because services are smaller code bases functioning independently from other services. When a service is in need of an update or even replacement, there is a limited amount of code one has to dig through in order to understand how the service works. Performance can be improved as the best performing stack for each service can be chosen, however this does come with some extra communication costs between each service. CI/CD is easily implementable with services and thus the testing goal is satisfied. Documentation wise, guidelines for how new services can be added should be created. Modularity is almost a given with the loosely coupled services.

From this architecture, the decision can also still be made to further split services into a micro-services architecture [18]. Though currently, this seems a bit too ambitious for the scale of ScenWise.

Communication services
For the communication between services, an Event-Driven architecture [19] could be great, because of the live data streams. Even so, there is nothing like that in place at the moment. The possibility of adding in event-driven messages between services and maybe extend them to the user will be explored. It is, however, the question whether this is in the scope of our project regarding the requirements given. For now, the traditional, request/response messaging will be used.

At the moment both versions of the back end are created using Java Spring [6]. The only part which is not easily replaceable in the aforementioned new system is the API Gateway. There are a few technologies which are considered for this API Gateway. The first one is from scratch, it should be fairly simple to forward requests to specific endpoints. The API Gateway is a very good place for some extra functionalities, it would be nice to be able to use functionalities from gateway frameworks. An open-source API Gateway which was considered is Kong [21]. Kong, however, does not support aggregating data, which could prove useful at some point in our project. Another framework is the Spring Cloud Gateway [22]. This Spring Project gives the ability to provide security, monitoring/metrics, resiliency and is non-blocking. Above all that it also supports long-lived connections like web-sockets, which can be useful should event messages be implemented in the future. It is also built using Spring which is, as stated before, well known in ScenWise. Zuul2 [23] was also considerd. It is similar to Spring Cloud Gateway, but, as support for Spring Cloud Gateway is larger and it is easier to set up, went with Spring Cloud Gateway was chosen.

For the new services that are going to be used, it needs to be checked whether there is a potential performance upgrade in switching from Java Spring to another stack.

6.4. Data Storage

This section will discuss the database technology to use in the solution. There are two main types of databases, relational and non-relational. Relational databases like MySQL [24] and PostgreSQL [7] represent and store data in tables and rows. They're based on a branch of algebraic set theory known as relational algebra. Meanwhile, non-relational databases like MongoDB [25] represent data in collections of JSON documents. Another option the team considered is simply storing the historical data on the file system.

To make a well informed choice of database technology it is important to analyse the data it needs to store. A previous bep project measured that the main data source, The NDW "measurement" stream[1], has a write load of about 16000 new records per minute [3]. This was confirmed, and 16000-17000 new records per minute is an accurate measurement.

This datastream is needed for a minimal viable version of the replay function. Replaying more distinct data types, like matrix signs or accidents requires more data to be stored. A record in this data stream consists of a UUID, a timestamp, an id corresponding to the location of the measurement, an finally a speed value and a flow value. The client requires that detailed data needs to be stored for two weeks. Detailed data means one data point for every minute. This means that about 322.560.000 measurement records will be stored for the first two weeks. After the first two weeks, the data is stored for an additional two weeks, but this data can be less detailed. The data storage needed can be reduced by aggregating this data. The first two weeks have a data point every minute. For the second

[1] http://opendata.ndw.nu/measurement.xml.gz

two weeks, this interval can be increased to five minutes. This would already mean 5 times less data needs to be stored, while still being able to accurately analyse the traffic situation development.

6.4.1. Relational databases

Relational databases include: Oracle Database 12c [26], PostgreSQL [7], mySQL [24] and IBM Db2[27]. Because the client does not have budget for a paid database service, only free to use database solutions were considered. This excludes Oracle Database 12c and IBM Db2.

PostgreSQL and MySQL are both free to use open-source databases. They are both very popular choices, though PostgreSQL's popularity is still rising. PostgreSQL is more feature rich. It is scalable and can handle terabytes of data. It can handle the large dataflow that is required [28]. It has many predefined ready to use functions. It also supports JSON, which is nice to have. PostgreSQL also adheres to the ACID (atomicity, consistency, isolation, durability) principle, an important principle in database technology. Another advantage is that Scenwise currently uses PostgreSQL as database solution, which means their developers are familiar with it.

6.4.2. Non relational databases

A popular non-relational database is MongoDB. MongoDB is a non-relational database that stores its data in JSON like files. Non-relational database do not use SQL. This is a disadvantage, because PostgreSQL is currently being used and SQL queries have already been written. Many of these queries contain joins. In non-relational databases, there are no joins like there would be in relational databases. When a join is needed, it requires two or more queries which are manually joined in the code. This creates a risk for mistakes and bugs.

Another disadvantage is that MongoDB does not automatically treat operations as transactions in the same way a relational database does. Instead, you must manually choose to create a transaction and then manually verify it, manually commit it or roll it back. The documentation on the MongoDB site warns that, without taking some potentially time-consuming precautions, the success or failure of a database operation cannot be atomic. This means that part of a query succeeds, while another part fails. This leaves the database in a faulty state. This violates the ACID (atomicity, consistency, isolation, durability) principle.

6.4.3. Using the file system

The main function of the database in SmartRoads 1.0 is to enable the replay function. To this end it needs to store data in chronological order, only to later retrieve it in the same order. It does not need to modify the stored records in any way. Because of these simple requirements, the team considered to use the file system instead of a database. Though, this would create some issues. Databases solve numerous issues that exist when using the file system as a database. They comply to a set of rules, called the ACID principle (atomicity, consistency, isolation, durability). Because database software ensures that transactions always happen according to the principle, many issues are solved, like file lock issues. These kind of issues can arise when multiple users try to read or write to a file at the same time. Another disadvantage is that the structure of the data cannot be easily modified. If, in the future, the client wishes to extend the replay functionality and store additional data, this can not easily been done.

6.4.4. Conclusion

The team chose to use PostgreSQL as means of data storage, because of the superior performance over other database solutions. It is also know that it can handle the amount of data needed for SmartRoads, because this is specified in the PostGreSQL limitations [28]. In specific cases, storing and retrieving files directly via the file system can be faster than using PostgreSQL. However, this introduces new problems and risks (mentioned in section 6.4.3), which do not outweigh the performance gained. To ensure quick retrieval of data, which is needed for the replay function, an index needs to be created on both the timestamp and location field.

Success Criteria

To conclude the research, we decided to sharpen some of the requirements based on the results of the research. We first identified the current issues in the system after which we determined what could be able to fix these issues. But in order to confirm whether we actually solved the problem after finishing our project we needed a way to determine this. Therefore we decided to determine success criteria and some corresponding metrics allowing us to determine certain success criteria for the current issues. And because of the following, we are now able to determine whether we were actually able to solve the issues at the end of the project.

7.1. Performance

First of all, we wanted to determine a proper metric to measure the performance of our application in order to make it clear whether certain requirements were met. Based on our results, we noticed that most of the performance issues were caused in the front end of the system. Which is why we decided to use the built-in chrome profiler that allows us to monitor the costs of the different areas of the front end. Based on the result gained from these profilers, we noticed that the most time was taken by retrieval and rendering of the data on the map. Therefore we wanted to set the success criteria for the performance to ”For every user story appendix C.1, when performed, the time taken by scripting and rendering should be at least 40% less in the new system when compared to the current system”.

7.2. Maintainability

Secondly, we initially wanted to determine a metric for the maintainability of the systems as well. However, as we concluded in section 4.5, the maintainability of the components of the current system is incredibly low. Therefore we decided to not define a metric for the maintainability itself, but made requirements concerning the code quality. For which we focus on two areas, documentation and testing. For the documentation, we set the requirement to ”All components (folders, modules, services, APIs, classes and components) should contain documentation and/or a readme”. For testing, we decided to set 80% coverage as the success criteria.

7.3. Scalability

We also wanted to define a certain metric to define the scalability of a system. However, just like the maintainability, the current scalability of the system is incredibly low and is not designed to scale for larger areas and additional modules. Due to the current state of the system and the amount of time we have for the implementation, we therefore decided to not define a metric for these requirements. And because we most likely won’t have enough time to implement most of the modules, we decided that we wanted to define the success criteria to be integration plans of planned modules.

Bibliography

[1] P. David Coward, “A review of software testing”, *Information and Software Technology*, vol. 30, pp. 189–198, Apr. 1988. (visited on 10/21/2019).

[2] C.-Y. Liu, M.-R. Shie, Y.-F. Lee, Y.-C. Lin, and K.-C. Lai, “Vertical/horizontal resource scaling mechanism for federated clouds”, in *2014 International Conference on Information Science & Applications (ICISA)*, IEEE, 2014, pp. 1–4.

[3] J. Smit, M. van Niekerk, R. Oosterbaan, D. van Gelder, and S. Tromer, “Developing a platform for traffic data analysis”, Also available as `http://resolver.tudelft.nl/uuid:1f39c7d1-7cbe-4985-a032-5ea71d5daa49`, Bachelor’s thesis, TU Delft, Delft, the Netherlands, Jun. 2019.

[4] F. Bredius, T. Naber, B. Tjiong, and V. Leroy, “Smart traffic management system”, Also available as `http://resolver.tudelft.nl/uuid:e54b67cf-0593-42e1-a07c-3e072b71276e`, Bachelor’s thesis, TU Delft, Delft, the Netherlands, Feb. 2020.

[5] AngularJs, *Angularjs api*, Accessed: 2020-5-5, 2020. [Online]. Available: `https://angularjs.org/`.

[6] *Spring framework documentation*, Accessed: 2020-4-30. [Online]. Available: `https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html`.

[7] *Postgres documentation*, Accessed: 2020-4-30. [Online]. Available: `https://www.postgresql.org/docs/`.

