UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Daniel Kütt

Delta Building Visualization – Admin Tool

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond Hendrik Tunnel, MSc

Tartu 2019

Delta Building Visualization – Admin Tool

Abstract:

The thesis describes the creation of the Admin Tool for the Delta Building Visualization. During this thesis the existing visualization program was refactored and two additional programs were created: Server program, based on ASP.NET Core, and Admin Tool program, based on Aurelia framework. The data collection and processing was delegated to the Server and the Admin Tool allows using user's custom data in the visualization and also control what is being visualized, remotely over the web.

Keywords:

Visualization, Unity, computer graphics, ASP.NET Core, REST API, Aurelia, web application, client-server architecture, Model-View-Controller, GUI design

CERCS: P170 Computer science, systems

Delta õppehoone visualisatsioon – administratiivne tööriist

Lühikokkuvõte:

Antud töös loodi Delta õppehoone visualisatsioonile administratiivne tööriist. Töö käigus restruktureeriti Delta õppehoone visualisatsiooni programm ning loodi lisaks kaks uut programmi: ASP.NET Core-l põhinev server ja Aurelial põhinev administratiivne tööriist. Serverile delegeeriti andmete kogumine ja töötlus, tööriista kaudu aga võimaldati visualisatsioonis kasutada enda andmeid ning visualiseeritavat kontrollida läbi veebirakenduse.

Võtmesõnad:

Visualisatsioon, Unity, arvutigraafika, ASP.NET Core, REST API, Aurelia, veebirakendus, klient-server arhitektuur, Model-View-Controller, graafilise kasutajaliidese disain

CERCS: P170 Arvutiteadus, süsteemid

Table of Contents

1 Introduction

The Delta building is the new (currently under construction) joint study and research building of the Faculty of Economics, the Institute of Computer Science and the Institute of Mathematics and Statistics of the University of Tartu[1]. In the lobby of the new building there is planned a Video Wall Screen displaying a 3D visualization of the structure, the inhabitants and the weather outside. The visualization also shows the timetable (as text), the current time and temperature outside the campus.

Students A. Voitenko and A. Nikolajev created the first version of the Delta Building Visualization (DBV) as their BSc theses [1] [2]. That project consisted of the Delta building visualization model, which was capable of displaying the current time, date, weather information, visualizing people in the campus building and more. The DBV project is now further improved in the current thesis and by BSc theses of E. Linde (lighting and weather effects) and M. Perli (actor pathfinding and behavior) [3] [4].

The goal of this thesis is to further develop the DBV project so that the visualization could be controlled and configured remotely over an internet connection. The DBV project is refactored into three components for separation of concerns: the Client, the Server and the Admin Tool (see chapter 3), each responsible for one concern.

This separation removes the need to edit scripts or scenes when we want to control or configure the visualization. Otherwise modification would usually require a new build of the application. The separation also allows us to make the DBV Client more easily visualize our custom data. This has three main benefits: firstly, development of the project in the future will be faster due to the ability to simulate different scenarios which saves the developers time. Secondly, hard to predict bugs with real-time data, that comes from several external APIs, can be reproduced more easily. Finally, this tool can be used to make demonstrations of the DBV when promoting University of Tartu and the Institute of Computer Science at different events.

[1] https://eik.ut.ee/en/portfolio/the-new-delta-building/

Chapter 2 of this thesis covers the functional and non-functional requirements based on which the Admin Tool was developed. In Chapter 3 the architecture of the new solution is covered in subchapter 3.1 and subchapters 3.2 - 3.4 cover the implementation of the Client, the Server and the Admin Tool (respectively). Chapter 4 details the testing of the solution and in Chapter 5 conclusions are made and potential future improvements are suggested. In the Appendix both the source code of the project and the compiled build can be found along with a chart detailing the architecture of the project and screenshots of the Admin Tool.

2 Requirements

In the following subchapters the functional and non-functional requirements of the Admin tool are provided. The requirements are listed as user requirements: the goal is to represent what functionality will the system provide to the user [5]. They are based on the problem description from the Introduction and discussions with the thesis supervisor.

2.1 Functional Requirements

The functional requirements focus on what parts of the visualization can be controlled to show the user's desired outcome. Five functional requirements were established for the Admin Tool project.

F1 The student actors in the visualization should be controllable remotely

In the initial implementation, the student actors [2] in the visualization are sent to rooms based on the data from Cumulocity [1] thus the actors are not directly controllable. The Admin Tool should make it possible to send actors to a room and remove them from it as the user wishes using a graphical user interface (GUI).

F2 The weather should be controllable remotely

In contrast to F1, the data on which the weather effects are shown comes from external third party services [1] and thus there is no control over what is shown in the visualization. The Admin Tool should allow the user to change the weather effects (whether it is raining, snowing or a sunny day), the intensity of these effects and the temperature (which is displayed on the GUI above the timetable)

F3 The timetable should be controllable remotely

Educator actors are sent to classrooms based on the timetable from the Study Information System (SIS) [1].In order to have full control over the actors, the timetable needs to be customizable from the Admin Tool. This way the user can change where educator actors are being sent. As a result, the timetable that is displayed on the timetable GUI element can also be changed.

F4 The time of day should be controllable remotely

The time of day (displayed on the timetable GUI element) gets information from the system time and thus the displayed time depends on the machine running the visualization. Custom times should be possible to set from the Admin Tool so that the time of day on the timetable

and the visualization's calculated time of day (lighting outside the building) can be controlled.

F5 The cameras should be controllable remotely

The visualization uses 6 different (virtual) cameras, positioned throughout the building, that move back and forth and through which the visualization loops, showing different parts of the building. It should be possible to select, which camera feed is shown, and the camera position should be tweakable (within the range of its movement)

2.2 Non-Functional Requirements

In addition to the functional requirements listed in the previous chapter, three non-functional requirements were set with the aim of increasing the usability of the solution.

