**Developing a decision support web application for waste collection management**

Towards the optimization of Swiss municipalities’ waste collection management

**Master’s thesis**

**Programme:** Master of Arts in Business Informatics

**Author:** Manuel Gonçalves Barroso

**Supervisors:** Dr. Reinhard Bürgy and Vera Fischer

University of Fribourg

Department of Informatics

Decision Support & Operations Research Group

**Printing:**

**Year:** 2021

**Place:** Fribourg

Logo, company name

Description automatically generated

# Table of contents

[Table of contents II](#_Toc79160570)

[List of figures IV](#_Toc79160571)

[List of tables V](#_Toc79160572)

[List of source codes VI](#_Toc79160573)

[List of abbreviations VII](#_Toc79160574)

[1 Introduction 1](#_Toc79160575)

[2 Literature Review (Look again at citations!!!) 2](#_Toc79160576)

[3 Software Development Process 4](#_Toc79160577)

[3.1 Requirements Engineering 5](#_Toc79160578)

[3.2 Server 7](#_Toc79160579)

[3.3 Database 30](#_Toc79160580)

[3.3.1 Technologies 31](#_Toc79160581)

[3.3.2 Implementation 31](#_Toc79160582)

[3.4 Client 32](#_Toc79160583)

[3.4.1 Technologies 32](#_Toc79160584)

[3.4.2 Implementation 32](#_Toc79160585)

[3.5 Testing 32](#_Toc79160586)

[3.5.1 Technologies 33](#_Toc79160587)

[3.5.2 Implementation 33](#_Toc79160588)

[4 Manual 34](#_Toc79160589)

[4.1 Common web app functionalities 34](#_Toc79160590)

[4.1.1 Sign Up 35](#_Toc79160591)

[4.1.2 Log In 35](#_Toc79160592)

[4.1.3 Log Out 36](#_Toc79160593)

[4.2 Waste collection-specific functionalities 36](#_Toc79160594)

[4.2.1 Creation of a project 37](#_Toc79160595)

[4.2.2 Management of users 39](#_Toc79160596)

[4.2.3 Management of projects 40](#_Toc79160597)

[4.2.4 Management of a particular project 41](#_Toc79160598)

[5 Conclusion 47](#_Toc79160599)

[Appendix 48](#_Toc79160600)

[Bibliography 52](#_Toc79160601)

# List of figures

[Figure 1: Homepage of the web app 34](#_Toc79160602)

[Figure 2: Sign Up form 35](#_Toc79160603)

[Figure 3: Log In form 36](#_Toc79160604)

[Figure 4: Navigation drawer after the users' login 37](#_Toc79160605)

[Figure 5: Navigation drawer for user with administrator rights 37](#_Toc79160606)

[Figure 6: Create new project form 38](#_Toc79160607)

[Figure 7: Users management in the web app 39](#_Toc79160608)

[Figure 8: Projects management in the web app 40](#_Toc79160609)

[Figure 9: Navigation drawer after selecting a project 41](#_Toc79160610)

[Figure 10: Simplified workflow of the app 42](#_Toc79160611)

[Figure 11: Garbage scenarios list example 42](#_Toc79160612)

[Figure 12: Edit vehicle type example 43](#_Toc79160613)

[Figure 13: History example collection point scenarios 44](#_Toc79160614)

[Figure 14: Results list 44](#_Toc79160615)

[Figure 15: Example clipping of result view 45](#_Toc79160616)

[Figure 16: Clipping of the 'Calculate Result' view 46](#_Toc79160617)

# List of tables

**No table of figures entries found.**

# List of source codes

[Source code 1: Vue single file component template example 31](#_Toc79160618)

[Source code 2: Example XML file for the creation of a new project 38](#_Toc79160619)

# List of abbreviations

API *application programming interface*

DSS *Decision Support System*

JSON *JavaScript Object Notation*

# Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip. Ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laboru.

# Literature Review (Look again at citations!!!)

The goal of this chapter is to, firstly, provide explanations of basic core concepts in the thesis at hand. Some of the explained concepts are decision support systems (DSS), waste, waste management and waste collection management. Secondly, the development of a waste collection DSS is motivated by literature reviewing the benefits of DSS and waste management. Thirdly, proposed design and development principles of DSS are reviewed in existing literature in order to get an overview of these findings for the development of the DSS. Lastly, some developments of waste collection DSS are reviewed, and its conclusions are presented.

Keen and Scott Morton define decision support systems as computer-based support for decision-making managers who deal with semi structured problems (P. G. W. Keen & Scott Morton, 1978, p. 97). They claim that DSS need sufficient structure to be of value and improve the effectiveness of decision processes but are tools that do not automate the decision process, i.e., manager’s judgements are essential when using DSS (P. G. W. Keen & Scott Morton, 1978, p. 2).

Pongrácz defined waste as either a thing which in its structure and state in the given time and location has no utility to its owner or an output without owner and purpose (Pongrácz, 2002, p. 129). Waste management was defined as the control of waste-related actions (which includes the creation, handling and utilization of waste) in order to protect the environment or conserve resources (Pongrácz, 2002, p. 131).

Waste collection management is a subset of waste management and the collection of waste begins with garbage bins holding waste and ends after the transportation to a place where the waste is processed, transferred or disposed (Chandrappa & Das, 2012, p. 70).

Properly managing waste collection is an important task since poor management can lead to an overflow of garbage at the storage sites (Chandrappa & Das, 2012, p. 65). Poor waste management (which often includes waste collection management but is not limited to it) has been argued to decrease the quality of ground water (Nkolika & Onianwa, 2011; Vasanthi et al., 2008), communities’ social cohesion (Owusu, 2010), public health (Ziraba et al., 2016) and increase environment pollution (Apostol & Mihai, 2012).

Past studies have found DSS to increase the effectiveness of decision makers. Ferguson & Jones (1969) concluded, following their experimental study on the usage of a DSS by 300 white collar workers and students, that decision making skills were mostly improved. Sharda et al. (1988) set up a decision-making game and found that participants made more effective decisions if they were allowed to use a DSS. The same study also concluded that the introduction of a DSS also lead to a decreased profit variance of decision makers (Sharda et al., 1988, p. 153). Findings from a nine week simulation game by Barr & Sharda (1997) show that overall decision performance was improved by a DSS. Interestingly, the study also showed that groups which at the beginning of the experiment were allowed to use a DSS and later not, decreased performance to a level lower than groups which were never allowed to use a DSS. To groups which began the experiment without DSS and were later allowed to use one, the opposite happened, i.e., their decision performance increased to a level higher than the group that was allowed to use a DSS over the full nine weeks. These findings were in the first case attributed to a dependency of the group on the DSS and in the latter case to an increased decision problem understanding before the introduction of a DSS (Barr & Sharda, 1997, p. 143).

Pick (2008) argued that a DSS can not only improve decision quality, but also the decision-making process itself in a, possibly, subtle way. He gives an example where he explains that a decision maker can explore a problem thoroughly with a DSS which would improve his problem understanding. This will possibly lead to a better decision process, but it is difficult to quantify the impact (Pick, 2008, p. 719).

A literature review of around 30 studies identified twelve commonly mentioned benefits of DSS. These benefits included more effective decisions, reduced labor hours, reduced costs, better teamwork and enhanced communication (P. G. Keen, 1980, pp. 32–36).

Igbaria et al. (1996) praised the benefits of a vehicle routing DSS. In that study, the authors found that the introduction and usage of the DSS lead to reduced labor hours, more efficient schedules, decreased number of drivers and enhanced flexibility as well as morale of office workers (Igbaria et al., 1996, p. 215).

Rose et al. (2016) used several techniques, including surveys, interviews and a workshop with various stakeholders, to study which design (and delivery) factors affect the use of DSS by farmers in the UK and identified 15 of them. Some of the identified design factors were ease of use, performance of the app, relevance to the user, IT skills needed to operate the app and facilitating conditions (i.e., if an effective usage of the DSS possible) (Rose et al., 2016, p. 173).

Alvarez & Nuthall (2006) studied the usefulness of DSS for farmers in two places in Uruguay and New Zealand. Their findings show that the application in order to be useful must fit the farmers habits of doing their work and provide a certain level of ease of use (Alvarez & Nuthall, 2006, p. 58).

Ahn & Grudnitski (1985) conceptually identified for the development of DSS important factors to monitor which not only capture application design, but also organizational and behavioral views. They summarized their findings in nine key points some of which are pointed out in the following sentences. The DSS should fit the habits of users, e.g., it should allow users to work with the app in a way they are familiar with (Ahn & Grudnitski, 1985, p. 29). The cooperation of DSS users and designers is of high priority and should be encouraged (Ahn & Grudnitski, 1985, pp. 29–30). Organizational structures also interact with the development of the DSS and need to be monitored since they can impact its success (Ahn & Grudnitski, 1985, p. 30).

# Software Development Process

In this chapter the application development process is described.

The software development process can be partitioned in four parts which are the requirements engineering, the database development, the server-side application development and the client-side application development. While the requirements engineering lays out a plan for the development of the entire application, the other three components follow a bottom-up approach where the database is the piece of software that is furthest away from the user and the client-side application is the closest to the user. The server-side application is then the connection between the database and the client-side application.

In theory, these four components could be concluded one after the other, but in practice they provided more of a general guideline in the development process, where it was possible to adapt previous steps in later stages (e.g., the requirements could be adapted when the server-side development was in progress). The reason that those steps were not concluded in a purely chronological order is that, on one hand, it is difficult to completely grasp the requirements at the beginning of the project (adaptations of requirements may happen in later stages of the process) and, on the other hand, since the database, the server-side and the client-side applications are connected between each other unexpected changes in one of those components might lead to changes in the other two components.

The server-side and client-side development can additionally be split in a coding and a testing part. The coding and the testing of the server-side and client-side application was done in an iterative way where first some part of the application was coded and then tested as needed.

The first subchapter is going to describe the requirements engineering of the application. Concretely, it will look at the use cases that were developed and the prototype that was created at the beginning of the process. The second subchapter is going to look more in depth into the database design. The third subchapter looks at the architecture of the server-side application and how it interacts with waste collection optimization algorithms developed by Bürgy et al. 20xx **(what and how to cite here?)**. Lastly, the fourth subchapter explains the functioning of the client-side application. In each of the subchapters the used technologies are going to be shortly presented.

## Requirements Engineering

Requirements engineering processes do (among other things) elicit, analyze, negotiate, specify and validate software requirements and are considered one of the most critical steps in the software development process (Tahir & Ahmad, 2010, p. 1). It can have a negative impact on later stages of the software development process if it is poorly executed and is crucial in order to finish software projects successfully since it is its foundation (ur Rehman et al., 2013, p. 1). This clearly shows the importance of requirements engineering and why this step should not be skipped.

The chosen technique to specify software requirements was a screen mockup. Screen mockups have been shown to be very useful for understanding the requirements of a project (Ricca et al., n.d., 2014, 2010). Additionally, a use case diagram was created. Use case diagrams, even though they are vague, help to not lose the fundamental functionalities that the application must have out of sight (McLaughlin et al., 2007, p. 297).