[8] *Java socket*, Accessed: 2020-5-12. [Online]. Available: `https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html`.

[9] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi, “Why is it important to measure maintainability and what are the best ways to do it?”, in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 377–378.

[10] Anonymous, *Stack overflow developer survey 2019*, Accessed: 2020-5-5, 2019. [Online]. Available: `https://insights.stackoverflow.com/survey/2019#technology`.

[11] ReactJS, *A javascript library for building user interfaces*, Accessed: 2020-5-5, 2020. [Online]. Available: `https://reactjs.org/`.

[12] Karma, *Karma*, Accessed: 2020-5-6, 2020. [Online]. Available: `https://karma-runner.github.io/latest/index.html`.

[13] Jasmine, *Jasmine: Behaviour-driven javascript*, Accessed: 2020-5-6, 2020. [Online]. Available: `https://jasmine.github.io/`.

[14] Angular, *Angular features*, Accessed: 2020-5-5, 2020. [Online]. Available: `https://angular.io/features`.

[15] *The google-maps platform*, Accessed: 2020-5-8, 2020. [Online]. Available: `https://developers.google.com/maps/documentation`.

[16] *Mapping and analysis: Location intelligence for everyone*, Accessed: 2020-5-8, 2020.

[17] *Mapbox gl js api*, Accessed: 2020-5-8, 2020.

[18] S. Newman, *Building microservices*, Feb. 2015. [Online]. Available: `https://samnewman.io/books/building_microservices/`.

[19] M. Richards, *Software architecture patterns by mark richards*, Feb. 2015. [Online]. Available: `https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/`.

[20] *Service orientated architecture patterns*, Accessed: 2020-5-7. [Online]. Available: `https://patterns.arcitura.com/soa-patterns`.

[21] *Kong gateway*, Accessed: 2020-5-8. [Online]. Available: https://konghq.com/kong/.

[22] *Spring cloud gateway*, Accessed: 2020-5-8. [Online]. Available: https://spring.io/projects/spring-cloud-gateway.

[23] *Zuul*, Accessed: 2020-5-8. [Online]. Available: https://github.com/Netflix/zuul/wiki.

[24] O. Corporation, *Mysql*, Accessed: 2020-5-5, 2020. [Online]. Available: https://www.mysql.com/.

[25] I. MongoDB, *Mongodb*, Accessed: 2020-5-5, 2020. [Online]. Available: https://www.mongodb.com/.

[26] O. Corporation, *Oracle*, Accessed: 2020-5-5, 2020. [Online]. Available: https://www.oracle.com/nl/database/.

[27] IBM, *Ibm db2*, Accessed: 2020-5-5, 2020. [Online]. Available: https://www.ibm.com/analytics/db2.

[28] R. Stones and N. Matthew, *Beginning Databases with PostgreSQL: From Novice to Professional*. 2006, ch. Appendix A. [Online]. Available: https://link.springer.com/content/pdf/bbm%5C%3A978-1-4302-0018-5%5C%2F1.pdf.

[29] Google, *Waze*, Accessed: 2020-5-5, 2020. [Online]. Available: https://developers.google.com/waze.

Glossary

ACID stands for: atomicity, consistency, isolation, durability. These are principles used in database software.. 23

Analytics.SmartRoads The application developed by a BEP group[1] in 2019 to improve on SmartRoads 1.0. 4–7, 15, 20, 36

back end Relating to or denoting the part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data. 1, 4–7, 10, 11, 13–15, 19–22, 34, 37, 44

continuous integration Tools for a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration can then be verified by an automated build and automated tests. 1

database A structured set of data held in a computer, especially one that is accessible in various ways. 5, 7, 10, 11, 13–15, 20–23, 27

framework An abstraction in which software providing generic functionality can be selectively changed by additional user-written code. 1, 5–7, 15, 18, 22

free flow When traffic can continue without hindrance. 6

front end Relating to or denoting the part of a computer system or application with which the user interacts directly. 4–7, 10, 11, 14, 15, 18–21, 24, 34, 38, 40, 42, 44

life cycle cost The cost of a software system over its life cycle, from Development to Entry into Service, all the way to Disposal. 16

matrix sign A digital traffic sign above a highway to inform drivers. Also known as Matrix Signaalgever (MS). 3, 6, 7

repository A central location in which data is stored and managed. 1, 5, 13–15

Rijkswaterstaat The Dutch Highway Agency. 3

SmartRoads 1.0 An application developed by Scenwise for traffic engineers to monitor the traffic situation. 1, 4–7, 10, 11, 13–15, 17–20, 23, 27, 36

software stack A set of software subsystems or components needed to create a complete platform such that no additional software is needed to support applications. 2

[1] `https://repository.tudelft.nl/islandora/object/uuid%3A1f39c7d1-7cbe-4985-a032-5ea71d5daa49?collection=education`

Acronyms

API Application Programming Interface. 5–7, 10, 13, 19, 20, 22, 36

BEP Bachelor End Project also known as the TU Delft Computer Science Bachelor Project. 1, 4, 14

DRIP Dynamisch Route-informatiepaneel. 5

JSON JavaScript Object Notation. 23

LTE Long-term evolution. 21

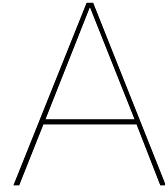
MS Matrix Signaalgever. 27

NDW Nationale Databank Wegverkeersgegevens. 3, 13, 14, 20

SWOT-analysis Strengths Weaknesses Opportunities and Threats analysis. 7, 8

UI User Interface. 6

UUID Universally Unique Identifier. 22



ProjectForum Description

Company Background
Scenwise B.V. is company with a lots of experience in Data Science & 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 (e.g. traffic monitoring, data fusion, Big Data, Machine Learning)

Our customers are the Dutch Highway Agency (Rijkswaterstaat), Nationale Databank Wegverkeersgegevens (NDW), provinces, large cities, ITS system suppliers, Feyenoord stadium and recently also the city of Edmonton in Canada.

Project description
Scenwise has developed the SmartRoads platform. SmartRoads is used by our customers for monitoring and analyse traffic situations. For example:

- Feyenoord Stadium uses SmartRoads to monitor the traffic during a day with football match;
- Rijkswaterstaat uses SmartRoads for road-tests of Driving Automation.

Scenwise intends to extend the functionalities of SmartRoads to support a wide range of customers. For this reason, we start a new project to rebuild SmartRoads. The development will include:

1. Flexible web-interface supporting different maps layers (e.g. Google Maps, ESRI, OpenStreetmaps, Mapbox, etc.);
2. Big Data back-end incorporating a wide range of Open Data and proprietary data;
3. Real-time data analyse functionalities;
4. High performance replay-function;
5. Interface with Decision Support Applications (e.g. Automatic Accident Warning);
6. Interface with mobile apps to provide traveller advises.

Within this project, a team of students will work together to conduct research, make design decisions, develop the application inclusive testing. The back-end should be a new big data platform using different data sources. The front-end should be web-based using modern graphical interfaces which different real-time graphics to provide adequate information to the end-users. This is an innovative project with a lots of technical challenges.

We are looking for enthusiastic students with skills and interest in database techniques, data processing, geographical information systems (GIS), visualization techniques and algorithms who are willing to take the challenge of developing a new platform to meet the upcoming requirements of our customers.

Scenwise will provide detailed information for making design decisions and guidance in making both the technical designs and the graphical interfaces designs. Setting up test scenario's for software acceptance is also a part of the project. Since this is a new development, there are enough rooms for the project team to make design decisions. If the project team is familiar with Java Spring (back-end), PostgreSQL, Python, React or Angular (front-end), then they may re-use part of the current source code.

B
Requirements

Requirements to increase performance	**Design Goal**	**MoSCoW classification**
SmartRoads 2.0 must be able to retrieve the following datastreams from The Nationale Databank Wegverkeersgegevens (NDW): • The NDW ”incidents” stream [1] • The NDW ”measurement” stream [2] • The NDW ”Wegwerkzaamheden” stream [3] • The NDW ”trafficspeed” stream [4] • The NDW ”traveltime” stream [5] • The NDW ”Matrixsignaalinformatie” stream [6]		*Must have*
The SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0. • Rendering of data on the map needs better performance e.g. retrieved and rendered within 1 minute.	*Performance*	*Must have*
Database retrieval of the needed traffic information is too slow. Resulting in: • The current replay function is too slow and higher performance is required. • The retrieval and processing of a specific amount of traffic data regarding road length and duration must be under a specific amount of time.	*Performance*	*Must have*
SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads features: • Situational messages. • Implement analytics contour plot. • Implement analytics cloud plot.		*Should have*

Table B.1: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance

Requirements for new features	Design Goal	MoSCoW classification
A new design should make the updates of the road-network configuration file easier or even automatic. An improved design should be introduced to reduce manual handling and the chance of errors.		*Should have*
SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands	*Performance & Scalability*	*Must have*
SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands (e.g. Waze [29]), next to the NDW data.		*Should have*
Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.		*Should have*

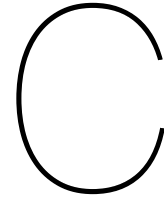
Table B.2: List of requirements which are about implementing new features, along with their corresponding design goal and importance

Requirements for in-house development	Design Goal	MoSCoW classification
Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.	*Maintainability*	*Must have*
Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.	*Maintainability*	*Must have*
SmartRoads 2.0 must have good code quality. • The code written by us needs to have at least an 80% test coverage. • The code written by us needs to be fully and clearly documented.	*Maintainability*	*Must have*
Back end must be built in a modular way, such that new features can be added easily in the future.	*Maintainability*	*Must have*
Front end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.	*Maintainability*	*Must have*
Implement Route planner as a module/service.	*Maintainability*	*Could have*
For the following future modules/services a custom integration plan should be created. • Authentication. • User Profiling. • Route Planner. • Automatic Accident Detection. • Scenario Designer. • Traffic Light Analytics.		*Could have*
Implement the Authentication as a module/service.	*Maintainability*	*Could have*

Table B.3: List of requirements which aim to ease in-house development, along with their corresponding design goal and importance

Requirements to make it more configurable to customers	Design Goal	MoSCoW classification
Implement the User profiling as a module/service. • Add user profiles, with settings per profile. • Users should be able to temporarily change the settings of his/her user profile.	*Maintainability*	*Could have*

Table B.4: List of requirements which aim to make SmartRoads more configurable to customers of Scenwise, along with their corresponding design goal and importance



Research Results

C.1. User Stories

C.1.1. Story 1

As a traffic controller at Feyenoord, I want to see the traffic flow in the region of Rotterdam as colors on road segments, so that I can streamline the arrival and departure of soccer teams, supporters and emergency services.