NF1 The Admin Tool should work on the Chrome web browser

The Chromium based Chrome web browser is the most widely used web browser in the world[2] so by ensuring that it is works on Chrome (and by extension on Chromium browsers - Opera, Microsoft Edge), the Admin Tool will be usable by a number of different browsers.

NF2 The Admin Tool should consider limitations of mobile devices

The DBV can be shown in an environment where people do not have access to their personal computers. For example: showing the visualization at a conference, debugging the visualization running in the Delta building etc. In these cases, one can use their mobile phone to use the Admin Tool so it should be possible to use the Admin Tool to its full extent on a mobile device.

NF3 The Client should react to the inputs of the Admin Tool in under 100 milliseconds

100 milliseconds is considered the maximum amount of time that a system can take to respond to the user for the interaction to feel instantaneous [6]. Thus, the Client should react to the inputs coming from the Admin Tool so that the user of the tool feels like they are interacting directly with the Client.

[2] http://gs.statcounter.com/browser-market-share

3 The Implementation

In order to satisfy the functional requirements, the existing project was refactored and two new programs were created, as can be seen on Figure 1.

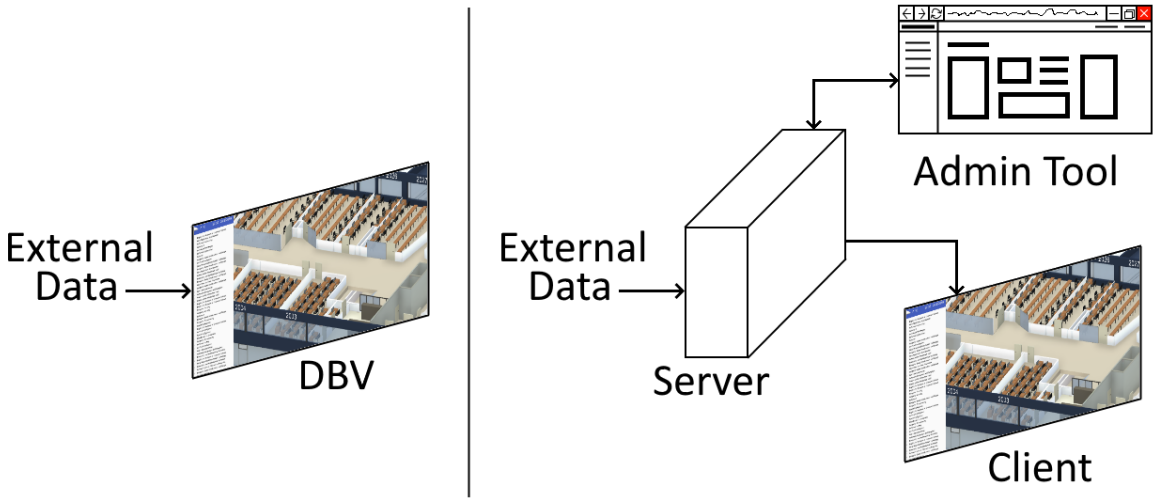


Figure 1. The old (left) and new (right) structure of the project.

Chapter 3.1 gives a general overview of the designed architecture of the solution, including the three new components: the Client, the Server and the Admin Tool. After that chapters 3.2 - 3.4 go into details of each of the new component respectively.

3.1 Architecture

Adding the Admin Tool to the DBV in order to modify the input data of the visualization required refactoring of how the data is received from external sources. To accomplish this the Model-View-Controller (MVC) architectural pattern was followed. MVC pattern is commonly used for decoupling views and models by establishing a communication protocol between them [7] and thus fits as a pattern to use for the project.

Following MVC, the existing DBV project is split into the corresponding three layers as can be seen on Figure 2 (a more detailed version of the chart is located in Appendix I).

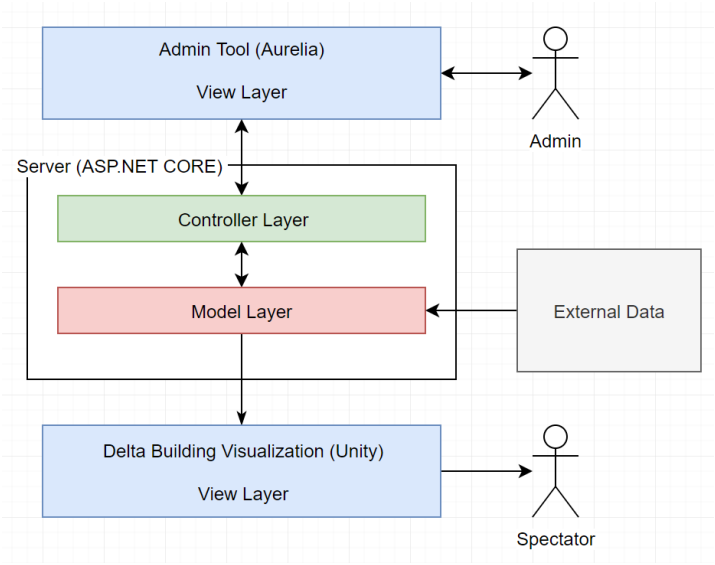


Figure 2. The MVC pattern applied to the DBV project.

The first component/layer of the MVC pattern is the Model, which contains the data that is being visualized. Second layer is the View, which is makes the data visible to users and responds to updates from the Model. The third and final layer is the Controller, which allows modifying the model and thus makes it possible to change, what is shown in the View layer.

First the querying of the external data (i.e. weather, agent counts in a room, the SIS timetable info) is delegated to the Server. The Server oversees collecting the data on behalf of the Client, and it stores it in an internal state. This way the Client is now completely in the View layer, containing only the logic for visualizing the data.

The internal state of the Server consists of multiple smaller states: the Cumulocity state contains information about how many people are in the Delta building rooms (info from the Cumulocity API), weather state contains information about the weather etc. Information for these states comes from a job scheduler, which is responsible for periodically querying the data from the respective APIs (Cumulocity for the sensor data, SIS for the timetable, OpenWeatherMap for the weather data) and relaying the info to the states, which then process the data as needed. This all together makes up the Model layer.