The screen mockup was developed iteratively by meeting the stakeholders of the project regularly. The first meeting notes were taken about how the application could look like. Then for the subsequent meeting a screen mockup was developed and presented to the stakeholders which could then input their improvement propositions. Then before each subsequent meeting the mockup was improved based on the inputs of the stakeholders which again could input their improvement propositions. This process was repeated until the mockup was judged to sufficiently correspond to what the stakeholders imagined.

[insert image representing the process]

The developed screen mockup shows, on one hand, generic application functionalities such as log in and sign-up functionalities. On the other hand, it shows application specific functionalities such as a scenario-based input specification functionality (e.g., the user can input how much garbage is produced on each node and save this specification) and a solution request functionality which allows the user to request individualized solutions for its specific municipality by choosing its input parameters (which are among other things the scenario-based input specifications). Additionally, the administrator should be able to create a new project for a municipality by uploading its map to the application, manage project permissions (i.e., who is allowed to access a certain project) and delete users if needed. The developed use case diagram of the application can be seen on graphic xxx.

[show: use case diagram]

After agreeing on the functionalities, the next step was to specify the general architecture of the project. First, the components that are going to make up the application are: the database, the server, the client-side application and the waste collection optimization algorithm.

**The client-side application**

This is the user interface. It will allow the users to interact with the application. The user can input data and request solutions.

**The database**

The application data will be stored in the database. Those data include user data (email, password, ...) and project data (scenario-based input data, solutions data, ...).

**The waste collection optimization algorithm**

This piece of software was already developed by Bürgy et al. 20xx. It still needs to be considered when building the application since the application needs to provide input data to the algorithm and receive output data from it.

**The server**

This is the core of the application. It interacts with the other three components by receiving data from the client-side application, by reading and writing to the database and by inputting and receiving data from the waste collection optimization algorithm.

[show inter-component architecture]

## Technologies

### PostgreSQL

PostgreSQL was chosen as the database to store the applications data. It is an open-source object-relational database management system (*1. What Is PostgreSQL?*, 2021). SQL (structured query language), which is a programming language that permits the querying and manipulation of data in a relational database (*SQL vs. NoSQL Databases*, 2021), is largely supported by PostgreSQL (*1. What Is PostgreSQL?*, 2021).

There are a few hierarchical units that we must be aware of when working with PostgreSQL. The top hierarchical unit of PostgreSQL is the database cluster which is managed by exactly one database server instance and contains (possibly) many databases (*18.2. Creating a Database Cluster*, 2021).

A database can then contain several schemas which can contain several tables. Schemas allow to separate database objects in groups in order to make them easier to manage (*5.9. Schemas*, 2021). A table, on the other hand, is a unit where data is stored in a row-based manner where each row has the same named columns (*Concepts*, 2016).

### JavaScript

JavaScript is a programming language that is mostly known as the scripting language of the world wide web and is used in a web site/page to define its behavior (*About JavaScript - JavaScript | MDN*, n.d.). JavaScript can also be coded server-side (*About JavaScript - JavaScript | MDN*, n.d.).

Technically, JavaScript is based on the ECMAScript programming language (*JavaScript Language Resources - JavaScript | MDN*, n.d.) for which the association ECMA International defines the standards (*Standards Archive*, n.d.).

### Typescript

TypeScript is a programming language that extends JavaScript with static types which provide an additional layer of security when programming applications since type errors can be detected at compile time (*Typed JavaScript at Any Scale.*, n.d.). Any JavaScript code is valid in TypeScript since it is an extension of JavaScript and, technically, even converted to JavaScript before its execution (*Typed JavaScript at Any Scale.*, n.d.).

### NodeJS

NodeJS is a JavaScript runtime environment which allows to build server-side applications (*Introduction to Node.Js*, n.d.). Technically, NodeJS runs on the V8 JavaScript engine which is the same engine that runs the Google Chrome browser (*Introduction to Node.Js*, n.d.). NodeJS runs on one thread and can execute code asynchronously. E.g., if an input/output operation is performed the thread will not get blocked, but instead continue to execute other operations until the input/output operation is done and can then be resumed (*Introduction to Node.Js*, n.d.).

#### NodeJS libraries

##### Winston logging library

The winston logging library aspires to be a simple but still universal and flexible logging library. It supports multiple log channels and message levels (*Winston*, 2010/2021).

Every winston logger instance can be configured to have several log channels. A log channel (in winston terminology called 'transport') is the destination of the log which can e.g., be the console or a file. For each log channel a message level ('silly', 'debug', 'verbose', 'info', 'warn' or 'error') can be specified. The specified message levels of the log channel and of the log message determine if an information will be logged or not. E.g., if the message level set on a log channel is 'info', a message will be logged if and only if it has the message level 'info' or higher (i.e., 'info', 'warn' or 'error') (Perry, 2019).

##### node-postgres

Application data will be stored in a PostgreSQL database (see subchapter 3.3). The interfacing between the server and the database is done using the node-postgres modules which allow to interact with PostgreSQL databases (Carlson, n.d.-c).

In order to connect to the PostgreSQL database either a connection Client or Pool (which contains a reusable list of connection clients) must be created. There are some advantages in using a connection Pool over a Client. One reason is that the use of connection Pools comes with a performance increase. The reason is that a connection Client executes one query at a time when connected to a PostgreSQL database which would lead to a lower performance compared to the connection Pool which manages the execution of several queries among its list of connection Clients. Another reason to use a connection Pool over a Client is that the handshake that is performed when connecting a Client to the PostgreSQL database is very time-consuming (20-30 milliseconds). These costs can be minimized since a connection Pool manages a (reusable) list of connection Clients (Carlson, n.d.-b).

The creation of a connection Pool (or a Client) requires connection information of the PostgreSQL database. This connection information includes the database user, password, host, port and name. This information can be provided in different ways e.g. by providing a connection string in the form 'postgresql://USER:PASSWORD@HOST:PORT/DB\_NAME' (Carlson, n.d.-a).

Database queries can then be executed by calling the query() instance method on the Pool and passing the query string as an argument (Carlson, n.d.-b).

##### Koa

Koa is a web framework for NodeJS that allows to create web APIs.

A Koa object contains an array of middleware functions which will be executed upon request. The middleware functions are specified by passing them as parameters to the instance method use() on a koa object. A specified middleware passes control downstream to the next middleware by invoking the async next() function. When there are no more middlewares to execute the control flow will go back upstream (*Koa - next Generation Web Framework for Node.Js*, n.d.).

[insert easy koa middleware example here]

There are several NodeJS modules that can be used with the Koa web framework. These modules include:

###### koa-router

Allows to configure routing which means that we can specify how the API will respond to a request to a certain endpoint (which is a HTTP method and a URI) (*Express Basic Routing*, n.d.). Thus, a router middleware function will only be executed if the request is targeted at its specified endpoint. We can specify a route on a koa-router by invoking its instance methods use(), get(), post(), del() or patch(). While the use() method will specify a route that can be matched regardless of the requests HTTP method, routes specified with the other methods (i.e. get(), post(), del() and patch()) can only be matched if the respective HTTP method was used for the request. We then have to provide to these methods a URI as parameter as well as a function that specifies how the request is handled. The specified route can only be matched if the requests URI matches with the beginning of the URI we specified on the route. The specified routes on the koa-router are then added as middleware to the koa object by invoking the use() instance method on the koa object and passing to it the routes middleware functions (which are retrieved with the instance method routes() on the koa-router) as parameters.

[insert good example for koa-router]

Additionally, the koa-routers allowedMethods() instance method returns a middleware that can handle OPTIONS requests.

###### koa2-cors

What is CORS?

CORS stands for Cross-Origin Resource Sharing. It is a mechanism based on certain HTTP-headers that permits a server to specify which origins (i.e., a combination of scheme, domain and port) other than its own can access its resources from a browser. In its simplest use, CORS is handled by setting two HTTP-headers, one request header (which is the Origin header) and one response header (which is the Access-Control-Allow-Origin header). The Origin header will always, as its name says, show the origin of the request. The Access-Control-Allow-Origin header will show which origins are allowed to access the resources of the server. If the Origin and Access-Control-Allow-Origin do not match the request will fail. There is also the possibility to send so called preflighted requests which allow to determine if the actual request can be sent without encountering a CORS failure (*Cross-Origin Resource Sharing (CORS) - HTTP | MDN*, n.d.).

Additionally, CORS can handle credentialed requests (i.e., with HTTP cookies) by setting the Access-Control-Allow-Credentials response header to true. If the Access-Control-Allow-Credentials response header is not set to true, the browser will reject the response (*Cross-Origin Resource Sharing (CORS) - HTTP | MDN*, n.d.).

The koa2-cors module provides CORS middleware for Koa. It allows to set several CORS headers.

###### Additional modules

Additional modules used in the development of the Controller are the koa-bodyparser which provides us a middleware that parses the body that was sent through the HTTP request (so it is accessible in the Koa app through ctx.request.body) and the koa-logger which logs information about requests and responses to the Koa app (thus can be useful for debugging the application).

#### Chokidar

Chokidar is a file watching library (*Chokidar*, n.d.). It concretely allows e.g., to specify what a program should do when a file is added to a folder.

#### bcrypt

A library to generate and check hashed passwords (*Bcrypt*, n.d.).

#### jsonwebtoken

This NodeJS module implements JSON Web Tokens (*Jsonwebtoken*, n.d.).

### Vue.js

The client-side application of the project was built using the Vue.js (or simply Vue) framework.

From a technical perspective, VueJS implements the Model-View-ViewModel (MVVM) design pattern which has three components (View, ViewModel and Model). The Model in Vue.js is represented by plain JavaScript objects while the View is represented by the DOM. The ViewModel sits in between the View and the Model and is responsible for syncing the data between them. The ViewModel sets up the data bindings on the View and gets notified when the Models data changes, achieving reactivity of the View and the Model (*Getting Started - Vue.Js*, n.d.).

One of Vues most important concepts is the components system. It allows to define self-contained and reusable elements that can be nested in other elements respectively nest other elements in them (*Introduction — Vue.Js*, n.d.) **(source: https://vuejs.org/v2/guide/#Composing-with-Components)**. Let's consider an arbitrary example: a travel blog. We have a root component which nests other components, e.g., a toolbar and the content. The content component has descriptive text in it and nests another component (which is a list of travel pictures). Each travel picture of the list is a component which is nested inside the list component. We see that this component system allows to hierarchically organize the frontend application.

A Vue component can hold several properties such as e.g., props (which are data that are passed from a parent component to a child component), data, methods and a template (*Components Basics | Vue.Js*, n.d.). The template property of a component has an HTML-based syntax that allows to declaratively render the component data to the DOM[[1]](#footnote-1) (*Template Syntax — Vue.Js*, n.d.). A component can be defined in a file (referred to as Single File Component) which has the .vue extension. The .vue file can specify a template using the template tag and add logic to the component inside the script tag (*Single File Components — Vue.Js*, n.d.).