C.1.2. Story 2

As a traffic controller at Feyenoord, I want to see the maintenance sites in the region of Rotterdam, so that I can streamline the arrival and departure of soccer teams, supporters and emergency services.

C.1.3. Story 3

As a traffic controller at Feyenoord, I want to see the accidents in the region of Rotterdam, so that I can streamline the arrival and departure of soccer teams, supporters and emergency services.

C.1.4. Story 4

As a traffic controller at Feyenoord, I want to see the traffic flow on 3 road segments clockwise around Feyenoord over the last 24 hours, so that I can streamline the arrival and departure of soccer teams, supporters and emergency services.

C.1.5. Story 5

As a traffic controller at Feyenoord, I want to see the matrix sign A15L 65.25 above the road (Zuidoost rotterdam), so that I can streamline the arrival and departure of soccer teams, supporters and emergency services.

C.1.6. Story 6

As a traffic controller at Feyenoord, I want to see a replay of the region of Rotterdam from the prior day, so that I can evaluate our methods to streamline the arrival and departure of soccer teams, supporters and emergency services.

C.2. API Research

HTTP request method	API call	Response size	SmartRoads 1.0 time	Analytics.SmartRoads time
POST	activate-telpunten	206B	15ms	
POST	compute-critical-density	205B	16ms	
PUT	show-monica	177B	19ms	
GET	drip-all	568.23KB	139ms	
GET	status	268.72KB	458ms	
GET	freeflow-grt	138.53KB	43ms	
GET	init-map-data	60.2MB	1217ms	
GET	intensity-history	13.06KB	81ms	
GET	msi	3.19MB	609ms	
GET	msi-relations	13.22MB	2630ms	
GET	parking-availability	13.78KB	16ms	
GET	car-data	260B	19ms	
GET	traffic-speed	139.47KB	465ms	
GET	traveltime-history	20.55KB	57ms	
GET	traveltimes	136.15KB	632ms	
GET	traveltimes-avg-history	38.79KB	1804ms	
POST	roads/traject	37.01MB		3.59s
POST	roads/point	16.79MB		1215ms
GET	drip-date/date	575.21KB	46s	
GET	msi?date=	3.41MB	706ms	
GET	traveltimes?date=	137.03KB	886ms	
GET	status?date=	351.51 KB	601ms	
GET	traffic-speed?date=	139.47KB	2.3s	
POST	roads/traject?date=	36.97MB		3.87s
POST	roads/point?date=	16.82MB		1331ms

Table C.1: API calls and their responses

C.3. Database Research

C.4. Back end Research

Path	Total Read Time	Total Bytes Read	Read Count
E:\Studie\BEP\Code\Old repos\speedmap-develop\measure_repo_data\2020-05-10\data.dat	5 min 14 s 950 ms	1,10 MB	66
E:\Studie\BEP\Code\Old repos\speedmap-develop\measure_repo_data\2020-05-11\data.dat	3 min 44 s 306 ms	474,75 kB	36

Figure C.1: File reads by the measurement repository

Hot Methods

Filter Column Stack Trace

Stack Trace	Sample Count	Percentage
com.scenwise.measurement_repository.server.formats.FileFormatV2.readMeasurement(ByteBuffer)	21	36,21%
com.oracle.jrockit.jfr.DurationEvent.write()	4	6,90%
java.util.TimerThread.mainLoop()	3	5,17%
com.oracle.jrockit.jfr.InstantEvent.write()	3	5,17%
sun.rmi.transport.DGCImpl.checkLeases()	2	3,45%
java.nio.DirectByteBuffer.put(int, byte)	2	3,45%
oracle.jrockit.jfr.VMJFR.getThreadBuffer(int)	2	3,45%
com.scenwise.measurement_repository.server.repository.Repository.readMeasurements(MeasurementsRequest)	2	3,45%
java.io.DataInputStream.readByte()	1	1,72%
java.nio.DirectByteBuffer.getLong(int)	1	1,72%
java.lang.ThreadLocal$ThreadLocalMap.access$000(ThreadLocal$ThreadLocalMap, ThreadLocal)	1	1,72%
java.nio.DirectByteBuffer.putInt(long, int)	1	1,72%
java.nio.DirectByteBuffer.putLong(long, long)	1	1,72%
com.oracle.jrockit.jfr.InstantEvent.commit()	1	1,72%
java.nio.Bits.swap(long)	1	1,72%
com.oracle.jrockit.jfr.InstantEvent.shouldWrite()	1	1,72%
com.scenwise.measurement_repository.server.formats.FileFormatV2.readBlock(ByteBuffer, FilterCollection, List, int)	1	1,72%
java.lang.AbstractStringBuilder.ensureCapacityInternal(int)	1	1,72%
java.util.concurrent.ConcurrentHashMap.get(Object)	1	1,72%
java.nio.DirectByteBuffer.<init>(long, int)	1	1,72%
com.scenwise.measurement_repository.shared.filters.FilterCollection.satisfies(MeasurementEntry)	1	1,72%
java.nio.DirectByteBuffer.putLong(int, long)	1	1,72%
java.util.HashMap.computeIfAbsent(Object, Function)	1	1,72%
java.lang.Integer.equals(Object)	1	1,72%
java.nio.DirectByteBuffer.putLong(long)	1	1,72%
java.lang.ThreadLocal$ThreadLocalMap.getEntryAfterMiss(ThreadLocal, int, ThreadLocal$ThreadLocalMap$Entry)	1	1,72%
com.oracle.jrockit.jfr.EventToken.isEnabled()	1	1,72%

Figure C.2: Hot methods in the measurement repository

Method Profiler

Filter Column Stack Trace

Stack Trace	Sample Count	Percentage
java.util.TimerThread.run()	3.511	89,70%
java.lang.Thread.run()	228	5,83%
~ UNCLASSIFIABLE ~.()	166	4,24%
java.util.TimerThread.mainLoop()	7	0,18%
org.apache.tomcat.util.net.NioBlockingSelector$BlockPoller.run()	2	0,05%