Each of the smaller states can also be modified by their prospective controller class. The controllers have methods that can be called remotely to supply the Model with custom data from the user using the Admin Tool, but first this data must pass validation (it must fit the model of the Model that is being modified). The different methods together make an API and overall form the Controller layer of the application. The Controller and Model layer are both in the Server.

The other half of the View layer is the Admin Tool, which is a web application. The Admin Tool is capable of displaying the state of the application, but also has the capability of communicating with the Controller layer and thus modifying the Model. So, in conclusion, the View layer consists of the Client and the Admin Tool.

3.2 Client

The Client program is the existing visualization application that was made with the Unity game engine. Since the data gathering and initial processing is now a responsibility of the Server, the Client had to be refactored so that it would communicate with the Server to receive the data that it would have queried itself in the previous version. In Chapter 3.2.1 the implementation of the communication protocol is covered and in Chapter 3.2.2 the communication procedure is explained further.

3.2.1 Communication Protocol

Since the Client is no longer doing the data gathering itself, it relies on the Server to provide this information. The communication between the Client and the Server is done using asynchronous TCP sockets as they are very light-weight, they work well with the nature of the communication and they allow the communication format to be defined by the author [8].

The chosen communication format was created by the author and it is based on JSON[3] due to two reasons. Firstly, the JSON format is easily readable, so it is easy to debug, should there be problems with the data. And, secondly, the APIs that are being queried support returning data in the JSON format and this has already been used in the previous version of the DBV project meaning that we do not have to refactor the data processing on the Client side (the logic that determines how the visualization reacts to the data).

Components that previously handled the data gathering are now dependent on the TCP Listener and had to be refactored to obtain the data through the observer pattern, as illustrated by Figure 3.

[3] https://en.wikipedia.org/wiki/JSON

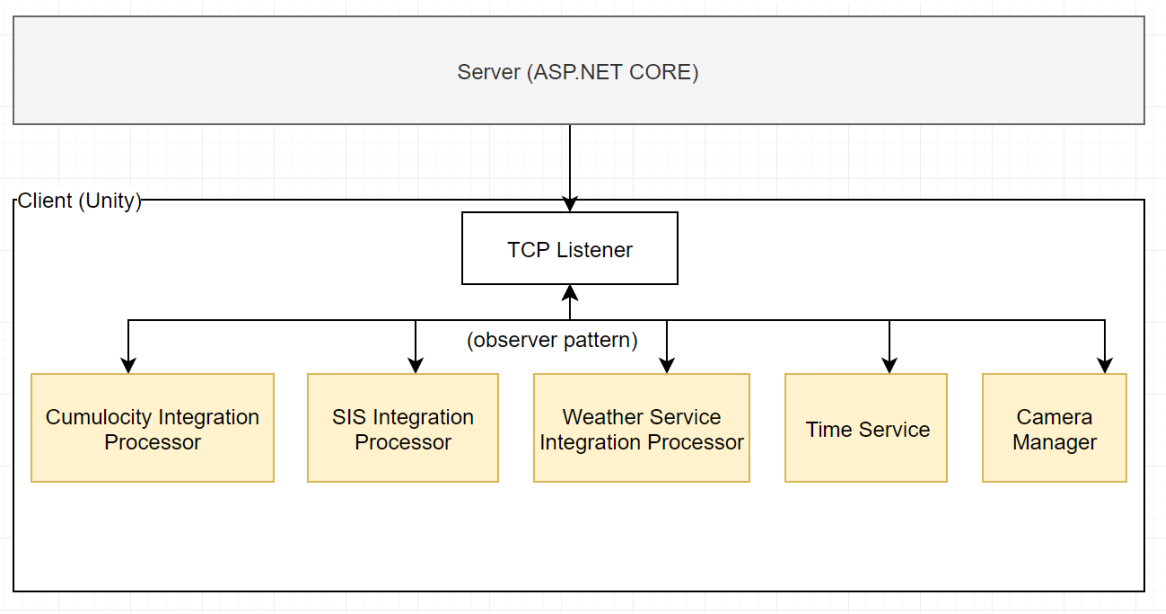


Figure 3. Components of the observer pattern: subject (white) and observers (yellow).

The observer pattern was chosen because it increases code reusability by not tightly coupling components [7]. As a result, the observer pattern allows us to quickly add more components that depend on external data in the future without having to modify existing components.

The observer pattern is implemented by defining interfaces for the Subject and the Observer, in which observers implementing the Observer interface can subscribe to the Subject (the TCP Listener on Figure 3) so when the Subject receives an update, it notifies all of its Observer listeners. In our project the pattern was adapted and modified by adding a clause to the subscription process: the observers can tell the subject, what data are they exactly interested in. For example, if the TCP Listener receives the JSON data below, it would notify all observers that were subscribed to “camera” data with the following:

```
1. {
2.   "name": "camera",
3.   "manual": true,
4.   "state": {
5.     "activeCamera": 2,
6.     "position": 50
7.   }
8.}
```

This is because of line 2 in the JSON data which specifies that the state data contains information about the “camera” state, the information that gets passed to the observers is on lines 4 - 7 (the state JSON object).

3.2.2 Communication Process

When the Unity application is started, in the starting phase of the application the TCP Listener is started, which then tries to connect to the location which is specified in its config file. Using a config file allows us to change the host that we are trying to connect to without having to recompile the visualization client. The config file functionality is provided by the SharpConfig plugin[4].

If the connection fails, the Client closes the socket and releases the allocated resources, waits a few seconds and tries connecting again. This way the application does not require a restart, if the Server is not running at the time of the application launch and it also makes the application able to reestablish its connection with the Server should it be disrupted due to network issues.

After the initial connection is made with the Server, the Server returns its internal visualization state, which is used to quickly get the visualization working. Responsible for this is the state observer, which sole purpose is to process the initial state response by notifying all subscribed observers with the initial state of the visualization.

When the initial connection process is completed, the TCP Listener stays in listening mode, waiting for new data from the Server and when it arrives, the specified observers are notified and the listening process starts again. This is repeated till the application is shut down. The whole process is depicted on Figure 4.