The following two images provide an example single file component and how it is displayed. We can see how the data is declaratively rendered:

[definition of simple single file component]

[screenshot of display of previous code]

During the projects development other libraries and frameworks were used that are compatible with Vue. These include:

* Vuex
* Vue-router
* Vuetify

#### Vuex

Managing a common state across multiple Vue.js components can become tedious in more complex applications. The shared data must be passed as props down the component hierarchy to the components which use that data. Then every time the data (or props) modification is triggered in a component, that component must initiate an event chain which fires up the component hierarchy until it reaches the component that holds the shared data which can then finally be modified (*What Is Vuex? | Vuex*, n.d.).

Vuex is a state management library for Vue web applications. It provides a centralized store for the Vue.js applications shared data. The advantage of managing shared data in Vue with Vuex instead of the traditional way is that it allows any component to access (i.e., read and modify) data from the centralized store directly through the methods (in Vuex called getters, mutations and actions) which the programmer defines (*What Is Vuex? | Vuex*, n.d.).

#### Vue Router

Vue allows to conditionally render components (or other elements) using its conditional statements inside components templates. Unfortunately, as an application gets more complex, it becomes increasingly tedious to maintain those conditional statements across the entire application. The Vue Router allows to manage in a simple way which content is displayed when and reduce the number of conditional statements in the application.

Concretely, Vues official router allows to configure component-based routing (*Vue Router*, n.d.) i.e., depending on which route is active, a different component is displayed. Other features of the Vue Router include configuration of nested routes and route parameters (*Vue Router*, n.d.).

#### Vuetify

Vuetify is a User Interface framework for Vue. The framework is very complete and, among other features, allows to use and style its predefined UI components (*Why You Should Be Using Vuetify*, n.d.).

### Jest

Jest is a JavaScript testing framework (*Jest*, n.d.). It allows to easily set up test suites using its API. Functionalities of Jest include setting up (named) tests (*Getting Started · Jest*, n.d.), checking values using matchers (*Using Matchers · Jest*, n.d.), testing asynchronous code (*Testing Asynchronous Code · Jest*, n.d.), setup and teardown of tests (*Setup and Teardown · Jest*, n.d.) and creation of test doubles (which is replacement code for some parts of the production code (“Unit Testing in Node.Js,” 2019)) (*Mock Functions · Jest*, n.d.).

### Other Technologies

Some other important Technologies used are summarized in the Appendix B. These include JSON and XML.

## Development

Xxx

### Server

After considering the screen mockup, the general architecture and the use case diagrams the server application has been structured the following way.

[insert class diagram]

In the following sections the different parts of the server-side software are explained.

#### Model

At the core of the server is the data model. The model consists of several classes which have been designed to split the data into logical units. The classes that were developed are:

User

Represents a user of the application. It is needed to manage logging in to the web application as well as to manage the access to the different projects (Project). Each user has an email, a password (which will be hashed by the Controller before it is stored) and a list of projects (Project) which specifies which projects the user can access. Additionally, it is specified if the user is an administrator or not (administrators have access to all projects).

MapNode

Represents a node in a map. Each node has an id (which is unique), a x-coordinate, a y-coordinate and a population size. Additionally, it is specified on each node if it can serve as a vehicle depot of a vehicle (VehicleType) in a result (Result) and if it can serve as a waste depot in a collection point scenario (CollectionPointScenarioVersion).

MapArc

Represents an arc in a map. It connects two nodes (MapNode), a source node and a destination node, and it stores the distance between those two nodes.

Graph

Represents a map of a municipality. It consists of a list of nodes (MapNode) and a list of arcs (MapArc).

GarbageScenarioVersion

Represents a concrete garbage scenario. It contains a timing and an estimation of waste on each node of the graph.

GarbageScenario

Represents a garbage scenario with all its concrete garbage scenarios (GarbageScenarioVersion). It contains the title of the garbage scenario and a list of concrete garbage scenarios (GarbageScenarioVersion). Every time a garbage scenario is modified (i.e. the waste estimation of a node (MapNode) is changed) a new concrete garbage scenario (GarbageScenarioVersion) will be created and added to the list of concrete garbage scenarios.

CollectionPointScenarioVersion

Represents a concrete collection point scenario. It contains a timing and it specifies on each node if it is a potential collection point or not.

CollectionPointScenario

Represents a collection point scenario with all its concrete collection point scenarios (CollectionPointScenarioVersion). It contains the title of the collection point scenario and a list of concrete collection point scenarios (CollectionPointScenarioVersion). Every time a collection point scenario is modified (i.e., the specificatio if a node (MapNode) is a potential collection point or not is changed) a new concrete collection point scenario (CollectionPointScenarioVersion) will be created and added to the list of concrete collection point scenarios.

VehicleTypeVersion

Represents a concrete vehicle type. It contains a timing, an average speed for a tour, an average speed when going to the depot, an average stop time and a vehicle capacity. Additionally, it specifies on each arc if the vehicle type can drive on it or not.

VehicleType

Represents a vehicle type with all its concrete vehicle types (VehicleTypeVersion). It contains the title of the vehicle type and a list of concrete vehicle types (VehicleTypeVersion). Every time a vehicle type is modified (i.e., the specification if the vehicle type can drive on an arc (MapArc) is changed) a new concrete vehicle type (VehicleTypeVersion) will be created and added to the list of concrete vehicle types.

Tour

Represents a tour calculated by the waste collection optimization algorithm following a solution request from the user through the web app. It belongs to a result (Result). It contains a timing, the time it takes to accomplish the tour, the estimated total waste collected during the tour, the tour nodes with their order and with their estimated collected waste and the concrete vehicle type (VehicleTypeVersion) that performs the tour.

Facility

Represents the location of a facility on the map calculated by the waste collection optimization algorithm following a solution request from the user through the web app. It belongs to a result (Result). It contains a node (MapNode) and a waste capacity.

Result

Represents the requested solution from the user through the web app. The input data from the user and the output data from the waste collection optimization algorithm are stored here. It contains a timing, a concrete garbage scenario (GarbageScenarioVersion), a concrete collection point scenario (CollectionPointScenarioVersion), a list of concrete vehicle types (VehicleTypeVersion) with its possible waste depot nodes (MapNode), the model (i.e. 'k1', 'k2', 'k3'), the maximal walking distance for a citizen to deposit his/her garbage, the minimal waste capacity of a facility (only for the model 'k2'), the total cost of the solution, a list of tours (Tour), a list of facilities (Facility) and a boolean value specifying if the solution has been already calculated by the waste collection optimization algorithm. When a solution gets requested by a user, a result will get created using the public static async createResult method which also writes an input XML-file with the data given in by the user. The input XML-file is left in a folder and the waste collection optimization program then gets executed with the file provided as argument. The waste collection optimization program will leave an output XML-file in another folder as soon as it has calculated the results. The output XML-file will then be read and added to the result using its public setResultData() method by the controller.

Project

Represents an entire project. It contains the project title, a list of users that can access the project (User), a map (Graph), a list of garbage scenarios (GarbageScenario), a list of collection point scenarios (CollectionPointScenario), a list of vehicle types (VehicleType) and a list of results (Result). When creating a new project two SQL files are written, one for setting up the new project schema in the database (which is immediately executed) and the other for deleting it from the database (in case it will be necessary in the future).

Model

The Model class represents all projects and all users of the application and was developed using the Singleton software development pattern, i.e. it assures that there is exactly one model instance which can be retrieved using its public static createModel() method. It contains a list of all users and all projects and, when created, also sets up the projects-users connections (by adding them to the user or project lists inside the projects respectively users instances). It handles the adding and deleting of projects and users from the application.

Each of these classes generally possess a (static) function to create instances of them and write the corresponding data to the database. Except for the Model class these classes also contain a (static) function which, when invoked, reads the database and returns the respective instances of that class. Additionally, these classes contain various getters and setters as well as adders and deleters (i.e. functions that add elements to an array variable or delete them from the array variable). If an object is part of another object, a reference is stored to the latter object. E.g., an arc (MapArc) must be part of a map (Graph), therefore the arc (MapArc) stores a reference to that map (Graph).

#### Controller

Why developing a Controller?

In order to have a properly functioning waste collection application server there are two needs that must be satisfied. Firstly, the server needs an interface in order to allow the client-side application to interact with it. Secondly, when the server receives information from the client-side application (and the waste collection optimization algorithm) through its interface, it needs to update the waste collection application model based on the information it receives. Those two needs are satisfied by developing a Controller.

Theory

For the client-side application to interact with the server a REST web API using the Koa web framework was developed. The data is delivered to the client in JavaScript Object Notation (JSON) format.

REST API

An API (application programming interface) is a set of rules which determine how different programs can communicate with each other (*What Is an Application Programming Interface (API)*, 2020). An API allows that an application accesses a resource from another application. The application requesting a resource is called client while the application containing a resource is called server (*Rest-Apis*, 2021). Today’s most common APIs are web APIs which allow access to resources over the internet (*What Is an Application Programming Interface (API)*, 2020).

An API functions by a client sending a request to the API through its URI (Uniform Resource Identifier). The API then calls the server after receiving the request. The server then returns a response to the API following the request of the client. The API then sends the data it received from the server to the client (*What Is an Application Programming Interface (API)*, 2020).

[insert graph explaining functioning of API]

REST (Representational State Transfer) which was first proposed by Fielding (2000) in his doctoral thesis is an architectural style for distributed hypermedia systems.[[2]](#footnote-2) Roy Fielding derived REST by imposing six architectural constraints from a few network-based architectural styles and thus described it as a hybrid style. The six architectural constraints are:

Client-Server Separation

Leads to a separation of concerns. The user interface and the data storage are separated from each other.

Statelessness

Enforces that each client request contains all the necessary information to process it. There cannot be any stored context on the server.

Cache

Response data must be labeled as cacheable or not. Cache is data that can be reused by the client for future, equivalent requests and can thus reduce the client-server interactions.

Uniform interface

Data is exchanged through a standardized interface.

Layered System

REST allows architectural components to be structured hierarchically where each component exclusively knows the component that it is interacting with.

Code-on-demand

This is the only optional constraint. It allows clients to download executable code and thus extend its functionalities.

**(cite Roy Fieldings properly)**

While Fielding (2000) developed REST from a theoretical perspective, his work gives little insight on how to practically implement a concrete REST Web Service. According to Rodriguez (2008) nowadays a concretely implemented REST Web Service follows four principles. These are:

Statelessness

As already specified by Fielding (2000), each request contains all necessary information.

In order to achieve a stateless service, the client is entirely responsible for storing any session state. When the client makes a request, it needs to include in the HTTP headers and body every piece of data that the server needs in order to respond i.e., the client shall make few assumptions about the server adding any context to the request (Rodriguez, 2008).

Use HTTP methods

The HTTP methods POST, GET, PUT and DELETE correspond to the CRUD operations (create, read, update, delete). Each HTTP method should only be used to execute the CRUD operation it is mapped to, i.e., a POST request creates a resource, a GET request reads a resource, a PUT request updates a resource and a DELETE request deletes a resource (Rodriguez, 2008).

Directory structure-like URIs

The design of REST Web Service URIs should be easy to understand (or even self-explanatory). This is achievable by defining directory structure-like URIs. E.g., the URI "http://www.myservice.org/discussion/topics/{topic}" exposes discussions about the topic "{topic}". On the other hand, if we wanted to expose the discussions in a certain language we would expose the URI "http://www.myservice.org/discussion/languages/{language}" (Rodriguez, 2008).