Figure C.3: Method profiler of the main back end

C.5. Front end Research

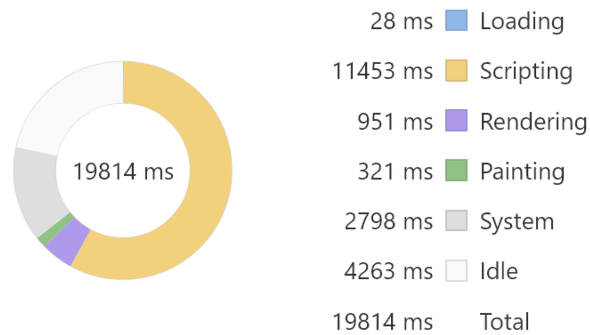


Figure C.4: Summary of starting the front end

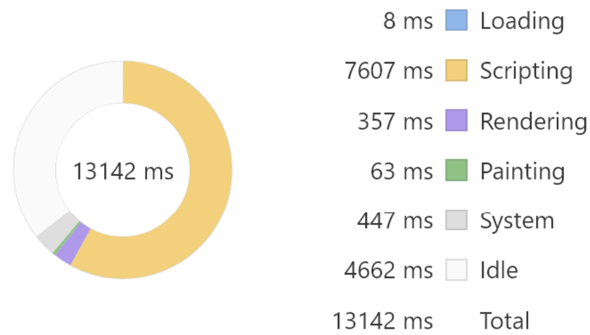


Figure C.5: Summary of selecting 3 roads

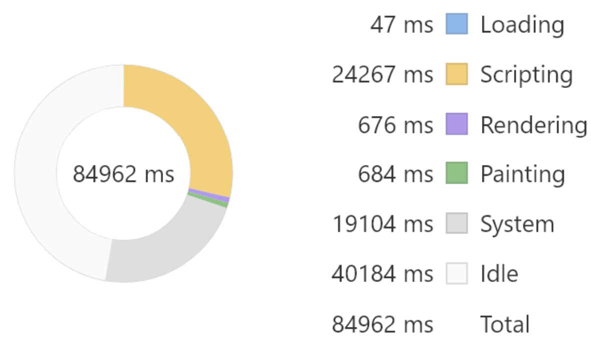


Figure C.6: Summary of the replay function

C.6. Current application vs Proposed solution

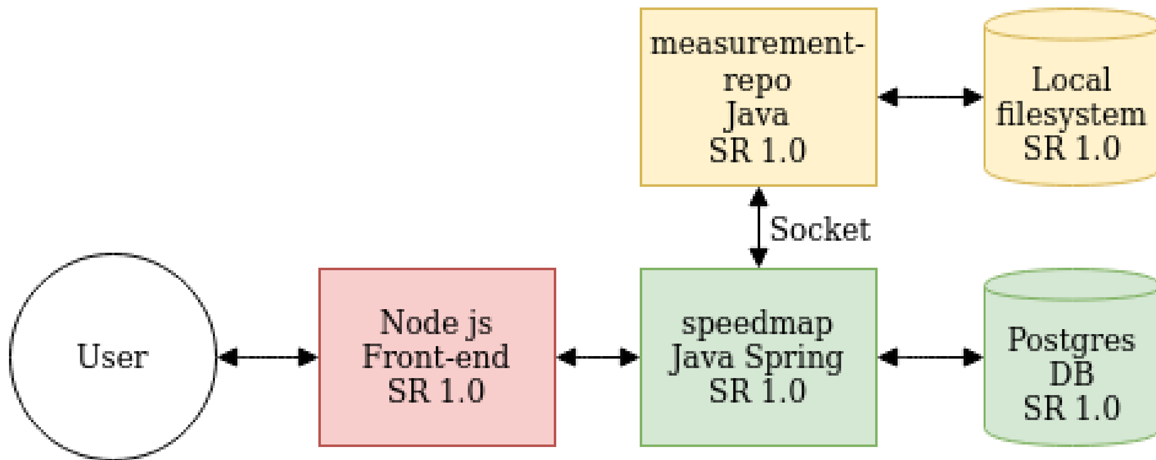


Figure C.7: Overview of the current application

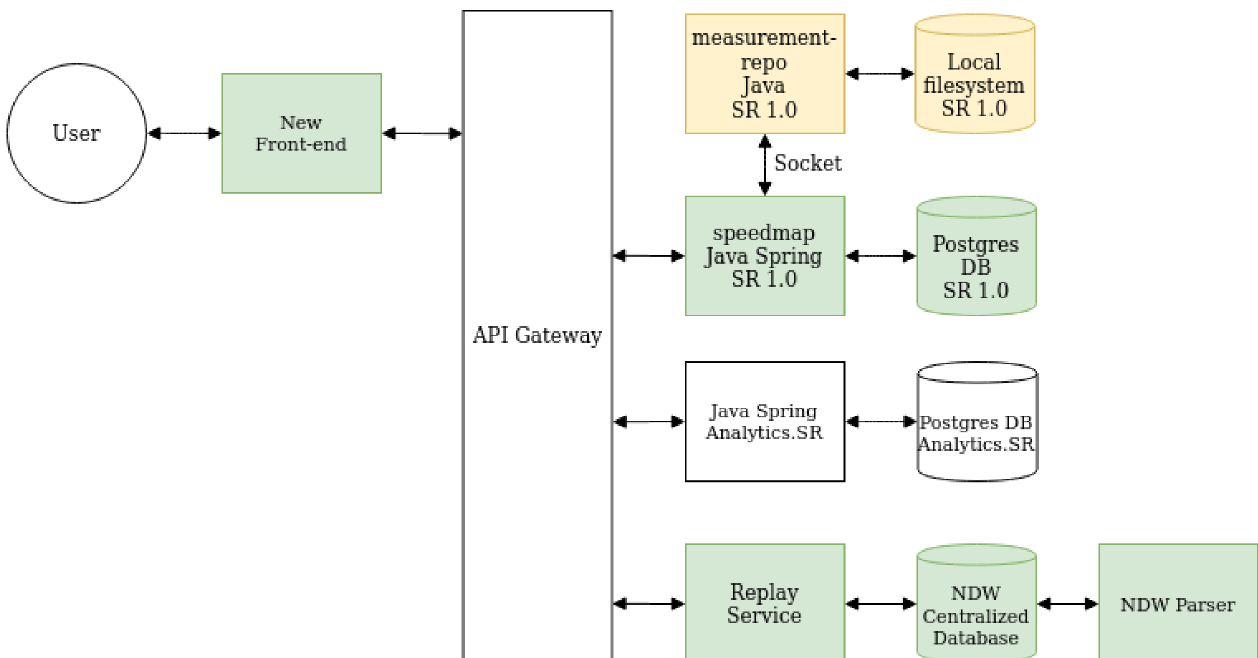
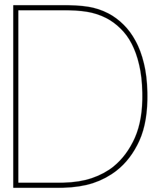


Figure C.8: Overview of our proposed solution

Problems in the current application	Proposed solution
Lack of documentation	All new code will have documentation
Lack of tests	New code will be tested
Data in database is stored double	Database will be redesigned
Data collection happens manually	Automated data collection into database
Use of local filesystem	Centralized database using engine
Front end renders as whole	Use React component rendering
Front end requests all data at startup	Only request data for what can be seen

Table C.2: Problems of the current and proposed solutions



Revised Requirements

Requirements to increase performance	**Design Goal**	**MoSCoW classification**
SmartRoads 2.0 must be able to retrieve the following datastreams from The Nationale Databank Wegverkeersgegevens (NDW): • The NDW ”incidents” stream [1] • The NDW ”measurement” stream [2] • The NDW ”Wegwerkzaamheden” stream [3] • The NDW ”trafficspeed” stream [4] • The NDW ”traveltime” stream [5] • The NDW ”Matrixsignaalinformatie” stream [6]		*Must have*
The SmartRoads 2.0 GUI must have better performance than SmartRoads 1.0. • Rendering of data on the map needs better performance e.g. retrieved and rendered within 1 minute. • To achieve this we need to apply component rendering instead of rendering complete front end • To achieve this the front end must only request what can be seen by the user	*Performance*	*Must have*
Database has multiple issues causing the retrieval of the needed traffic information being too slow. The following requirements will solve it: • The current replay function is too slow and higher performance is required. • The retrieval and processing of a specific amount of traffic data regarding road length and duration must be under a specific amount of time. • The data must be stored more eficiently e.g. smaller tables, more foreign keys and less duplicate information. • A lot of data collection and manipulation happens manually and must now be automated.	*Performance*	*Must have*
SmartRoads 2.0 should have at least the same features as Analytics.SmartRoads features: • Situational messages. • Implement analytics contour plot. • Implement analytics cloud plot.		*Should have*

Table D.1: List of requirements which aim to increase the performance of SmartRoads, along with their corresponding design goal and importance

Requirements for new features	**Design Goal**	**MoSCoW classification**
A new design should make the updates of the road-network configuration file easier or even automatic. An improved design should be introduced to reduce manual handling and the chance of errors.		*Should have*
SmartRoads 2.0 must be able to display free-flow, speed and accidents of the entire road network of the Netherlands	*Performance & Scalability*	*Must have*
SmartRoads 2.0 should retrieve, process, visualize and store other data feeds, that cover the entire Netherlands (e.g. Waze [29]), next to the NDW data.		*Should have*
Introduce an addition tab ”Exceptional situations”. On the ”exceptional tab”, only the exceptional situations which demand attention of the traffic operator will be shown.		*Should have*

Table D.2: List of requirements which are about implementing new features, along with their corresponding design goal and importance

Requirements for in-house development	Design Goal	MoSCoW classification
Make a new architecture design from the current architecture of Scenwise to a modular- or service-based architecture.	*Maintainability*	*Must have*
Make a plan for future modules/services, such that a new student group or a developer from Scenwise knows how to integrate with the system.	*Maintainability*	*Must have*
SmartRoads 2.0 must have good code quality. • The code written by us needs to have at least an 80% test coverage. • The code written by us needs to be fully and clearly documented.	*Maintainability*	*Must have*
Back end must be built in a modular way, such that new features can be added easily in the future.	*Maintainability*	*Must have*
Front end must become modular, in order to tailor software for distinct customers, without rewriting existing functionality.	*Maintainability*	*Must have*
Implement Route planner as a module/service.	*Maintainability*	*Could have*
For the following future modules/services a custom integration plan should be created. • Authentication. • User Profiling. • Route Planner. • Automatic Accident Detection. • Scenario Designer. • Traffic Light Analytics.		*Could have*
Implement the Authentication as a module/service.	*Maintainability*	*Could have*

Table D.3: List of requirements which aim to ease in-house development, along with their corresponding design goal and importance

Requirements to make it more configurable to customers	Design Goal	MoSCoW classification
Implement the User profiling as a module/service. • Add user profiles, with settings per profile. • Users should be able to temporarily change the settings of his/her user profile.	*Maintainability*	*Could have*

Table D.4: List of requirements which aim to make SmartRoads more configurable to customers of Scenwise, along with their corresponding design goal and importance

Long-term Evolution Design

ScenWise

Long-Term Evolution Design

by

J.L. Buijnsters
D. Hofman
J.G.P. Klein Kranenbarg
C. el Moussaoui
K. Zheng

Project duration:	April 20, 2020 – June 28, 2020	
Supervisors:	Dr.ir. H. Wang,	TU Delft, BEP Coordinator
	Ir. O.W. Visser,	TU Delft, BEP Coordinator
	Dr.ir. B.H.M. Gerritsen	Coach
	Ir. K. F. Chan	Client

TU Delft

Contents

1 Introduction **1**
1.1 Overview 1

2 Architecture migration **2**
2.1 Current architecture 2
2.2 Final architecture 2
2.3 Migration steps 4

3 Software Development Lifecycle Guidelines **8**
3.1 Containerization 8
3.2 Documentation 8
3.2.1 Front-end 8
3.2.2 API Gateway 8
3.2.3 Services 8
3.3 Static-analysis 9
3.4 Testing code 9
3.4.1 Front-end 9
3.4.2 Services 9
3.5 Agile workflow 10
3.6 Version control 10
3.7 CI/CD 11

4 Communication Protocols **12**
4.1 OpenAPI Specification 12
4.2 Event-driven communication 14

5 Adding a new Service **15**
5.1 Prerequisites new service external workforce 15
5.2 API Gateway integration 15

Glossary **17**

1
Introduction

1.1. Overview

In this document, the guidelines and practices of the high-level architecture for the future software of Scenwise are mapped out, as well as the envisioned future guidelines regarding software development lifecycles. The aim is to ensure new projects will be of high code quality and integratable with low effort. This means that all new software developed in the future will form one big integrated system.