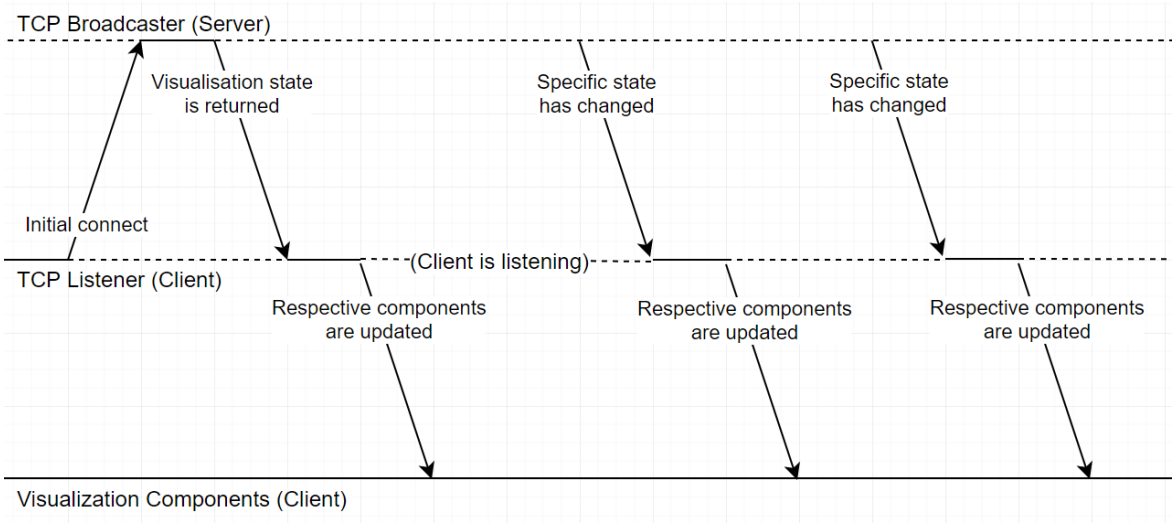


Figure 4. The communication process.

[4] https://github.com/cemdervis/SharpConfig

Currently the communication process is only one way i.e. the client does not send the server information about its different processes. For example, the server is not aware of the actor's exact locations, only that there should be *n* number of actors in a specific room. The communication process from the Server's side is covered in the next chapter.

3.3 Server

The Server is the first of the two new applications that were created as a part of this thesis. It is built on top of Microsoft's ASP.NET Core framework, a free and open-source web framework[5].

This framework was chosen because it is based on the .NET Framework, which is what the Unity engine also uses for its scripting language. Working on the same framework allows us to use the same constructs and potentially even reuse the business logic code. Another reason for choosing ASP.NET Core is it being cross-platform (due to it also running on the .NET Core framework), which gives us more flexibility in the future, when we want to host the server on a remote (virtual) machine. Finally, the author did not have any experience with the framework before and wanted to use this as a learning opportunity.

In Chapter 3.3.1 we go into more detail about the Model layer of the application, covering the design and implementation of the internal states of the Server. Chapter 3.3.2 describes the creation of the REST API through which the Admin Tool can communicate with the Server.

3.3.1 Internal states

The internal states are essentially the model layer of the application meaning that they contain the core information that the application is running on [7]. This is achieved by gathering the data from external sources (Cumulocity sensor data, OpenWeatherMap API's weather data etc.), processing it and then notifying the connected clients. In addition to this, the data is also stored internally in memory for future use.

The first step of implementing the model layer was to determine the actual model of the data. To do this we analyzed the visualization to see, what is being used by the visualization and vice versa. After this, corresponding data structures were created to hold this

[5] https://en.wikipedia.org/wiki/ASP.NET_Core

information in an easily accessible way. They are referred to as the internal states of the application, because they contain the data which the Client is visualizing.

An example of this is the building state which contains the following information about the rooms of the Delta building: room name, as indicated by the room number (1004, 1037 etc.), room size, which tells how many seats for student actors are in this room (35 for room 1004, 256 for room 1037) and the human count, which indicates how many student actors are in the aforementioned rooms (comes from sensor data).

In total 5 internal states were created to satisfy functional requirements F1 - F5:

- the building state – from the example above (F1),
- the weather state – what weather effects are shown in the visualization (F2),
- the SIS state – what timetable is shown and where are the educator actors (F3),
- the time state – what time of day and what timescale is the visualization using (F4),
- the camera state – what camera feed is the visualization using and in which position is the camera (F5).

The second step of implementing the model layer is to ensure that the model is up to date with the latest information and state managers were created for this purpose. The state managers contain within themselves 2 states: automatic and manual. The automatic state is the state containing real-time data from the external data sources, the manual state however has a combination of the real-time data and data set by the user of the Admin Tool. Having two separate states allows us to keep one state up to date with current information even if the visualization is in the manual state - the information does not get lost and is ready to be used.

The time and the camera state do not require any additional work besides wrapping and initializing them – the time state gets its time from the system clock and the camera state is initially set to automatic. The building, weather and SIS state, however, depend on external data and this has to be constantly queried in the background.