Send JSON or XML (or both) requests/responses

JSON and XML formats allow to present data objects in a simple and human-readable form. Resources should be transferred in one of these two data interchange formats (Rodriguez, 2008).

JSON Web Token (JWT)

JWT is an open standard that allows the secure transmission of data between parties. JSON Web Tokens can be digitally signed using a secret key with the HMAC (Hash-based Message Authentication Code) algorithm. JWTs are most used for handling Authorization. When the user logs into an application a JWT can be returned which then has to be sent by the client to the server every time a resource is accessed. The server then needs to authenticate the user by verifying the JWT and check if the user is authorized to access that resource (auth0.com, n.d.).

Implementation

The Controller class handles the interactions with the server and was developed using the Singleton software development pattern, i.e. it assures that there is exactly one Controller instance which can be retrieved using its public static createController() method. The public static createController() method stores a reference to the data model (i.e. all projects and users) in a variable and sets up the Koa object by adding the middlewares to it. Additionally, a file watch is set up which handles the file outputs of the waste collection optimization program.

The middlewares added to the koa object include the koa-logger, koa-bodyparser, koa2-cors and the koa-routers routes that were set up. A middleware that responds to OPTIONS requests was also added.

The routing middleware functions that were set up can be split in two groups. In the first group we have middleware functions that have generic preparatory purposes while in the second group we have specific handlers for our endpoints. The difference between these two types of functions can be easily understood with an example. When a GET request to the URI '/api/protected/project/fribourg' is made the Controller needs to first authenticate the user, then check if the user is allowed to access the project 'fribourg' and then send the data of the project 'fribourg' back. When a DELETE request to the URI '/api/protected/project/bern' is made the Controller needs to first authenticate the user, then check if the user is allowed to access the project 'bern' and then delete the project 'bern'. We see a pattern here. The first two steps of the GET and the DELETE request are almost the same (only the projectname being different). The first step needs to be executed every time a request is made to a URI starting with '/api/protected' while the second step needs to be executed every time a request is made to a URI starting with '/api/protected/project/{projectname}'. This means that we can set up generic preparatory middleware functions that are executed when a request is made to a URI that starts with those strings. On the other hand, the specific handlers of our endpoints is then the middleware function that handles (in our example) the retrieving of the project 'fribourg' data and the deletion of the project 'bern'.

An overview of the generic preparatory middlware functions as well as the specific handlers of our endpoints can be found in the Appendix A.

#### Database handler

Why developing a Database handler?

There are two reasons a Database handler class was created. The first reason is to allow to easily query the database without needing to think about its usual intricacies (e.g. creating a Pool, providing connection information, etc.) just by using one querying() method. The second reason is to allow the creation and deletion of projects which are complex tasks as it involves the creation, execution and deletion of sql-files.

Implementation

The DatabaseHandler class handles the interactions with the PostgreSQL database and was developed using the Singleton software development pattern, i.e. it assures that there is exactly one DatabaseHandler instance which can be retrieved using its public static getDatabaseHandler() method. The public static getDatabaseHandler() sets the connection Pool as well as the users and projects schema (including its tables) in the database up, if necessary.

The DatabaseHandler contains a public async querying() instance method which, after being called with a query string as argument, executes a database query. Additionally, the class contains the public async setupProject() and public async deleteProject() instance methods that, after being called with the project title string as argument, setup or delete a project from the database, respectively. The setup of a project involves the creation of two sql-files, one for setting up and one for deleting the database schema of the project and its tables. The setup sql-file will be executed after its creation (in order to setup the database schema and its tables). The deletion sql-file will be executed when (and if) the project is deleted using the public async deleteProject() instance method. The public async deleteProject() instance method will also delete both sql-files that were created by the public async setupProject() instance method.

#### Logger

What is and why develop a Logger?

When any kind of applications run, errors can happen and in that case the programmer needs to debug the application. In order to debug, he needs information about the state of the application at the point of failure and how the application reached that state. Only then is the developer able to fix the application.

Logging is the writing of diagnostic information to protocols which can be very helpful when debugging because it provides information on how the program reached the state of failure to the developer (Perry, 2019).

There are some libraries in Node.js (e.g., Winston or Log4js) that allow to manage logging in a simple way. Technically, logging could be done by simply using plain JavaScript, but logging libraries provide some functionalities that would be time-consuming to code from ground-up (e.g., severity levels of messages).

Implementation

The Logger class handles the logging of the application and was developed using the Singleton software development pattern, i.e. it assures that there is exactly one logger instance which can be retrieved using its public static getLogger() method. The logger instance contains two winston logger instances. One of these winston logger instances writes database query informations while the other is responsible for all other loggings. The winston logger instance for database query information has one log channel (which is a file log channel) and its message level is set to 'silly' (i.e., all messages will be logged). The winston logger instance for non-database query information has three log channels two of which are file log channels with the message level set to 'silly' and 'info' respectively and one of which is a console log channel with the message level set to 'info'.

The programmer has to choose between two instance methods of the logger when he wants to log a message which are the public dbLog() and public fileAndConsoleLog() methods. For both of these methods the developer has to pass two parameters to the function, the information he wants to log and its severity (i.e., the message level). The dbLog() method should be used if the log message comes from an interaction between the server and the database. Otherwise the fileAndConsoleLog() method should be used.

The reason the logger instance contains two winston loggers was to ensure that the log files are split by information source. When debugging, this allows to independently analyze the database interactions and the other program errors/informations without each of them polluting each other’s log files. The reason the winston logger for non-database query informations has three log channels is that it ensures that there are log files with all non-database query logs and log files only with logs of higher severity levels. The log files with the logs with higher message levels allow to more quickly find error logs when debugging (since these log files are not polluted with logs of low message levels). Finally, the winston logger for non-database query informations has a console log channel in order to immediately notify the programmer that the application encountered a problem.

The winston loggers are updated every day before the first log is performed. Concretely, the filenames of the file log channels will be updated and named after today’s date. Therefore, the logs from different days will be written in different log-files which further allows the developer to more quickly find relevant logs.

**(Check if needed to cite some things. See e.g., IBM winston logger tutorial)**

### Database

Why a Database?

The data that the user defines needs to be stored in order to be permanently available.

Implementation

In the PostgreSQL database cluster used for this project one database was specified. This database was named 'wastecollectiondata'. The schemas contained in the wastecollectiondata database are:

-A 'usersprojects' schema: This schema contains the tables users, projects and userprojects. These tables store the users data (email, admin status and password), projects data (projectname) and the user-project connections (i.e. which users can access which projects) respectively.

[show database schema of usersprojects schema]

-Projects schemas: Each project has its own schema named after the projectnames in the usersprojects.projects table. It generally corresponds to the server data model.

[show database schema of a project schema]

### Client

Overview

The prototype that was created during the requirements engineering process was used as a blueprint for the development of the client-side application.

The frontend was developed as a Single-Page Application. A Single-Page Application is a web application in which every interaction with it happens on one web page. In a Single-Page Application requests of small pieces of data to the server happen and are then used to modify the DOM. This is very different from a classical (multi-page) web application where after every interaction the browser loads a new web page from the server (Mesbah, 2009, p. 10).

Implementation

The state (data) of the Vue application as well as the connection to the server is managed in the Vuex store. The components connect to the store in order to retrieve (using Vuex' getters) and modify (using Vuex' actions) data. When an action from the Vuex store is invoked by a component, it sends a HTTP request to the backend server in order to post, delete or update data on the server. Additionally, after the POST, DELETE or PUT request was successful, the invoked action will also perform GET requests in order to update the state of the Vuex store.

The components responsibility is to retrieve the data they need from the store, process it, display it and, depending on user interactions with the component, trigger state modifications by invoking the Vuex' stores actions. UI components from Vuetify were used in components templates.

On the highest hierarchical level, the Vue application contains two components (defined in the App.vue file). One of these components is defined by the Navbar.vue file which defines two elements, a navigation header and drawer. The second displayed component depends on which route is active since the App.vue file defines a router entry point.

The routes that were defined on the vue router include:

-'public' routes, i.e., routes that can be accessed even if the user is not logged in. Those routes include e.g. the home route '/' which displays general information and '/loginorsignup' which allows the user to log in or sign up to the application.

-'protected' routes which always start with '/protected' and can only be accessed if the user is logged in. Protected routes contain several nested routes which include:

-several routes that display projects and users data (e.g. the '/protected/garbagescenarios' route displays the data of garbage scenarios from the currently selected project)

-the '/protected/calculateresults' route which displays the CalculateResults component which contains a form that allows to request results from the server.

-the '/protected/newproject' route that displays the NewProject component which allows to create a new project (can only be accessed by users with admin rights)

-several 'view' routes which display data of specific objects of the project.

E.g. the '/protected/view/garbagescenarios/:title/timestamp/:timing/viewmode/:viewmode' route displays data of a specific garbage scenario of the currently selected project. The specific garbage scenario that is displayed is determined by the parameter values of :title and :timing. The parameter value of :viewmode (can take the string values 'edit', 'oldversion', 'createnew' and 'readonly') is used inside the component to determine if certain elements should be disabled or not.

-several 'history' routes which displays lists of garbage scenario, collection point scenario or vehicle type versions.

E.g. the '/protected/history/garbagescenarios/:title' displays a list of the different garbage scenario versions of the garbage scenario with the title that is passed to the parameter :title.

Which route is active depends on the latest user interactions with the app. The workflows of the application are explained in the chapter 'Manual' in more detail.

## Testing

What is Testing?

Testing (of software) is the process of verification that a software works the way it is supposed to work. It can prove useful in order to prevent bugs (*What Is Software Testing?*, 2019).

There are several types of testing. The types of testing that were used in this project are acceptance and integration testing. While integration testing verifies if several software components work correctly together, acceptance testing checks if the software system as a whole functions the way it is supposed to (*What Is Software Testing?*, 2019).

Methodologies

The server as well as the client were tested during the development of the software. While the server was tested using mostly automated testing (with Jest) the client was tested manually. Test suites were written to test the different classes on the server. In these test suites not all but the 'critical' methods of the classes were tested. The critical methods are those that were judged more likely to fail. A method would be critical if it contains complex code (which means that the likelihood of the programmer implementing the code wrongly is higher) or code that connects with other software components (database, other classes). Simple getters and setters were not considered critical and thus not tested. Test doubles were developed for cases where the inclusion of production code was not practical and necessary.

While the testing of the server would be considered integration testing (i.e. testing if different components work correctly together) the client-side testing allowed to test the system as a whole (i.e. acceptance testing) **(do I need to cite that again?)** because the client makes requests to the server which in turn connects to the database (and the waste collection optimization program (maybe delete this because it was not tested)) thus involving the whole system. Testing on the client-side was done manually meaning that different test cases (e.g., logging in to the application, creating a new project, requesting solutions, etc.) were tested by clicking through the frontend application.

[components of the project]

# Manual

The developed DSS web app offers several functionalities. On the one hand, it offers common web app functionalities to the user including register, log in and log out functionalities. One the other hand, the app offers project specific functionalities such as creation of a project, management of projects, management of users and management of a particular project (which includes management of garbage scenarios, management of collection point scenarios, management of vehicle types and management of results). This chapter explains how to operate the web app by going in detail through the different functionalities which the app offers.