We first present a plan for an architecture migration from the current situation to the desired situation. It should become clear what is currently missing in the software architecture, but also how these problems can be fixed. Furthermore, the concept of services will be explained and what they are good for.

Next, an overview of the guidelines for software development lifecycles is given, this should enable developers to create high-quality software. It will also ensure that the current code quality problems will not be repeated in the future.

Then a layout of what you need to know about the communication technology that is used in this architecture. This will go a little bit in-depth on the technical parts. This will explain what the rules are for communication between the services and how to use them yourself.

And finally, an explanation will be given on how to add a new service to the software architecture of ScenWise. You should then have all the necessary knowledge to better your software development at ScenWise.

2 Architecture migration

2.1. Current architecture

The current architecture of ScenWise consists of multiple separate applications. Each application, in their simplest form, exists of a front-end, a back-end and a database (see figure: 2.1). This means functionalities and data are not shared between applications. For the front-end similarities, this implies duplicate features, e.g. map, drawing on the map, login, etc. For the back-end, this means duplicate data and functionalities.

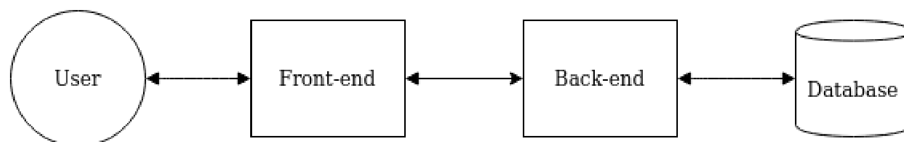


Figure 2.1: Solo application architecture

2.2. Final architecture

What is desired is an architecture where once some functionality has been created, other applications can reuse this functionality without having to rewrite the code. It is undesirable that duplicate data is stored separately for each application, as this uses extra resources. And in the case that tailored software has to be created for a specific client, modules can be reused from existing front-ends and back-ends, with the added benefit that this new tailored software's functionalities can also be integrated by other front-ends and back-ends.

A solution to make reusing functionality a standard practice, a service-based architecture will be created. Functionalities will be in the form of services. Services are small back-ends which only have a small number of coherent functionalities. To use these services a request can be sent to them just as you would normally do from the front-end to the back-end. An example service could be a Route Planner. This service will be responsible for finding a route between two points on a map. After this service has been created once, it can be used in all future applications. And if the Route Planner needs an update, e.g. a faster algorithm has been found, it can be updated in one place, in the Route Planner service, then all applications which use the Route Planner service will automatically use the newly updated service. In figure 2.2 the Route Planner service is taken from the services-cloud to show how this single service is connected to other services which use its outputs or use its inputs. The service-cloud is an abstraction for all independent services.

As multiple services will probably need the same data to function, it would make sense to create centralized databases. This means that the data collected is only stored in one specific database. All services that need that data can read it from that database. In figure 2.2 it is shown how the NDW data would be parsed by a specialized service called NDW (Nationale Databank Wegverkeersgegevens) parser, which fills the centralized database. From there all services which need that data can request it from the same centralized database.

To create custom-tailored software for specific customers it should be possible to connect to all needed functionalities from existing services. It would be very convenient if we can reach all these services from one place. This will be done by an Application Programming Interface gateway or API gateway for short. This gateway knows which services are available in the system and where they are. So for example, if a route needs to be found between two points on a map in the front-end, this can simply be asked to the API gateway. The API gateway will then find the Route Planner service and relay this question to it. The answer will then go back to the front-end from the Route Planner service through the API gateway. This can also be seen in figure 2.2, all distinct front-ends can use the same API Gateway to reach any service that is already functioning in the services-cloud. Thus all front-ends can now use the Route Planner service.

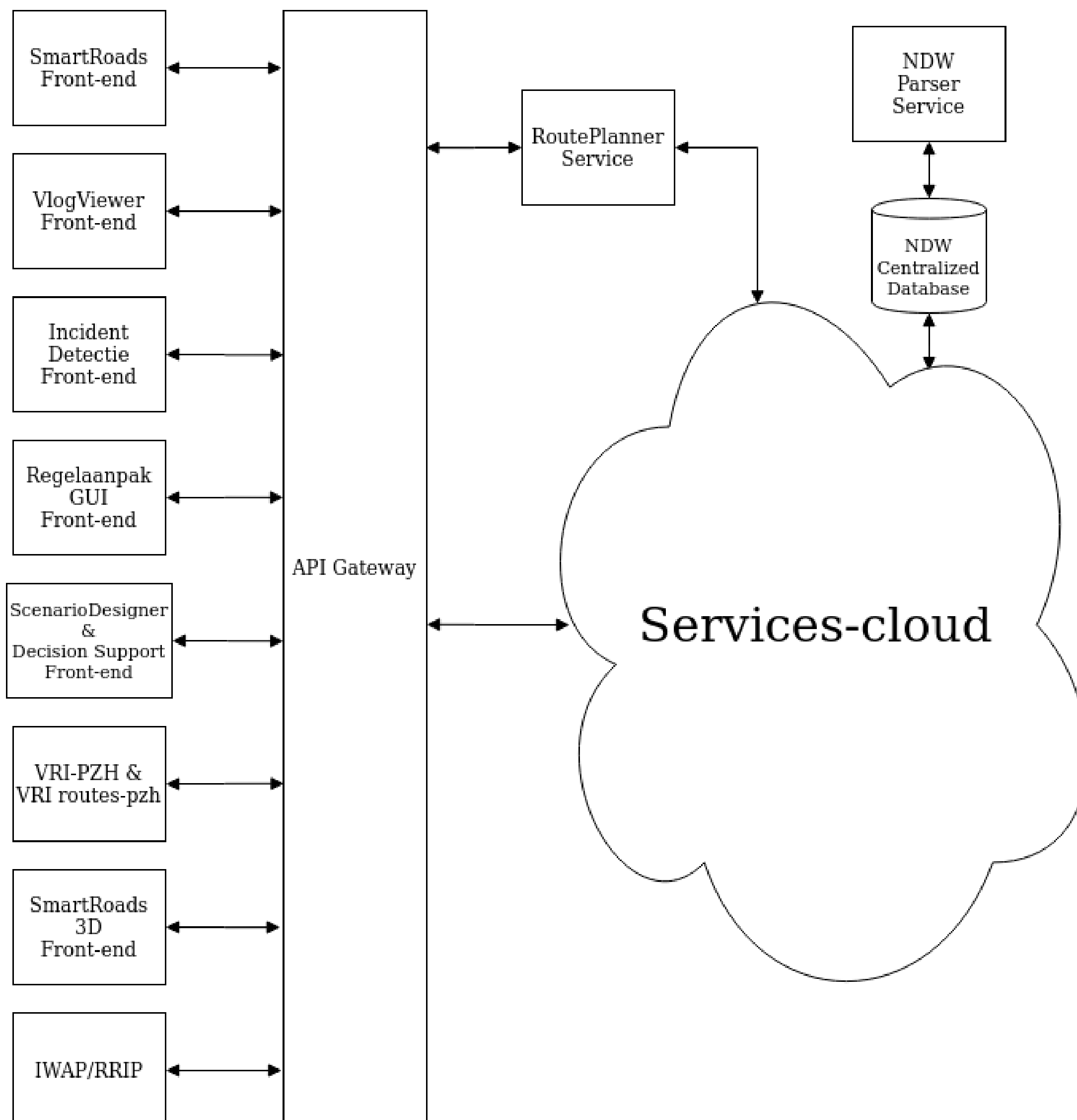


Figure 2.2: Future application architecture

2.3. Migration steps

At the moment, ScenWise has multiple of these solo applications on the market and in beta. It would be beneficial if, for example, the data, or at least some of the functionalities are shared between the applications. Ideally, ScenWise eventually owns an all-inclusive product were all functionalities of the different applications are accessible via one website.

To achieve this, a migration from the current solo application architecture to the described Service-Oriented Architecture [1] is needed. To do this the separate databases, front-ends and back-ends need to be combined. The aforementioned API gateway is a good start to combine the back-ends.

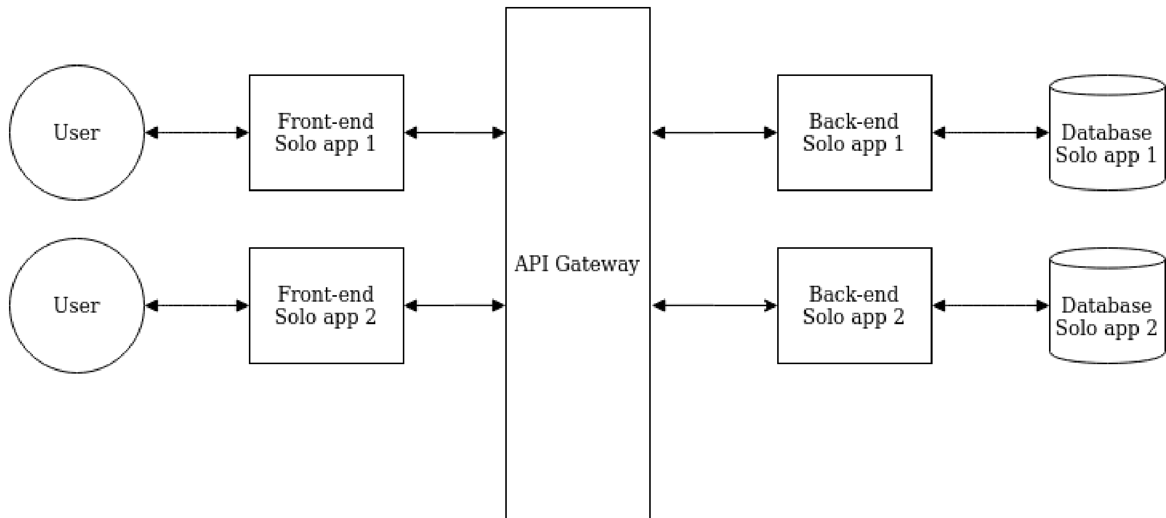


Figure 2.3: Api Gateway

The first step would be implementing this API Gateway. Therefore all existing endpoints should be connected. Endpoints are the places where requests can be made to the back-end for certain data. In figure 2.4 it is visualized how these endpoints can look in a solo application architecture. Three distinct endpoints with colours blue, red and black are visualized. Each endpoint has its own function and/or returns some data, e.g. the black endpoint provides the functionality of the Route Planner. In this case, the front-end needs these three endpoints to function correctly. When implementing the API Gateway, the gateway should offer these distinct endpoints.