The building state is kept up to date using the same long polling method as described in Voitenko's thesis [1] with slight alterations, but for the weather and timetable state a job scheduler was used. Since ASP.NET Core does not have support for regular background tasks built in, Quartz.NET v3 was used. Quartz.NET is a .NET port of a popular open source

Java job scheduling framework Quartz[6] and the version 3 introduces asynchronous task-based jobs[7]. Two jobs were created to regularly query the SIS and OpenWeatherMap APIs and update the respective states. Whenever any of the states get an update, the Client is notified of the changed state. This can be seen on Figure 5:

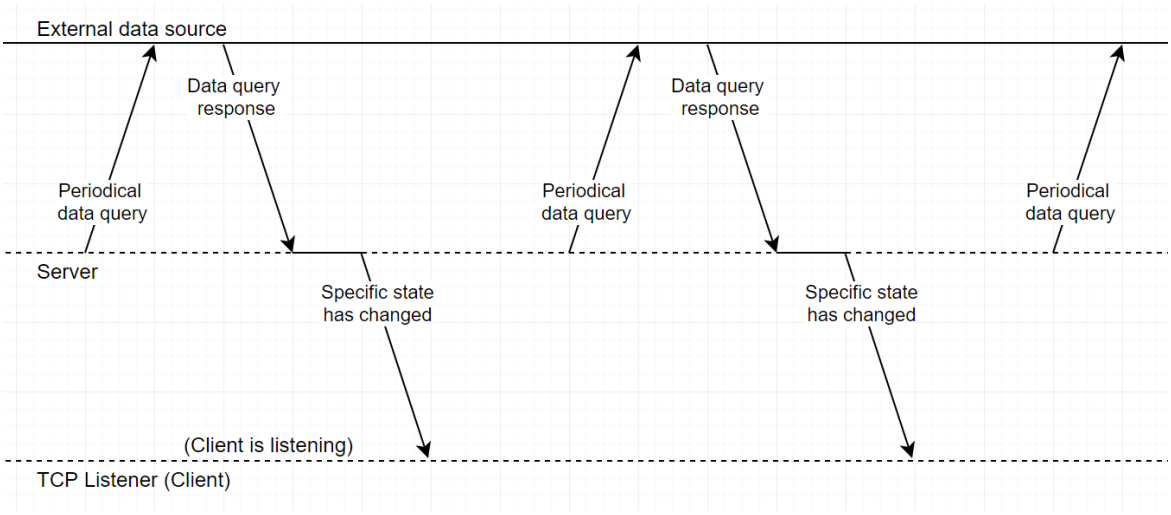


Figure 5. The state updating process.

While the states use internally a combination of primitives, strings, lists and dictionaries to hold the data, they all also expose an interface through which one can receive or overwrite the state with JSON data. The JSON functionality to ASP.NET Core is added by JSON.NET from Newtonsoft, a popular JSON framework for .NET[8]. Since the states can be converted into JSON and loaded from JSON, we can copy the automatic state over to the manual state when we do the corresponding switch. This is useful for when the user wants to switch the visualization to manual mode and change only a few things, as nothing will change in the visualization with just the switch.

By operating with JSON, we can use the same interface to send data in JSON format to both the Client and the Admin Tool. The Server-side setup for communicating with the Admin Tool is covered in the next chapter.

[6] https://www.quartz-scheduler.net/

[7] https://www.quartz-scheduler.net/documentation/quartz-3.x/migration-guide.html

[8] https://www.newtonsoft.com/json

3.3.2 REST API

The REST API exposed by the Server is a collection of interfaces which can be called using HTTP requests. The support for these APIs comes out of the box with ASP.NET Core. The callable methods are attached to controller classes that first validate the inputs. When the validation passes, they modify the model as the input and business logic dictate. These controller classes, as their name may hint, form the Controller layer.

The Controller layer in the DBV project consists of 5 separate controller classes, each one responsible for handling the communication and modification with one of the states. This is depicted on Figure 6 below:

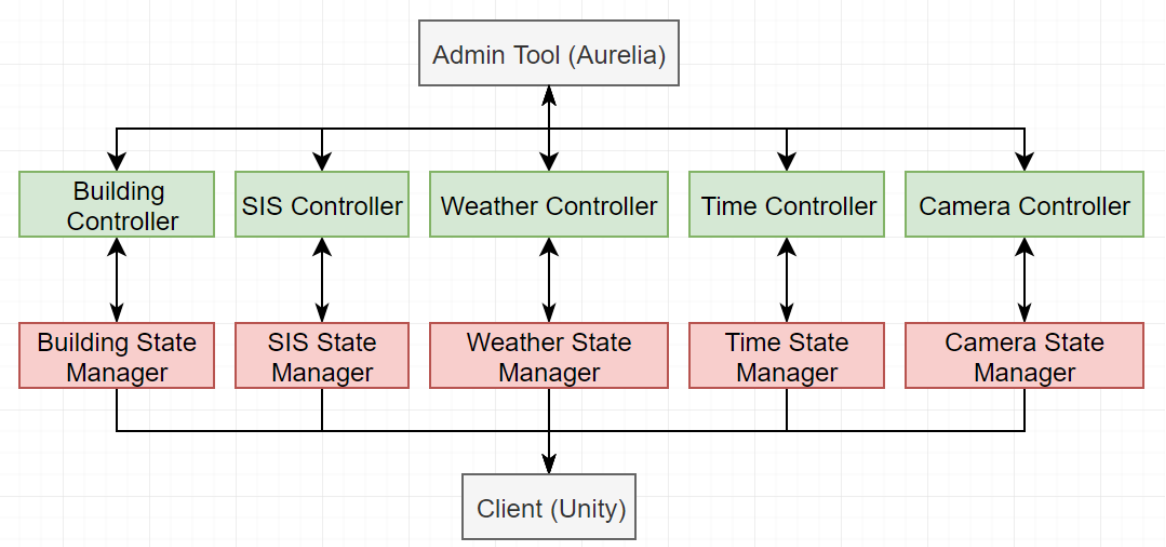


Figure 6. The relationship of controllers (green) and the state managers (red).

Using the available methods, we can modify the state of the application and control, what is being visualized by the client. It is done by making the API calls with a tool like Postman[9], which would require us to know the API endpoints and the requested format of the data, or through a client like the Admin Tool that has been interfaced for this exact purpose.

3.4 Admin Tool

The Admin Tool is the second of the two new applications that were created as part of this thesis. It is built in the JavaScript framework Aurelia, which is a collection of JavaScript modules with support for dependency injection, templating, binding, routing and more[10].

[9] https://www.getpostman.com/

[10] https://aurelia.io/

The Aurelia framework was chosen mainly due to the author's previous experience with it, but also because Aurelia has support for all of the functionalities that a modern web application needs (i.e. routing, templating, dependency injection etc.). In Chapters 3.4.1 and 3.4.2 we respectively cover the implementation of the design and functionality of the Admin Tool.