## Common web app functionalities

The user must, before he does anything else with the web app, sign up and then log in to the application. As soon as he accomplished the log in, he can exploit the waste collection-specific functionalities. When the user is finished using the application, he can log out.

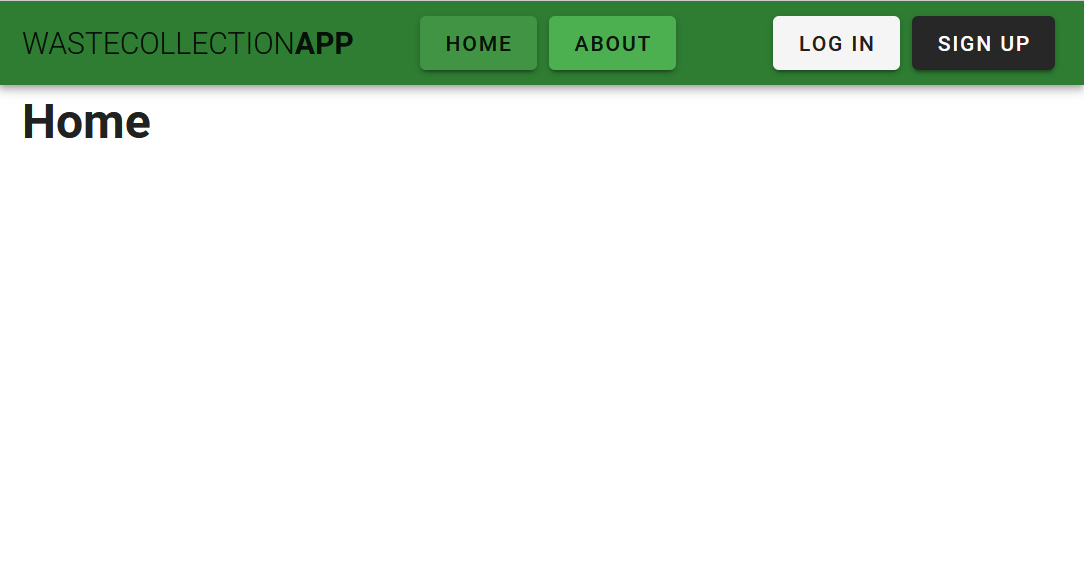


Figure 1: Homepage of the web app

Figure 1 shows the homepage of the web app. This is the interface the user sees when first accessing the web app in the browser.

### Sign Up

The user accesses the sign-up form by clicking on the ‘Sign Up’ button on the right side of the navigation header.

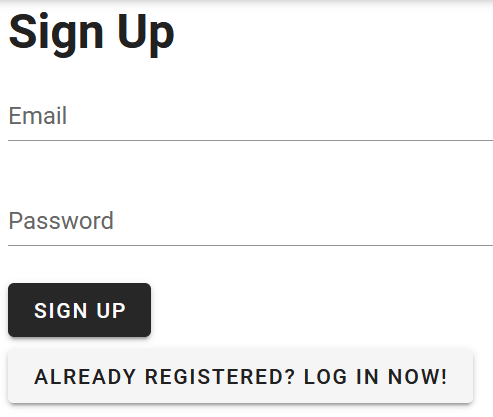


Figure 2: Sign Up form

The user fills out the form with an email address and a password. Then finishes the registration by clicking on ‘Sign Up’ below the form.

### Log In

A registered user can log in to the application. In order to do so, he accesses the log in page by clicking on ‘Log In’ in the navigation header.

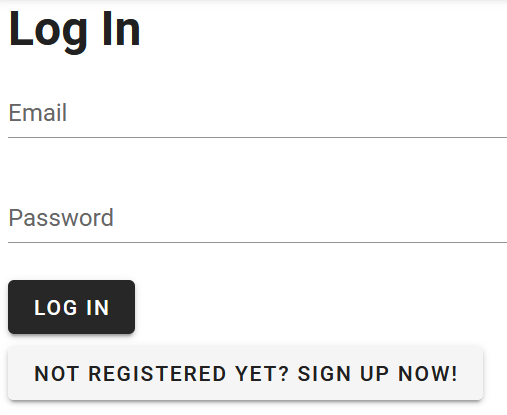


Figure 3: Log In form

Now the body of the web app displays a form. The user fills it out with his credentials and concludes the login by clicking on ‘Log In’ below the form.

### Log Out

As soon as the user is logged in to the application, a ‘Log Out’ button gets displayed on the right side of the navigation header. The user clicks on that button in order to log out. If the user does not log out manually, he will be automatically logged out 24 hours after his last login.

## Waste collection-specific functionalities

The user can, after logging in, exploit the specific functionalities of the waste collection web app. These functionalities include the creation of a project (only for administrators), management of projects, management of users and management of a particular project (which includes management of garbage scenarios, management of collection point scenarios, management of vehicle types and management of results).

It’s important to note that a navigation drawer gets displayed to the logged in user on the left side of the interface. The waste collection app specific functionalities are accessed through that navigation drawer.

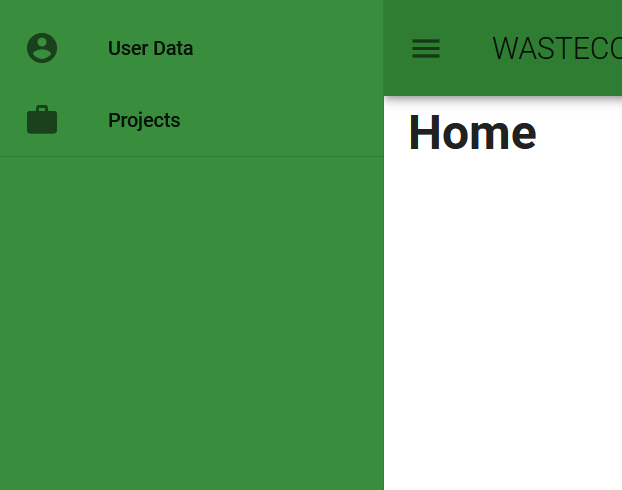


Figure 4: Navigation drawer after the users' login

### Creation of a project

Only administrators can create a project.

Graphical user interface, text, application

Description automatically generated

Figure 5: Navigation drawer for user with administrator rights

Only a user with administrator rights will see a ‘Create New Project’ button in the navigation drawer (see Figure 5). By clicking on that button, the form to create a new project will get displayed.

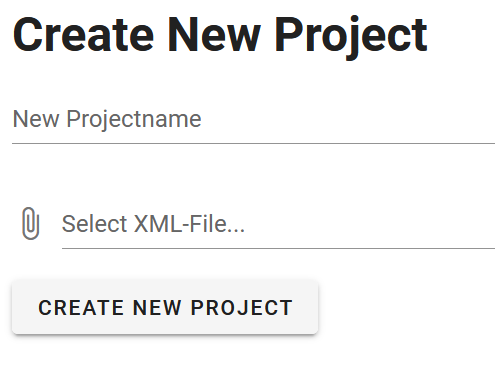


Figure 6: Create new project form

The administrator user now gives a title to the new project and uploads an XML file which contains municipalities’ map data. The user creates the project by clicking on ‘Create New Project’ below the form.



Source code 1: Example XML file for the creation of a new project

### Management of users

By clicking on ‘User Data’ in the navigation drawer a list of users gets displayed in the body of the web app. The list contains only one user (which is the logged in user) if the logged in user is not an administrator.

Graphical user interface, website

Description automatically generated

Figure 7: Users management in the web app

The users displayed in the list can be deleted by clicking on the corresponding ‘Delete User’ button. Additionally, the logged in user can view the projects which a listed user has access to by clicking on ‘See Projects’ in a user list element. Then, the user can add users’ rights to modify a project (only if the logged in user is an administrator) by selecting the project and clicking on ‘Add Project’ and remove users’ rights to modify a project with the corresponding delete button. Lastly, the user can access the functionality of managing a particular project by clicking on ‘Modify Project’.

### Management of projects

By clicking on ‘Projects’ in the navigation drawer a list of projects gets displayed in the body of the web app. The list contains only those projects which the logged in user is allowed to access.

Graphical user interface, application, website

Description automatically generated

Figure 8: Projects management in the web app

The projects displayed in the list can be deleted by clicking on the delete button (only if the user is an administrator). Additionally, the user can view the users which can access a listed project by clicking on ‘See Users’ in a project list element. Then, if the logged in user is an administrator, a user can be added to or deleted from a project with the ‘Add User’ respectively ‘Delete User’ button. Lastly, the user can access the functionality of managing a particular project by clicking on ‘Modify Project’.

### Management of a particular project

The management of a particular project includes the management of garbage scenarios, collection point scenarios, vehicle types and results. This section will briefly describe each of those functionalities. The management of garbage scenarios, collection point scenarios and vehicle types are similar and therefore grouped together.

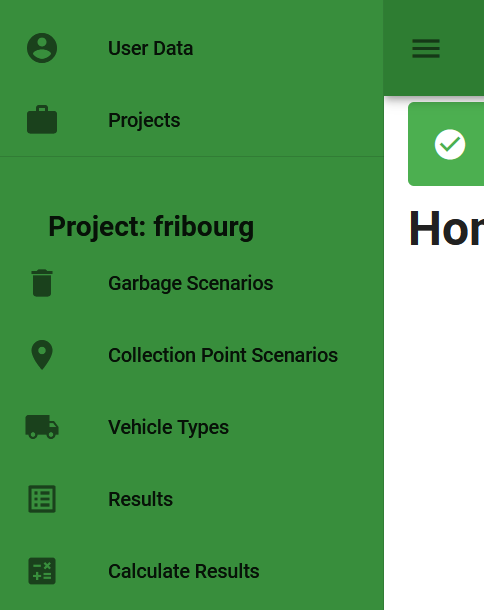


Figure 9: Navigation drawer after selecting a project

Before the user can request a result, he needs to first wait for an administrator user to create a project and then define at least one garbage scenario, one collection point scenario and, depending on the chosen model of the result, zero/one/several vehicle types.