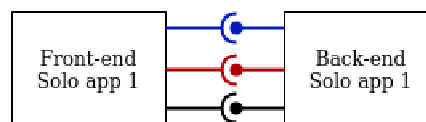


Figure 2.4: Solo Application endpoints

In figure 2.5 it is shown how two front-ends need the same endpoints for data or functionality. And how these front-ends are not directly connected to their back-ends anymore, but to the API Gateway. However it should be noticed that both back-ends supply all three endpoints, thus both back-ends provide a black endpoint with Route Planner functionality. The API Gateway can choose which endpoints it is going to use, this could be configured based on performance.

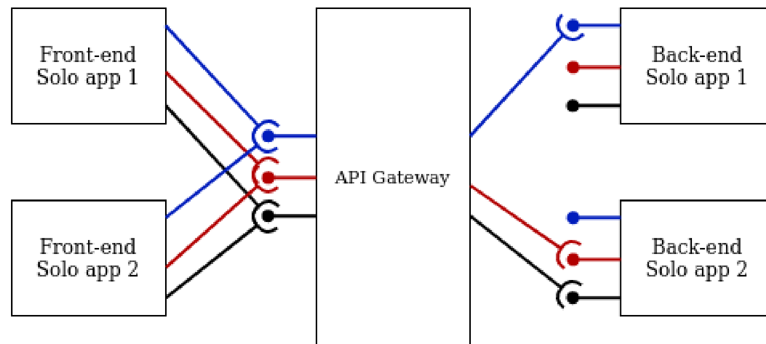


Figure 2.5: API Gateway endpoint implementation

And lastly in figure 2.6, it is shown how the black endpoint, which is responsible for the Route Planner functionality, is now provided by a separate Route Planner service. This means that the API gateway now uses the black endpoint of the Route Planner Service for route planning requests, instead of the previously used black endpoint from the Back-end of the Solo app 2.

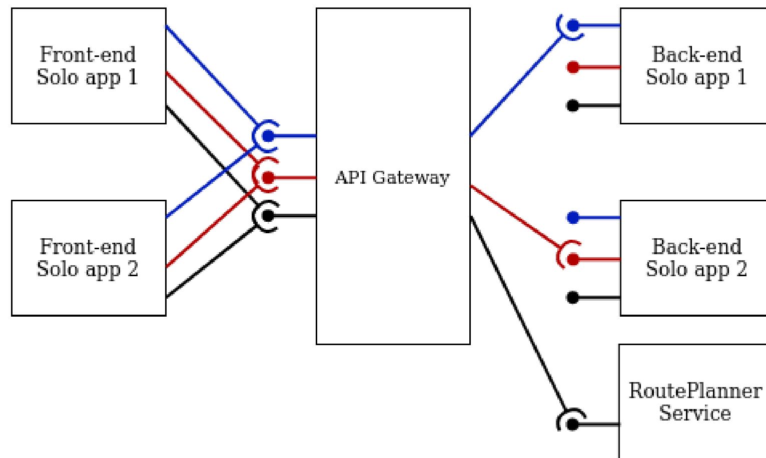


Figure 2.6: Connecting new service endpoint

As you can see in figure 2.3, each solo application front-end is now connected with the API Gateway. From here, the functionalities and data of the desired back-end can be used as shown in figure 2.6. The API Gateway can be seen as a Traffic Controller, pun intended, as it will redirect each request from the front-end to the correct place in the back-end. This means that all front-ends connected to this API Gateway can use all back-end functionality connected to this API Gateway.

This step alone is very useful for the development of new front-ends, as you can reuse all functionalities reachable by endpoints from previously developed back-ends. In figure 2.7, you can see how such a front-end connects with the API Gateway and has access to all functionalities from the connected Solo app back-ends. In this case, we are creating SmartRoads 2.0.

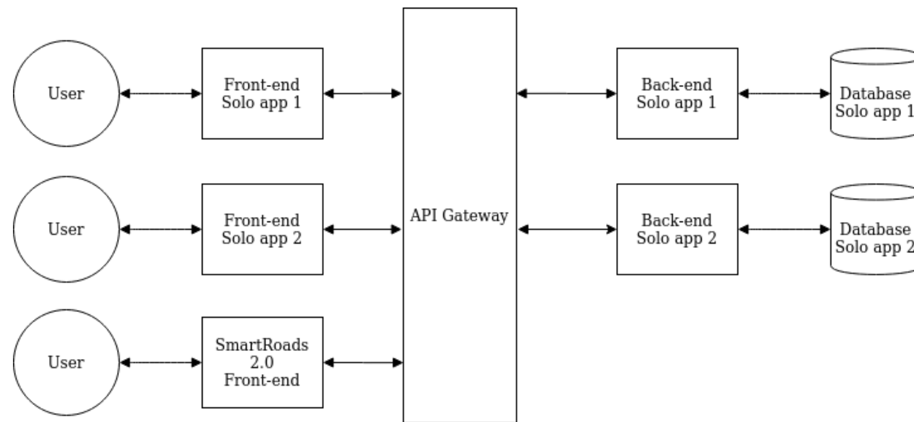


Figure 2.7: New Front-end

From here, the first steps to the new services-cloud can be taken. At the moment, each back-end has a lot of functionalities and endpoints. These need to be split up in clear services which only contain very similar functionality. They should be able to function autonomously, meaning that they should not depend on other services. Splitting up the back-end will be done with the so-called Strangler pattern [2]. This means that, systematically, groups of similar functionalities should be taken from a back-end and put in a service.

With this method, services will gradually take over more and more functionality from the old back-end. This will occur until there comes a time that all functionalities have been turned into services and the old back-end is no longer useful and can be taken down.

Some functionalities rely on data to function. In the Solo application architecture, the back-end gets this data from its database. As discussed previously it would make sense to create a centralized database which all services who need that data can use. This will ensure that data is not stored multiple times in different databases and is not stored in different formats. An example of storing data in a different format would be a date presented by dd/mm/yyyy or mm-dd-yyyy. By having it centralized, all services need to use the same database and thus the same data formats. This makes the services uniform and thus easier to use across multiple applications.

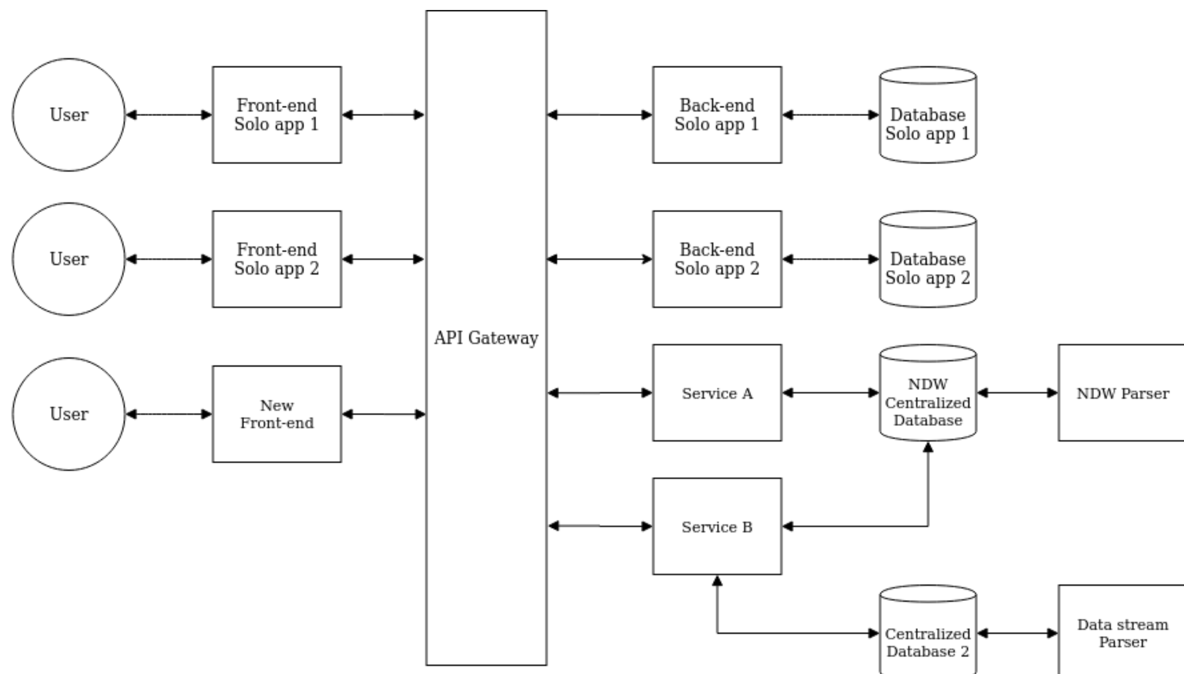


Figure 2.8: Services and centralized databases

As you can see in figure 2.8, a new centralized NDW database has been added. This database is filled by the NDW Parser. Both services A and B use the data from this database to provide for their functionalities. A second centralized database can be seen which is filled with data from another data stream. As you can see, Service B uses data from both databases to provide for its functionalities. At this point, all front-ends can make use of these services A and B.