3.4.1 Design

According to NF2, the application should be easy to use on a mobile device, the goal for design was set to build a responsive web application that would work on many different screen sizes.

To achieve this, the author chose the Bootstrap library to build and style the Admin Tool with. Bootstrap is a toolkit for building responsive and mobile-first experiences[11] and it is one of the most popular frameworks for doing so[12]. In addition to this, the author also decided to base the design off of an open source bootstrap starter template[13], to create the base for a dashboard-style responsive mobile-first web application which can be seen on Figure 7.

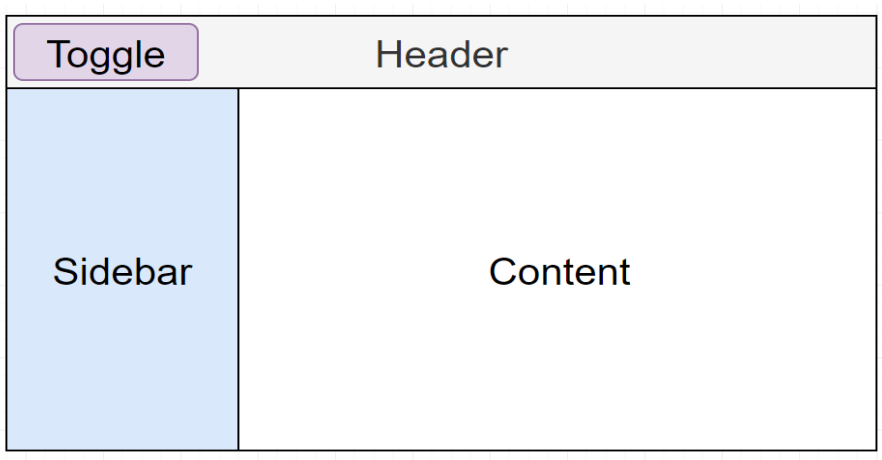


Figure 7. Layout of the Admin Tool: Header (gray), Menu (blue), Content (white) and a button to toggle the Menu (violet).

[11] https://getbootstrap.com/

[12] https://www.ostraining.com/blog/webdesign/bootstrap-popular/

[13] https://startbootstrap.com/templates/simple-sidebar/

A layout with a collapsible sidebar was chosen so that the user can maximize the area of the Content by hiding away the Menu using the Toggle Button. This is aimed toward screens with limited vertical space.

The Header is the top navbar which currently holds only the toggle button, on mobile this element is always visible in the top of the screen, so that users have easier access to the Sidebar. In the future the Header can be extended to have links to generic items such as Admin Tool settings or session management. Currently the Sidebar contains the navigation links that the user can use to navigate to different functionality tabs.

The Content section is the main section of each view - it is the part of the screen where the user interacts with the Client through the REST API of the Server. In the top of the Content section there is an information box notifying the user whether that specific module is using manual data from the user or real-time data from the respective data source. This is shown on Figure 8:

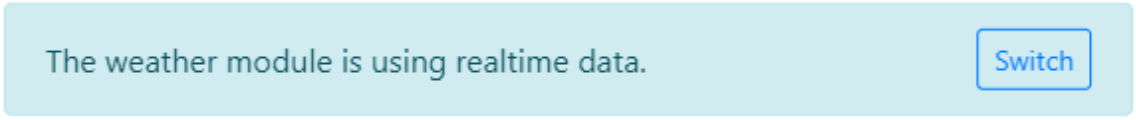


Figure 8. The weather module shown in the weather tab is in the automatic state.

By using the switch button shown on the figure, the user can quickly switch between the manual and automatic state.

3.4.2 Functionality

To communicate with the REST API exposed by the Server, five tabs were created: weather, building, camera, time and timetable tab. Each of the tabs communicates with one of the controllers as depicted on Figure 10:

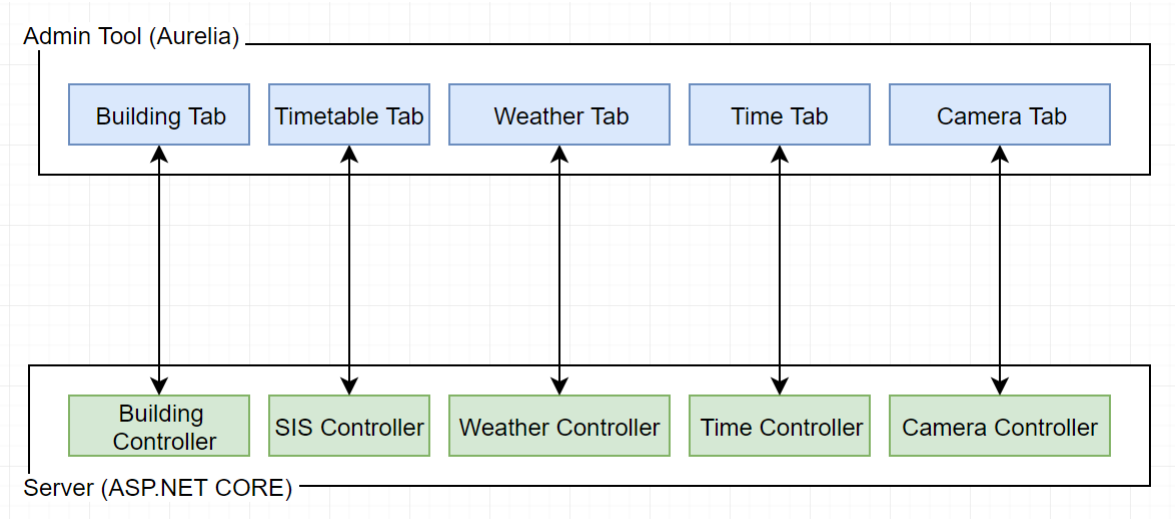


Figure 10. Admin Tool and Server communication.