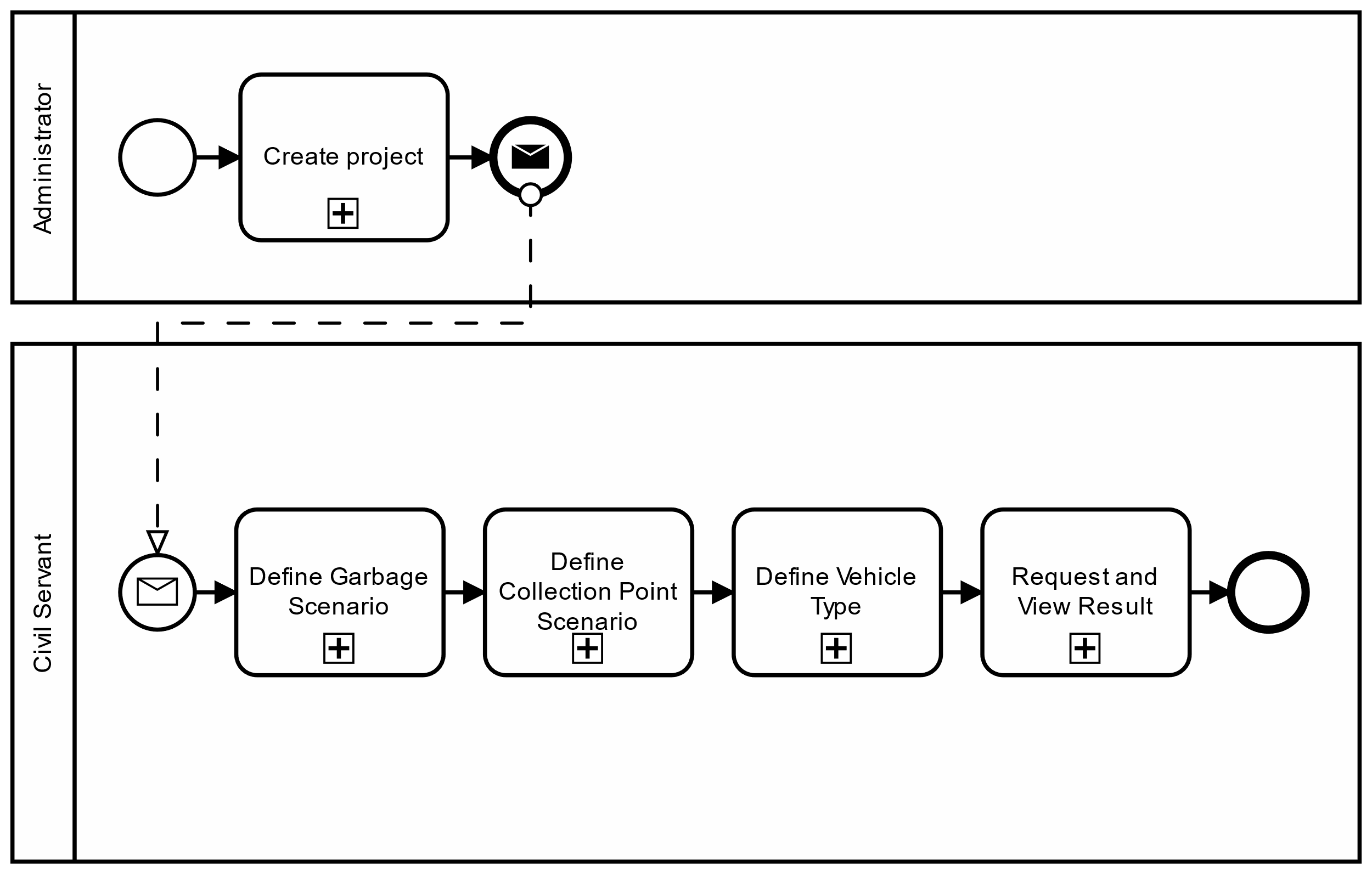


Figure 10: Simplified workflow of the app

#### Management of garbage scenarios, collection point scenarios and vehicle types

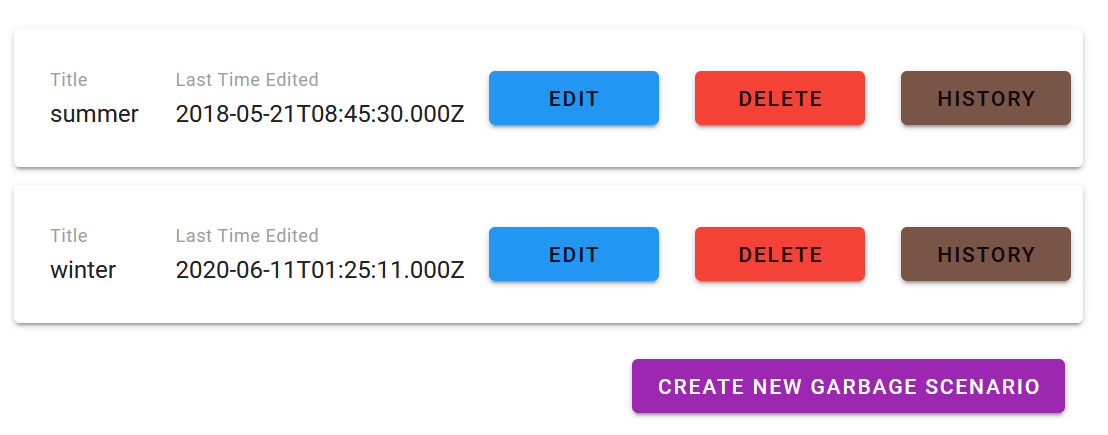
The user can manage garbage scenarios, collection point scenarios and vehicle types by clicking on the corresponding button in the navigation drawer. A list of the corresponding scenarios/types appears (if the project has any defined).

Figure 11: Garbage scenarios list example

The user can edit, delete and view the history of a scenario/type. Additionally, he can create a new scenario/type.

When the user creates or edits a scenario/type, he gets directed to a form where he specifies the data of that scenario/type. The data includes always the title of the scenario/type. For garbage scenarios estimates of waste on each node are given, for collection point scenarios each node is marked to be a potential collection point or not and for vehicle types the average tour speed, average depot speed, average stop time and vehicle capacity are set as well as each arc is marked to be drivable for the vehicle or not.

Graphical user interface, application

Description automatically generated

Figure 12: Edit vehicle type example

Every time a scenario/type is edited, a new scenario/type version is created. A list of the versions can be viewed with the ‘History’ button. By clicking on ‘See Details’ a version can be viewed and then restored by clicking on the save button at the end of the page.

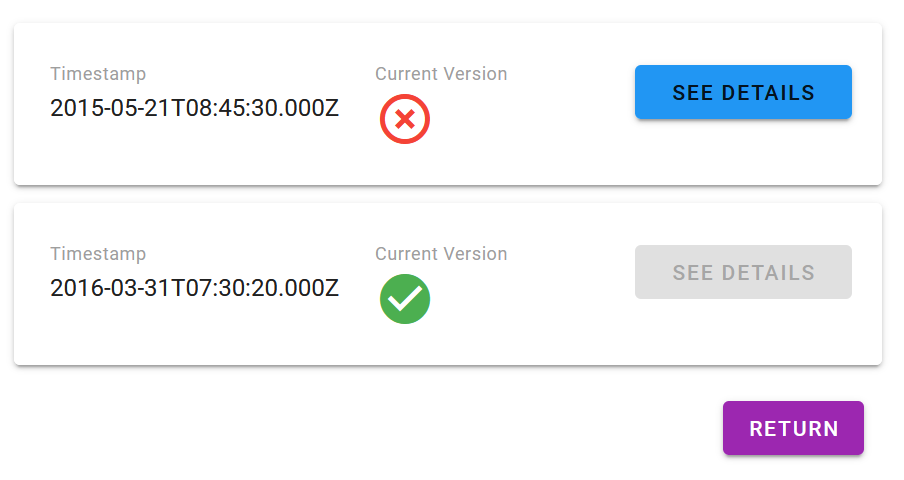


Figure 13: History example collection point scenarios

#### Management of results

A list of results is accessible through ‘Results’ in the navigation drawer.

Graphical user interface, text

Description automatically generated with medium confidence

Figure 14: Results list

Each element in the list contains information on the date of the result request, the model, the maximal walking distance, the minimum waste (if applicable) and the completeness (i.e., if the solution was already calculated by the waste collection optimization algorithm). The user can delete a result with the delete button and view a result with the ‘See Details’ button.

The view of a result contains, on one hand, the same information as in the results elements list. On the other hand, it contains information on the input scenario/type versions of the result (including a mark showing if the respective versions are the newest scenario/type versions of the respective scenario/type) and the tours and facilities data as (and if already) calculated by the waste collection optimization algorithm. The user can view a results’ scenario/type version data by clicking on ‘See Details’. Additionally, the chosen available waste depot nodes are displayed for each vehicle type version. Lastly, the user can request an updated result (which includes the newest scenario/type versions of each scenario/type) by clicking on ‘Update Result’ at the end of the page.

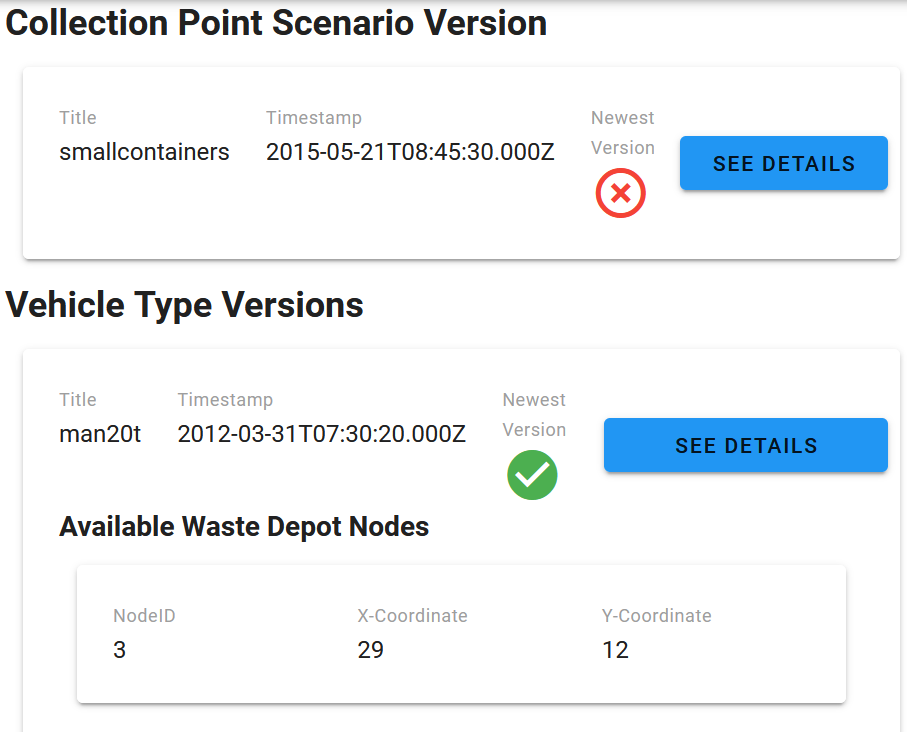


Figure 15: Example clipping of result view

A completely new result can be requested by clicking on ‘Calculate Result’ in the navigation drawer.

When requesting a new result, the user firstly chooses a model and then, depending on the model, specifies the input data of the result. Concretely, the user defines the maximal walking distance, the minimum waste (if applicable), the garbage scenario, the collection point scenario and the vehicle type(s) (if applicable). In order to be able to choose scenarios/types the user needs to define them first (see chapter 4.2.4.1). Additionally, when a vehicle type is specified as result input, the user needs to choose the available waste depot nodes among the given potential waste depot nodes.