Continuing this pattern will eventually provide a large and diverse service-cloud, where any service can be accessed by any front-end connected via the API Gateway.

The main benefit of this architecture is the high re-usability. Each service is a small code base which makes it much more maintainable. The API Gateway will allow monitoring of which services might be lacking performance-wise. Then the decision can be made to swap such a service with a new solution, without having to worry about all other services. It is very easy to add new functionality to the system in this architecture. Data is available via the centralized databases and core functionalities are available via the already existing services.

One of the pitfalls of this system is its complexity. This is why the API Gateway is very important. It is the connection to the big cloud of services, meaning that it is the key component to knowing which services are available and where. And for example, if a service is down because it has crashed, the API Gateway has to be fault-tolerant and know how to deal with a malfunctioning service.

We are going to minimize this complexity by creating very clear guidelines and communication rules in the next chapters. Finally, a tutorial will be given on how to correctly add a newly developed service to the service-cloud.

3 Software Development Lifecycle Guidelines

3.1. Containerization

We strongly advise to start adopting containerization for all existing components. We have used Docker [3] for the containers of the front-end, API gateway, and each service. Containers bundle all needed configuration files and dependencies required to run one of the system components. This container ensures that components can always run regardless of what software is available on the host machine.

3.2. Documentation

To make the code more understandable for both current and future developers, there needs to be sufficient documentation to explain each part of the code. Each component, e.g. front-end, gateway, or services, should have a README.md file which gives an overview regarding what the component is used for and how to set it up. This file can be found in the root directory of the component. Thanks to this file, developers can start using this part of the codebase within a reasonable amount of time, without any extensive debugging if they run into problems.

3.2.1. Front-end

In short, documentation should be written on a class/component, method, and in some cases, line level. That is, almost every method/function should be documented, and some important or hard to understand lines as well. Of course, methods/functions that are self-explanatory such as the simplest form of a getter or setter can be left out of documentation. The functions constructor(props) and propTypes() can also be left out of documentation. A method/function should be documented on its function, parameters, and if applicable the return statement.

3.2.2. API Gateway

OpenAPI 3.0 specifications for all end-points of each service. This makes it very clear what can be easily re-used. More information regarding this documentation can be found in chapter 4. Besides that, the code within the gateway should also be documented appropriately. Developers should be able to understand what the code does after reading the documentation of a route or a filter.

3.2.3. Services

Each service should have its README.md file. All code in future services should also get appropriate documentation. Make sure that regardless of what software stack is used, all functions are explained by the documentation. Any developer should be able to understand what a function does by taking a look at its documentation.

3.3. Static-analysis

Static-analysis is a code analysis without execution of the code. It can check coding style rules, like proper documentation, or find possible bugs. It is often one of the first steps in a CICD pipeline, as the lack of code execution makes it very fast providing quick feedback to the developers. There are tons of static-analysis tools available for most programming languages. We strongly advise to use some sort of static analysis for each component of the system and integrate those in the CICD pipeline, more information can be found in section 3.7.

3.4. Testing code

An important factor of attaining and maintaining high code quality is by testing.

3.4.1. Front-end

Unit tests

There are multiple ways of testing the front-end. In general, for each component, we test whether it renders correctly. We also make a snapshot of the HTML so that unwanted changes in the HTML can be detected. We also test state manipulation and each non-react function. We use the Jest and Enzyme dependencies to write our tests, and keep the coverage on a minimum of 80%.

3.4.2. Services

Unit tests

Each service should be unit tested. Meaning all individual functions should be tested by one or more tests. For example, a function is responsible for the addition of two numbers. It should be tested whether this function returns the correct value for 2 given numbers. May a developer accidentally change the addition character to a multiplication character this will be caught by our unit test. The benefit of this is that bugs can be found in a very early stage. And for example, when updating dependencies it will immediately be clear if functions do not behave as intended.

We advise setting a minimal rule of unit test coverage of 80% for each independent service. Note that this is a bare minimum, maximizing this coverage is strongly advised.

Contract tests

Contract testing is a type of testing which ensures that the system is aware of any changes in the interaction with an external service [4]. The contract of a service is the possible interactions which can be made with that service. As ScenWise has software projects with other companies, this would be a good way of making sure that, if for example, a development team from the other company change the contract of a service which is integrated to the services-cloud of ScenWise, this will not go unnoticed and the services of ScenWise which use the external service can be updated accordingly. In the following section, it will be explained how these contract tests can be used for integration testing the services of ScenWise. Contract testing will become specifically important when the company will get multiple teams maintaining services.

Integration tests

Integration tests are responsible for the testing of independently deployed services. Two types of integration tests can be specified, namely, the narrow integration tests and the broad integration tests [5]. The narrow integration tests are similar to unit tests, except that they are responsible for testing of interactions of the tested service with external services. In this case, the external services are mocked. The mocks can be supported by contract tests [4] of the external service, this makes the tests faster and more reliable. This type of integration tests doesn't require all services to be deployed. The broad integration tests are run on a system where all services are live. These test if multiple services work together as we expect them to, instead of only testing the code responsible for the interactions.

Based on the API specifications, we use broad integration tests. As explained, these tests are introduced to make sure that if a service's endpoint mutates or stops working we are aware of it. This means that we should find the problem before a customer would. Therefore, each endpoint should be tested. For the integration tests of the API Gateway, Postman [6] is used. Postman can run collections of tests periodically on the deployed services. This will give a clear overview of the health of the system.

These tests should also be integrated into the CI/CD pipeline.

Correctly implementing and keeping up to date of both types of integration tests in the new architecture will ensure that if something in the service-based system fails, ScenWise can deal with it before a customer has to deal with errors.

System testing
System testing includes testing the fully integrated system. The broad integration testing of the API Gateway can be seen as system testing the back-end. However, it is also desirable the fully integrated front-end is tested. In the CI/CD section 3.7 it is explained how a near-production version of the application is deployed live in a staging environment. This staging environment can be used for user acceptance testing, which is a type of system testing, for example, when a specific new feature is requested by a product manager. When the feature is developed by a developer it can be deployed on the staging environment where the product manager can test it out. May there be any feedback on performance, design, colour mismatch, etc, this can be done before the new feature is deployed on the production server.

3.5. Agile workflow

To work efficiently in a team on a large project, a structured workflow is necessary. An agile workflow helps to break down a large project into concrete smaller tasks. A good framework for an agile workflow is scrum. Scrum is based on sprints which usually take 1 or 2 weeks. A product owner or manager specifies which results he wants to achieve, often given by using user stories. They are sorted on priority. They can also be placed on a product backlog to keep a clear overview of what has to be done. This can be done on a whiteboard or an online board.

At the beginning of a sprint the team discusses what user stories are going to be implemented the upcoming sprint, the workload should be divided equally by estimating the amount of time per task. These tasks are placed on a sprint backlog, this like the product backlog can be a whiteboard or an online board. Each task has a corresponding status, for example, to-do, in-progress or done. In the event tasks from the previous sprint have not yet been completed then these will also be taken into account. If any problems or bugs arise during the sprint these tasks can be added to the current sprint after consultation with the product owner, or they are passed on to the next sprint.

Every day a daily scrum meeting is held in the morning at a fixed time. The meeting should take no longer than 15 minutes. Normally only the development team is present. Each team-member gets asked what he or she has done the previous day, what he or she is going to do today, if they ran into any problems or challenges, do they need help, or is there something which can be of interest for other team members? The answers to these questions should be prepared to ensure they are succinct. If there are any larger discussion, these should be held outside of this daily scrum meeting.

After the sprint, a sprint review can be held. Here the completed tasks can be showcased to the whole team and any other interested coworkers.

Within the team, a retrospective should be held. This is an evaluation at the end of the sprint about what went good and what went wrong. The main focus is to solve any problems which arose during the sprint and make sure these problems will not manifest themselves in the next sprint. This can be about anything relating the sprint for example team communication.

All these meetings are lead by a scrum-master, he or she is responsible for the whole scrum-process.

As the number of developers will most probably grow in the future of ScenWise we advise adapting such an agile workflow, as it will otherwise become increasingly harder to work efficiently on a large end-product.

3.6. Version control

In terms of version control, it is pretty straightforward. Two important branches come into play; the master and the develop branch. As the name indicates, the develop branch is for the development stage. To change code, e.g. implement a new feature or fix a bug, you branch from the develop into a separate new branch, with a descriptive title which is often the branch number following the title of the corresponding issue. After making the changes needed, you put in a merge/pull request to the develop

branch. This merge request gets approved by other developers, which check if all code quality is up to standard. When you feel comfortable deploying a new version of the code to production, you can make a merge request to merge develop branch into the master. After that, it is a simple case of deploying which will be explained in the next section.

3.7. CI/CD

To automate the integration of code and the checks that come with it, Continuous Integration (CI) should be used. As the names indicate, the use of this concept is to continuously integrate code that has been added, deleted, or changed. This allows developers to see with just one glance, whether the tests, static analysis and build(s) pass. This avoids unnecessary time consumption and mistakes in manually running tests, static analysis and build(s). Additional options can be added such as showing the change in test coverage, and the overall test coverage. To automate the deployment of the new code base, Continuous Deployment (CD) should be used. This stage can be run after a merge request to the development or master branch has been merged. An update on the development branch should be deployed on, for example, a staging environment on a beta server. A new version on the master branch should be deployed on the production servers.