Each of the tabs have different web form elements[14] to manipulate the corresponding state. For example, the Building tab has input fields, which let you specify the count of actors in a room. The input fields have labels which identify the room and there is also an element attached to the input field which shows the current and max actor count of a room. An example of this is shown on Figure 9:

Figure 9. Input fields of the building tab that enable modifying of the building state.

The user can write the room actor count in the field and by either pressing [Enter] or clicking the submit button, both of which submit the web form. When the form is submitted, the values get sent to the building controller (the Server) through a HTTP PUT[15] request, which validates the data. If the request is valid, the controller modifies the building state and returns the new state back through the controller to the Admin Tool, which then updates its display.

[14] https://www.w3schools.com/html/html_form_elements.asp

[15] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_message

Since the building state was modified, the state manager also notifies all connected Client applications of this and the visualization state changes (actors move around to represent the new state).

In the content portion of the screen there is also the information box mentioned in the previous chapter and displayed on Figure 8Figure 8. The weather module shown in the weather tab is in the automatic state.. Due to the whole tabs being too big to display in this chapter, they can be found in Appendix II.

In addition to the five tabs, a sixth tab was created for scenarios. These scenarios are a predefined set of instructions for the Server, that can be used from the Admin Tool to test the project and to showcase it. The scenarios try to emulate possible situations the visualization may have to work in while also demonstrating some of the edge cases (i.e. the whole visualization is filled to its' capacity).

4 Testing and Results

To ensure that the result of the thesis is of quality, the Admin Tool project was tested. The technique selected for the testing was functional testing[16] where functional and non-functional requirements listed in Chapter 2 were tested and checked, if they were satisfied. In addition to manually testing the solution, the author also shared development versions of this tool with Perli and Linde, who could use it while testing their own solutions [3] [4].

The testing was performed throughout development and with the final version of the Admin Tool, testing environment was Google Chrome version 74 on Windows 10 with the Server, Admin Tool and Client applications all running in the same machine. In addition to this, the Admin Tool was also tested with the author's personal smartphone Mi 8[17] using the same Wi-Fi network as the Server program.

Firstly, the RAM usage of the Server application was measured when the application states were all on automatic mode, no significant increase in its usage was detected. This is most likely due to the garbage collection[18] and on demand dependency injection that the Server application inherits from ASP.NET Core.

Secondly the functional requirements 1 - 5 were tested. While all of the requirements were met, the author noted two improvements which would improve the usage of the Admin Tool and help with further testing in the future. These improvements are covered in chapter 4.1.

Some of the testing also took place on the mobile phone (also in the Chrome browser). The author did not notice any features that were unusable because of mobile phone limitations, therefore concluding that NF2 is met.

Finally, the speed of the project was measured, measuring the time it took the Client to receive the information sent by the Admin Tool. All the other components besides the building tab/state took on average under 15 milliseconds for the information to reach the Client, with the initial requests taking longer due to resource allocation. Most of the time was spent on the processing of the HTTP request, as the time it took for the Server to notify the Client was constantly around 5 milliseconds. If we assume that the DBV project gets a similar configuration as the SIS so that the Admin Tool's HTTP request takes the same time

[16] https://en.wikipedia.org/wiki/Functional_testing

[17] https://www.gsmarena.com/xiaomi_mi_8-9065.php

[18] https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

to reach the endpoint as it does for the SIS, the average time added to the response would be 45 milliseconds, which added up totals at 60 milliseconds. This is less than 100 milliseconds and therefore satisfies the non-functional requirement NF3.

An exception to this is the building tab / state change, as this took on average 157 milliseconds to reach the Client due to the processing of the state change input in the Server that came from the Admin Tool taking longer. However, the author noted that this was not noticeable in the visualization due to the nature of the actors' movement and predicts that this will not be an issue even with the added latency.

4.1 Improvements

The testers (the author, E. Linde and M. Perli) noted that it was tedious to have to manually enter values for all fields, as a result templates were added to the tabs. These templates allow the user to set the corresponding state to the described values by clicking just one button. An example of this can be seen on Figure 11, where there are 3 weather templates.

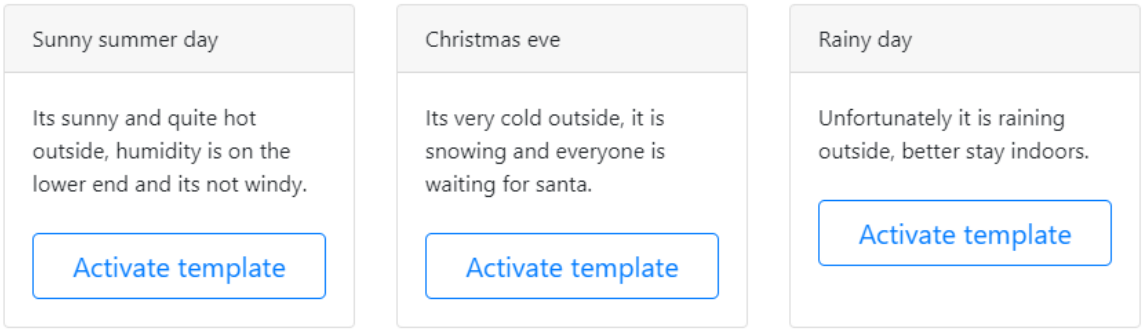


Figure 11. Weather templates for a sunny, a snowy and a rainy day.

By activating one of the templates, the user can quickly setup the visualization to show rain or snow weather effects.

In addition to the templates, buttons were added to randomly give values to the fields, where it would make sense. As a result, in the building tab of the Admin Tool, it was now possible fill out either a room, an entire floor or the entire building randomly (or empty it).

Another thing the testers noted was that while some of the changes took place fast, there were effects in the visualization which would happen over a long time (i.e. 30 minutes). To help showcase them a way to edit the timescale of the visualization was added, to the time tab of the Admin Tool.