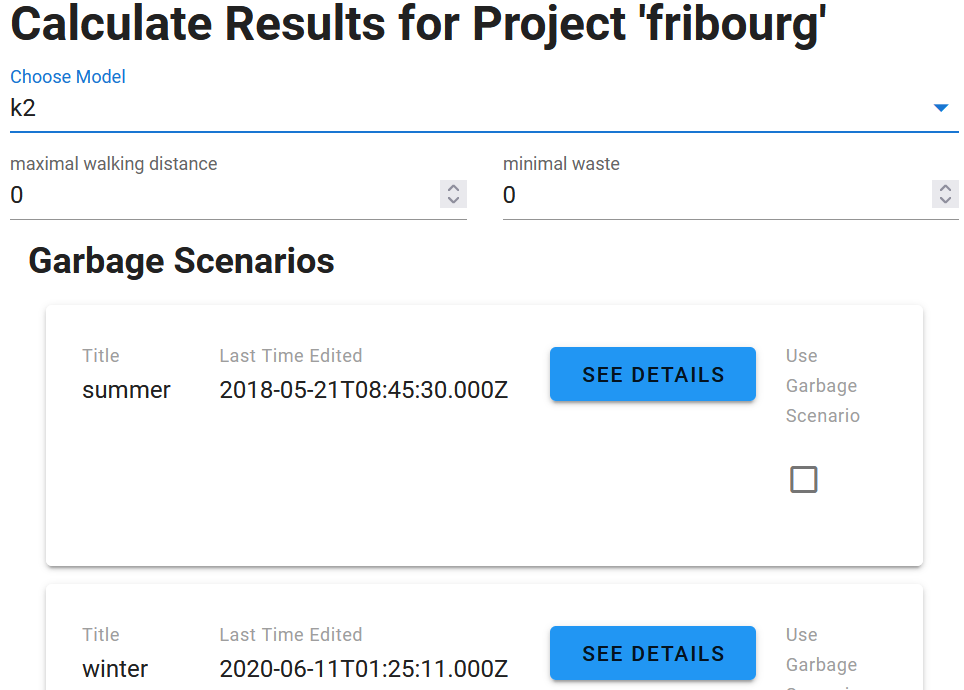


Figure 16: Clipping of the 'Calculate Result' view

The user requests the result by clicking on ‘Send Data’ in the bottom of the page. Alternatively, he can click on ‘Send Data and Request next Result’ in order to request the next result.

# Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

[…]

The development of this waste collection DSS web app is still in progress. The aim was, and continues to be, to create a DSS that provides useful information to waste collection management decision makers in various Swiss municipalities. As we have seen in the literature review, there are numerous design factors that determine the uptake of a DSS among its clients. These design factors should be considered in the continuing development of the web app.

Concretely, in order to increase the ease of use of the application a feature will be implemented which allows the user to interact with a map when he needs to manage scenarios or types instead of dealing with them in a table based manner.

# Appendix

[1. Appendix A: Image 12](#_Toc79017897)

[2. Appendix B: Table 13](#_Toc79017898)

[3. Appendix C: Code 14](#_Toc79017899)

1. Appendix A: Middleware functions

The specified generic preparatory middleware functions are:

/api/protected

Reads the JSON Web Token of the request (which is sent as a cookie) and verifies it in order to authenticate the user.

/api/protected/project/:projectname

Most importantly, finds the Project object with the provided projectname in the URI and checks if the user is allowed to access that project.

/api/protected/project/:projectname/user/:email

Most importantly, finds the user object with the provided email in the URI and checks if the user, who makes the request, is allowed to add or delete that user to respectively from the project.

/api/protected/user/:email

Most importantly, finds the user object with the provided email in the URI and checks if the user, who makes the request, is allowed to access that user.

The following handlers of our endpoints accept in some cases request bodies. The schema of the request bodies is visualized by using the TypeScript type declaration of objects. E.g. the following body has a property 'title' of type string and a property 'count' of type number:

{ title: string; count: number; }

The specified specific handlers of our endpoints are:

POST /api/public/register

Request body: { email: string; admin: boolean; password: string; }

The new user gets created using the users data from the request body. The password is hashed before creating the user.

POST /api/public/login

Request body: { email: string; password: string; }

The user object is found and the password from the request body is then compared with the stored hashed password. If the request bodys password is correct, a JSON Web Token gets signed and sent to the client as a cookie.

GET /api/public/logout

No request body necessary.

User gets logged out by setting the cookie holding the JSON Web Token to an empty string which expires immediately.

POST /api/protected/newproject/:projectname

Request body: { xml: string; }

Creates a new project with the projectname provided in the URI. The map (graph) of the municipality gets set using the xml data in the request body. This endpoint is only accessible to administrators.

DELETE /api/protected/project/:projectname

No request body necessary.

Deletes the project with the projectname provided in the URI. This endpoint is only accessible to administrators.

PUT /api/protected/project/:projectname

Request body: { newProjectname: string; }

Finds the project with the projectname provided in the URI. Then updates its projectname using the newProjectname value from the request body.

GET /api/protected/project/:projectname

No request body necessary.

Finds the project with the projectname provided in the URI. Then returns its data as a JSON object which includes nodes, arcs, garbage scenarios, collection point scenarios, vehicle types, results and users data.

GET /api/protected/projects

No request body necessary.

Returns data that describes which user(s) can modify which project(s). This endpoint is only accessible to administrators.

DELETE /api/protected/user/:email

No request body necessary.

Deletes the user with the email provided in the URI.

PUT /api/protected/user/:email

Request body: { newEmail: string; newAdmin: boolean; newPassword: string; }

Finds the user object with the email provided in the URI and updates its data to the values provided in the request body. The new password gets hashed before the update.

GET /api/protected/user/:email

No request body necessary.

Finds the user with the email provided in the URI. Then returns its data as a JSON object which includes its email, admin status (true/false), (hashed) password and the projects the user is allowed to access.

GET /api/protected/users

No request body necessary.

Returns data that describes which user(s) can modify which project(s). This endpoint is only accessible to administrators.

POST /api/protected/project/:projectname/user/:email

No request body necessary.

Adds the user with the email provided in the URI to the project with the projectname provided in the URI (i.e. the user gets the rights to access the project).

DELETE /api/protected/project/:projectname/user/:email

No request body necessary.

Deletes the user with the email provided in the URI from the project with the projectname provided in the URI (i.e. the user loses the rights to access the project).

POST /api/protected/project/:projectname/garbageScenario

Request body: { title: string; nodesWaste: { nodeid: number; wasteEstimation: number }[] }

Creates a new garbage scenario with the title provided by the request body and then adds a new garbage scenario version to it using the nodes waste data from the request body.

DELETE /api/protected/project/:projectname/garbageScenario/:title

No request body necessary.

Deletes the garbage scenario with the title in the URI from the project with the projectname in the URI.

PUT /api/protected/project/:projectname/garbageScenario

Request body: { title: string; nodesWaste: { nodeid: number; wasteEstimation: number }[]; newTitle: string; }

Updates the title of the garbage scenario with the new title provided in the request body and adds a new garbage scenario version to it using the nodes waste data from the request body.

POST /api/protected/project/:projectname/collectionPointScenario

Request body: { title: string; nodesPotCP: { nodeid: number; potentialCollectionPoint: boolean }[] }

Creates a new collection point scenario with the title provided by the request body and then adds a new collection point scenario version to it using the nodes potential collection points data from the request body.

DELETE /api/protected/project/:projectname/collectionPointScenario/:title

No request body necessary.

Deletes the collection point scenario with the title in the URI from the project with the projectname in the URI.

PUT /api/protected/project/:projectname/collectionPointScenario

Request body: { title: string; nodesPotCP: { nodeid: number; potentialCollectionPoint: boolean }[]; newTitle: string; }

Updates the title of the collection point scenario with the new title provided in the request body and adds a new collection point scenario version to it using the nodes potential collection points data from the request body.

POST /api/protected/project/:projectname/vehicleType

Request body: { title: string; averageSpeedTour: number; averageSpeedDepot: number; averageStopTime: number; vehicleCapacity: number; arcsActivated: { sourceNodeID: number; destinationNodeID: number; activated: boolean }[] }

Creates a new vehicle type with the title provided by the request body and then adds a new vehicle type version to it using the activated arcs, average tour speed, average depot speed, average stop time and vehicle capacity data from the request body.

DELETE /api/protected/project/:projectname/vehicleType/:title

No request body necessary.

Deletes the vehicle type with the title in the URI from the project with the projectname in the URI.

PUT /api/protected/project/:projectname/vehicleType

Request body: { title: string; averageSpeedTour: number; averageSpeedDepot: number; averageStopTime: number; vehicleCapacity: number; arcsActivated: { sourceNodeID: number; destinationNodeID: number; activated: boolean }[]; newTitle: string; }

Updates the title of the vehicle type with the new title provided in the request body and adds a new vehicle type version to it using the activated arcs, average tour speed, average depot speed, average stop time and vehicle capacity data from the request body.

POST /api/protected/project/:projectname/result

Request body: { garbageScenarioTitle: string; garbageScenarioTiming: string; collectionPointScenarioTitle: string; collectionPointScenarioTiming: string; vehTypeVersAndWasteDepotNodes: { vehicleTypeTitle: string; vehicleTypeTiming: string; availableWasteDepotNodes: { nodeid: number }[]; }[]; model: string; maxWalkingDistance: number; minWaste: number; }

Adds a result to the project with the projectname provided in the URI using the request bodies data. This initiates the calculation of a concrete result by the waste collection optimization algorithm by writing an input XML file to a folder which then is accessed and read by the waste collection optimization program.

DELETE /api/protected/project/:projectname/result/:resulttiming

No request body necessary.

Deletes the result with the result timing provided in the URI from the project with the projectname provided in the URI.

As soon as the waste collection optimization program has calculated a result, it creates an XML output file which is left in a specified folder. The server then gets notified by a file watcher that an XML output file has been created and loads the files data into the program (and database).

1. Appendix B: Other Technologies

JSON

JSON is a data interchange format. It does not depend on any programming language but uses concepts that are relatable to programmers of most programming languages which makes JSON an ideal for data interchange. It is based on two structures which are unordered sets of key-value pairs (also called objects) and ordered lists of elements. Equivalent structures to unordered sets of key-value pairs exist in most programming languages and are called e.g., dictionaries or hash tables while equivalent structures to ordered lists of elements are called e.g., arrays or lists. Each value of a key-value pair in an object can be an object, an ordered list, a string, a number or one of the values "true", "false" or "null" (*JSON*, n.d.).



Source code 2: JSON example

XML

XML is a data format that allows to represent structured information (*XML Essentials - W3C*, n.d.). XML is, as its name says, a markup language i.e., you can markup content with tags. Tags are recognizable by the opening and closing angle brackets ('<' and '>'). In order to organize data in XML, the data needs to be placed between a beginning and ending tag. Such structures are called elements. A concrete element could be:



Source code 3: XML example with begining and ending tag

Nesting of elements is also possible, and each element can additionally contain attributes which are specified inside a tag:



Source code 4: XML example with attributes and nested elements

Important to note is that a concrete XML-file will have exactly one root element.



Source code 5: Example XML file

The following source code 3 shows the content of a concrete XML-file (including an optional XML declaration on the first line).