4
Communication Protocols

In this chapter, the communication protocols will be laid out. Developers need to be able to use services created by other developers without having to work through their codebase. Besides, that communication protocols are specifically important in the envisioned architecture as a large number of services need to be able to correctly communicate with each other when new services are added to the system.

4.1. OpenAPI Specification

As stated it is important that developers can easily use services they are not familiar with yet. A good way to accomplish this is by documenting all reachable endpoints they can connect to. This means each service should have an interface which maps out all possible requests and their corresponding responses. A solution for describing REST (Representational State Transfer) APIs is the OpenAPI Specification [7]. This allows describing available endpoints, operation parameters and authentication methods. As REST is already in use within the company and as it is a very popular technology for communications using HTTP, this will be very useful. OpenAPI specification [7] files can be served as a simple user interface using Swagger UI [8]. An example endpoint for retrieval of status-messages can be seen in figure 4.1. It describes an HTTP get request with a date as an optional request query parameter. There is also an example response documented which will immediately make clear to a developer what he can expect of a response from this endpoint. There is even a ”Try it out” button to send a request and get a real response. This is the place where developers can learn what the gateway or a specific service has to offer.

In the OpenAPI Specification [7] files multiple responses can be described for the same request. For example, if an invalid input query parameter date is given to the status-messages endpoint it should return an HTTP 400 status which means a Bad Request. The HTTP-status-codes can be found here [9]. The services must respond with the correct status code. Therefore, it is important to make sure that the service responses always adhere to the OpenAPI specification.

SmartRoads 1.0 All endpoints connected to the SmartRoads 1.0 functionalities.

GET `/api/status-messages` Returns all relevant status-messages.

This endpoint can be used to retrieve all information about all relevant status-messages in the present. Or when given the date query parameter from a specific point in history.

Parameters Try it out

Name	Description
date string($date-time) (query)	The timestamp for which status-messages data should be retrieved. 2020-04-01T17:40:00.000Z

Responses

Code	Description	Links
200	A JSON object with status-messages.	No links

Media type
application/json;charset=utf-8
Controls Accept header.

Example Value | Schema


```
[
  {
    "situationId": "A-ALL-IM19087676NLD",
    "type": "VehicleObstruction",
    "specificType": "brokenDownVehicle",
    "startTime": 1576060348,
    "endTime": null,
    "source": "Verkeerscentrale",
    "probabilityOfOccurrence": "certain",
    "specificLocation": 7000,
    "offsetDistance": 2500,
    "specificLocationS": 7003,
    "offsetDistanceS": 100,
    "constructionWorksExtension": {
      "delay": "upToTenMinutes",
      "source": "Gemeente Tilburg",
      "overallStartTime": 1588862591,
      "overallEndTime": 1601650800
    },
    "constructionTexts": [
      {
        "type": "internalNote",
        "value": "Van de Coulsterstraat, Tilburg"
      },
      {
        "type": "warning",
```


Figure 4.1: Example documentation endpoint

As discussed in the previous chapter integration tests should be used to make sure the system is healthy and functioning. These integration tests should adhere to the OpenAPI specification files as these should be correct. If, for example, the status-messages endpoint is updated and would now use a different date format, then this should not only be updated in the integration tests, but also in the OpenAPI documentation because if there are any inconsistencies between the two, the integration tests should fail.

Unfortunately, there is no correctness check if the OpenAPI documentation is according the Postman integration tests this should be implemented in the near future. This can be easily done by running the integration tests on an OpenAPI mock server. We strongly advise getting such a system in place within the CICD to check the OpenAPI documentation.

A REST API is specifically useful for communication between the front-end and back-end. However, this request-response communication is less ideal between the services in the future services-cloud. This cloud should be using another way of communication, intending to be loosely coupled and highly scalable, namely Event-driven communication. Loose coupling means each service has little to no knowledge about other services in the system. This inherently makes the system highly scalable as adding, updating or deleting a service does not require any changes across the whole system. Such a system is very much desired in an ever-growing codebase.

4.2. Event-driven communication

In an event-driven architecture, a service will produce events after it has done its work. Unlike REST, services that produce events do not need to know the specific locations of other services. The events can be published to a queue which processes them and makes sure the services which need the event gets them. Or the events can be delivered via a pub/sub model [10]. After an event is delivered to the appropriate services each service will consume the event, which means performing their corresponding tasks, this can include creating follow-up events. For example, after a user has requested a new account an event will be produced, namely the 'create_user' event. This event contains the login information of the new user. An authentication service which is subscribed to this event via pub/sub or a queue creates the new user and sends a follow-up event, namely 'user_created', containing the email address of the new user. An email service which is subscribed to the 'user_created' event gets the event from the queue or pub/sub model and consumes it. It will send a welcome email to the new user.

Services operate independently without the knowledge of other services, which makes the system loosely coupled. They will only need to know the types of events which are interesting to them. This decoupling makes the system very scalable.

On top of all this, event-driven communication is very appropriate for the type of data ScenWise collects and processes. The main data feeds are live data. At the moment these data feeds are coming in via a pull-based matter which is inherent to the Request/Response model currently used. This means that in the case that new data is available every minute and this data would be polled every minute, there will always be a delay of 0 to 59 seconds before being up to date. In most cases, this live data could also come in via a pushed-based manner which is inherent to the event-driven model. Meaning, when there is new data available at the source, the data will be immediately pushed to the system. This always ensures a minimal delay in obtaining new data. At the moment the pull-based approach is used for the NDW data, even though NDW supports a pushed-based interface. We suggest that ScenWise switches to the pushed-based approach for all data streams which have this option.

Unfortunately setting up the basics for an event-driven communication architecture between services was out of the scope of our project as ScenWise demanded a better performing and fully functioning new version of SmartRoads 1.0. As a consequence, the focus was almost solely needed for the front-end and a functioning gateway. We want to emphasize that the implemented gateway and the new front-end are just a start to the new architecture. Implementing event-driven communication between services asks for investment, and is not necessary to create a functioning service-based architecture. We do however strongly advice to start implementing an event-driven architecture in this phase as from this moment all services are going to be created. It will be less costly implementing event-driven communication right from the start than over a few years. The longer ScenWise waits, the higher the effort will be.

If ScenWise is willing to make this investment then it will be equally important that the event types will also be properly documented. The events can be seen as a variation of the traditional REST endpoints. Unfortunately, OpenAPI specification [7] is not ideal for documenting these events. Luckily there are event-based alternatives like AsyncAPI [11].

5

Adding a new Service

5.1. Prerequisites new service external workforce

As ScenWise regularly has workforces which are not yet familiar with the software at ScenWise, for example, student projects, it is important to provide the group with appropriate information about the software architecture and the appropriate tools for the desired software development.

List of things a new group of developers should get from the start of their project:

- A front-end codebase for implementing new features in the desired UI
- Access to all remotely useful services and their documentation, for example, OpenAPI specifications
- A clear set of rules ScenWise imposes on their software, for examples see chapter 3
- A virtual private server for hosting a live version of the service
- Optional: Providing an OpenAPI specification file for the new service

5.2. API Gateway integration

To integrate the new service in the API Gateway and make it available to all front-ends the following steps must be taken. In the API Gateway configuration file (application.yaml) the host of the service must be set. In the case of a student project, the host can be set to the provided VPS. Next, the service route must be set by creating an extra routes entry in the same configuration file. If there is a need for extra more specific configurations please check the docs of the Spring Cloud Gateway [12]. Add the OpenAPI specification files for the routes which should be available through the gateway to the API Gateway docs. In the case that the service is under development, tag the routes as WIP (Work In Progress) in the OpenAPI specification file, to make sure other developers know this route is not yet available but will be in the near future. If the service is ready for production, make sure the routes are added to the integration tests of the gateway. And that's it, the endpoints of the new service should now be available via the API Gateway.

Bibliography

[1] *Service orientated architecture patterns*, Accessed: 2020-5-7. [Online]. Available: https://patterns.arcitura.com/soa-patterns.

[2] S. Newman, *Building microservices*, Feb. 2015. [Online]. Available: https://samnewman.io/books/building_microservices/.

[3] Docker, *Docker*. [Online]. Available: https://www.docker.com/ (visited on 06/18/2020).

[4] M. Fowler, *Contracttest*, martinFowler.com, Jan. 2011. [Online]. Available: https://martinfowler.com/bliki/ContractTest.html (visited on 06/22/2020).

[5] M. Fowler, *Integrationtest*, martinFowler.com, Jan. 2018. [Online]. Available: https://martinfowler.com/bliki/IntegrationTest.html (visited on 06/22/2020).

[6] Postman, *Postman*. [Online]. Available: https://www.postman.com/ (visited on 06/17/2020).

[7] SMARTBEAR, *What is openapi*. [Online]. Available: https://swagger.io/docs/specification/about/ (visited on 06/16/2020).

[8] SMARTBEAR, *Swagger ui*. [Online]. Available: https://swagger.io/tools/swagger-ui/ (visited on 06/16/2020).

[9] w3, *Http status codes*. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html (visited on 06/16/2020).

[10] Microsoft, *Implementing event-based communication*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications (visited on 06/17/2020).

[11] A. Initiative, *Asyncapi*. [Online]. Available: https://www.asyncapi.com/ (visited on 06/16/2020).

[12] Spring, *Spring cloud gateway*. [Online]. Available: https://spring.io/projects/spring-cloud-gateway (visited on 06/21/2020).

Glossary

containerization Containerization has become a major trend in software development as an alternative or companion to virtualization. It involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure. The technology is quickly maturing, resulting in measurable benefits for developers and operations teams as well as overall software infrastructure. 8