1. Appendix C: Code

# Bibliography

*1. What Is PostgreSQL?* (2021, May 13). PostgreSQL Documentation. https://www.postgresql.org/docs/13/intro-whatis.html

*5.9. Schemas*. (2021, May 13). PostgreSQL Documentation. https://www.postgresql.org/docs/13/ddl-schemas.html

*18.2. Creating a Database Cluster*. (2021, May 13). PostgreSQL Documentation. https://www.postgresql.org/docs/12/creating-cluster.html

*About JavaScript—JavaScript | MDN*. (n.d.). Retrieved August 7, 2021, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/About\_JavaScript

Ahn, T., & Grudnitski, G. (1985). Conceptual Perspectives on Key Factors in DSS Development: A Systems Approach. *Journal of Management Information Systems*, *2*(1), 18–32.

Alvarez, J., & Nuthall, P. (2006). Adoption of computer based information systems: The case of dairy farmers in Canterbury, NZ, and Florida, Uruguay. *Computers and Electronics in Agriculture*, *50*(1), 48–60. https://doi.org/10.1016/j.compag.2005.08.013

Apostol, L., & Mihai, F.-C. (2012). Rural waste management: Challenges and issues in Romania. *Present Environment and Sustainable Development*, *6*(2), 105–114.

auth0.com. (n.d.). *JWT.IO - JSON Web Tokens Introduction*. Retrieved August 6, 2021, from http://jwt.io/

Barr, S. H., & Sharda, R. (1997). Effectiveness of decision support systems: Development or reliance effect? *Decision Support Systems*, *21*(2), 133–146. https://doi.org/10.1016/S0167-9236(97)00021-3

*Bcrypt*. (n.d.). Npm. Retrieved August 6, 2021, from https://www.npmjs.com/package/bcrypt

Berners‐Lee, T., Cailliau, R., Groff, J., & Pollermann, B. (1992). World‐Wide Web: The Information Universe. *Internet Research*, *2*(1), 52–58. https://doi.org/10.1108/eb047254

Carlson, B. (n.d.-a). *Connecting*. Retrieved August 5, 2021, from https://node-postgres.com/features/connecting

Carlson, B. (n.d.-b). *Pooling*. Retrieved August 5, 2021, from https://node-postgres.com/features/pooling

Carlson, B. (n.d.-c). *Welcome*. Node-Postgres. Retrieved August 5, 2021, from https://node-postgres.com/

Chandrappa, R., & Das, D. B. (2012). *Solid Waste Management: Principles and Practice*. Springer.

*Chokidar*. (n.d.). Npm. Retrieved August 6, 2021, from https://www.npmjs.com/package/chokidar

*Components Basics | Vue.js*. (n.d.). Retrieved August 6, 2021, from https://v3.vuejs.org/guide/component-basics.html

*Concepts*. (2016, October 27). PostgreSQL Documentation. https://www.postgresql.org/docs/9.1/tutorial-concepts.html

*Cross-Origin Resource Sharing (CORS)—HTTP | MDN*. (n.d.). Retrieved August 6, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

*Express basic routing*. (n.d.). Retrieved August 6, 2021, from https://expressjs.com/en/starter/basic-routing.html

Ferguson, R. L., & Jones, C. H. (1969). A Computer Aided Decision System. *Management Science*, *15*(10), B-550. https://doi.org/10.1287/mnsc.15.10.B550

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

*Getting Started · Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/docs/getting-started

*Getting Started—Vue.js*. (n.d.). Retrieved August 6, 2021, from https://012.vuejs.org/guide/

Igbaria, M., Sprague, R. H., Basnet, C., & Foulds, L. (1996). The impact and benefits of a DSS: The case of FleetManager. *Information & Management*, *31*(4), 215–225. https://doi.org/10.1016/S0378-7206(96)01078-6

*Introduction to Node.js*. (n.d.). Introduction to Node.Js. Retrieved August 9, 2021, from https://nodejs.dev/learn

*Introduction to the DOM - Web APIs | MDN*. (n.d.). Retrieved August 6, 2021, from https://developer.mozilla.org/en-US/docs/Web/API/Document\_Object\_Model/Introduction

*Introduction—Vue.js*. (n.d.). Retrieved August 6, 2021, from https://vuejs.org/v2/guide/

*JavaScript language resources—JavaScript | MDN*. (n.d.). Retrieved August 7, 2021, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language\_Resources

*Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/

*JSON*. (n.d.). Retrieved August 6, 2021, from https://www.json.org/json-en.html

*Jsonwebtoken*. (n.d.). Npm. Retrieved August 9, 2021, from https://www.npmjs.com/package/jsonwebtoken

Keen, P. G. (1980). *Decision support systems and managerial productivity analysis*.

Keen, P. G. W., & Scott Morton, M. S. (1978). *Decision support systems: An organizational perspective*. Addison-Wesley Pub. Co.

*Koa—Next generation web framework for node.js*. (n.d.). Retrieved August 6, 2021, from https://koajs.com/

McLaughlin, B., Pollice, G., & West, D. (2007). *Head first object-oriented analysis and design* (1st ed). O’Reilly.

Mesbah, A. (2009). *Analysis and testing of Ajax-based single-page web applications.* s.n.

*Mock Functions · Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/docs/mock-functions

Nkolika, I. C., & Onianwa, P. C. (2011). Preliminary study of the impact of poor waste management on the physicochemical properties of ground water in some areas of Ibadan. *Research Journal of Environmental Sciences*, *5*(2), 194.

Owusu, G. (2010). Social effects of poor sanitation and waste management on poor urban communities: A neighborhood‐specific study of Sabon Zongo, Accra. *Journal of Urbanism: International Research on Placemaking and Urban Sustainability*, *3*(2), 145–160. https://doi.org/10.1080/17549175.2010.502001

Perry, J. S. (2019, January 17). Logging Node.js applications with Winston and Log4js. *IBM Developer*. https://developer.ibm.com/tutorials/learn-nodejs-winston/

Pick, R. A. (2008). Benefits of Decision Support Systems. In F. Burstein & C. W. Holsapple (Eds.), *Handbook on Decision Support Systems 1: Basic Themes* (pp. 719–730). Springer. https://doi.org/10.1007/978-3-540-48713-5\_32

Pongrácz, E. (2002). *Re-defining the concepts of waste and waste management: Evolving the theory of waste management*. Oulun Yliopisto.

*Rest-apis*. (2021, April 6). https://www.ibm.com/cloud/learn/rest-apis

Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., & Astesiano, E. (n.d.). *Usefulness of Screen Mockups in Use Case Descriptions-A Formal Experiment*.

Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., & Astesiano, E. (2014). Assessing the Effect of Screen Mockups on the Comprehension of Functional Requirements. *ACM Transactions on Software Engineering and Methodology*, *24*(1), 1:1-1:38. https://doi.org/10.1145/2629457

Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., & Astesiano, E. (2010). On the effectiveness of screen mockups in requirements engineering: Results from an internal replication. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–10. https://doi.org/10.1145/1852786.1852809

Rodriguez, A. (2008). Restful web services: The basics. *IBM DeveloperWorks*, *33*, 18.

Rose, D. C., Sutherland, W. J., Parker, C., Lobley, M., Winter, M., Morris, C., Twining, S., Ffoulkes, C., Amano, T., & Dicks, L. V. (2016). Decision support tools for agriculture: Towards effective design and delivery. *Agricultural Systems*, *149*, 165–174. https://doi.org/10.1016/j.agsy.2016.09.009

*Setup and Teardown · Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/docs/setup-teardown

Sharda, R., Barr, S. H., & McDonnell, J. C. (1988). Decision Support System Effectiveness: A Review and an Empirical Test. *Management Science*, *34*(2), 139–159. https://doi.org/10.1287/mnsc.34.2.139

*Single File Components—Vue.js*. (n.d.). Retrieved August 6, 2021, from https://vuejs.org/v2/guide/single-file-components.html

*SQL vs. NoSQL Databases: What’s the Difference?* (2021, June 17). https://www.ibm.com/cloud/blog/sql-vs-nosql

*Standards Archive*. (n.d.). Ecma International. Retrieved August 7, 2021, from https://www.ecma-international.org/publications-and-standards/standards/

Tahir, A., & Ahmad, R. (2010). Requirement Engineering Practices—An Empirical Study. *2010 International Conference on Computational Intelligence and Software Engineering*, 1–5. https://doi.org/10.1109/CISE.2010.5676827

*Template Syntax—Vue.js*. (n.d.). Retrieved August 6, 2021, from https://vuejs.org/v2/guide/syntax.html

*Testing Asynchronous Code · Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/docs/asynchronous

Tolhurst, D. (1995). Hypertext, Hypermedia, Multimedia Defined? *Educational Technology*, *35*(2), 21–26.

*Typed JavaScript at Any Scale.* (n.d.). Retrieved August 7, 2021, from https://www.typescriptlang.org/

Unit testing in Node.js. (2019, January 16). *IBM Developer*. https://developer.ibm.com/tutorials/learn-nodejs-unit-testing-in-nodejs/

ur Rehman, T., Khan, M. N. A., & Riaz, N. (2013). Analysis of requirement engineering processes, tools/techniques and methodologies. *International Journal of Information Technology and Computer Science (IJITCS)*, *5*(3), 40.

*Using Matchers · Jest*. (n.d.). Retrieved August 6, 2021, from https://jestjs.io/docs/using-matchers

Vasanthi, P., Kaliappan, S., & Srinivasaraghavan, R. (2008). Impact of poor solid waste management on ground water. *Environmental Monitoring and Assessment*, *143*(1), 227–238. https://doi.org/10.1007/s10661-007-9971-0

*Vue Router*. (n.d.). Retrieved August 6, 2021, from https://router.vuejs.org/

*What is an Application Programming Interface (API)*. (2020, August 19). IBM. https://www.ibm.com/cloud/learn/api

*What is software testing?* (2019, August 22). https://www.ibm.com/topics/software-testing

*What is Vuex? | Vuex*. (n.d.). Retrieved August 6, 2021, from https://vuex.vuejs.org/

*Why you should be using Vuetify*. (n.d.). Vuetify. Retrieved August 6, 2021, from https://vuetifyjs.com/en/introduction/why-vuetify/

*Winston*. (2021). [JavaScript]. winstonjs. https://github.com/winstonjs/winston (Original work published 2010)

*XML Essentials—W3C*. (n.d.). Retrieved August 6, 2021, from https://www.w3.org/standards/xml/core

Ziraba, A. K., Haregu, T. N., & Mberu, B. (2016). A review and framework for understanding the potential impact of poor solid waste management on health in developing countries. *Archives of Public Health*, *74*(1), 55. https://doi.org/10.1186/s13690-016-0166-4

1. The DOM (Document Object Model) constitutes the representation of data objects which embody the structure and content of a web document (*Introduction to the DOM - Web APIs | MDN*, n.d.). [↑](#footnote-ref-1)
2. Hypermedia was suggested to be defined as "any computer-based system that allows the interactive linking, and hence nonlinear traversal, of information that is presented in multiple forms that include text, still or animated graphics, movie segments, sounds, and music" (Tolhurst, 1995, p. 25). Hypertext, contrary to hypermedia, refers only to static information which can include e.g., text, pictures and tables but e.g., not audio or video (Tolhurst, 1995, p. 25). The World Wide Web which was originally described as hypertext by Berners‐Lee et al. (1992) is an example of hypermedia since it also presents non-static information such as audio and video. [↑](#footnote-ref-2)