5 Conclusion

As the result of this thesis the existing Delta Building Visualization solution (the Client) was refactored and two new programs (the Server and the Admin Tool) were created. The Client is now only responsible for visualizing data while the Server oversees gathering and processing it. Through the Admin Tool it is now possible to remotely control, what is being visualized by the Client.

Firstly, the requirements for the new solution were defined. The Client - Server - Admin Tool architecture was designed to satisfy the requirements. The Server program was created in ASP.NET Core and the Admin Tool in the Aurelia Javascript framework. The solution's functionalities were tested by the author and several improvements were made as a result of testing.

In addition to the improvements made by the author, during the development some ideas that were too big for the current scope of the thesis came up. One idea was that the visualization could support more sophisticated actor styles, for example different clothing items, accessories and actor colors. If this idea were to be combined with a more sophisticated integration with the SIS, the students of the institute could personalize the actors that represent them. This could turn the DBV project into a memorable experience of future students' university times.

The author would like to thank the friends, the colleagues, the boss and the family, all of whom provided (mostly) moral support during this project. The author is grateful for the cooperation of students Perli and Linde with whom working together was a nice experience. The author's gratitude goes to his supervisor Raimond-Hendrik Tunnel, who was helpful in all aspects of the thesis creation, who also provided moral support and life advice and who was interesting to argue with. And in the end, the author is very excited to see this project eventually being used in a live environment.

References

[1] Voitenko A. Delta õppehoone keskkonna visualiseerimine. University of Tartu, bachelor's thesis, 2018

[2] Nikolajev A. Delta õppehoone visualiseerimine ja optimeerimine. University of Tartu, bachelor's thesis, 2018

[3] Perli M. Agent Logic for Delta Building Visualization. University of Tartu, bachelor's thesis, 2019

[4] Linde E. Delta õppehoone visualisatsioon - visuaalsed efektid. University of Tartu, bachelor's thesis, 2019

[5] Sommerville I. Software Engineering (10th Edition), Pearson, 2016.

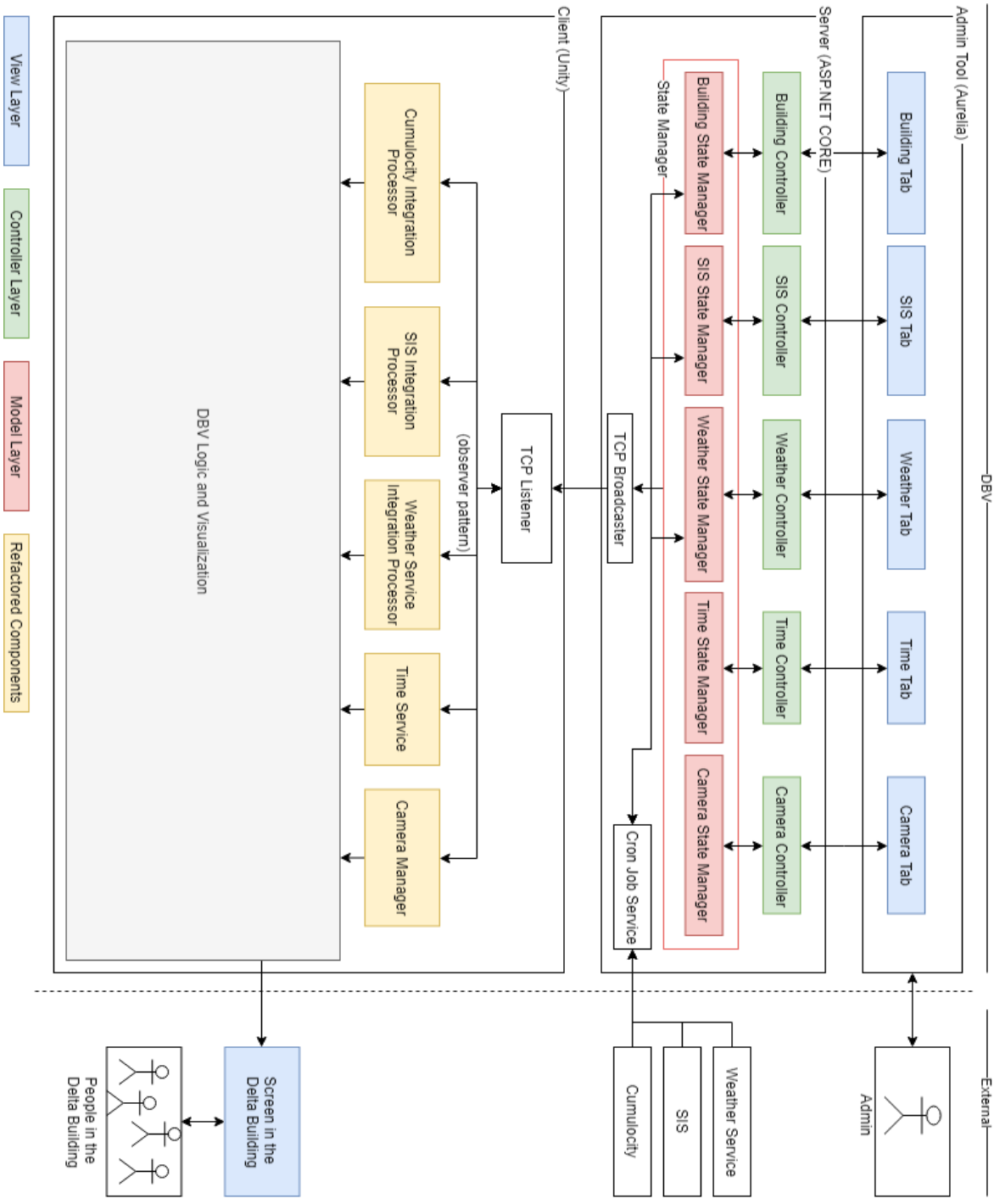
[6] Nielsen J. Usability Engineering, Academic Press, 1993.

[7] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley https://github.com/dieforfree/edsebooks/blob/master/ebooks/Design%20Patterns%2C%20Elements%20of%20Reusable%20Object-Oriented%20Software.pdf (24.03.2019)

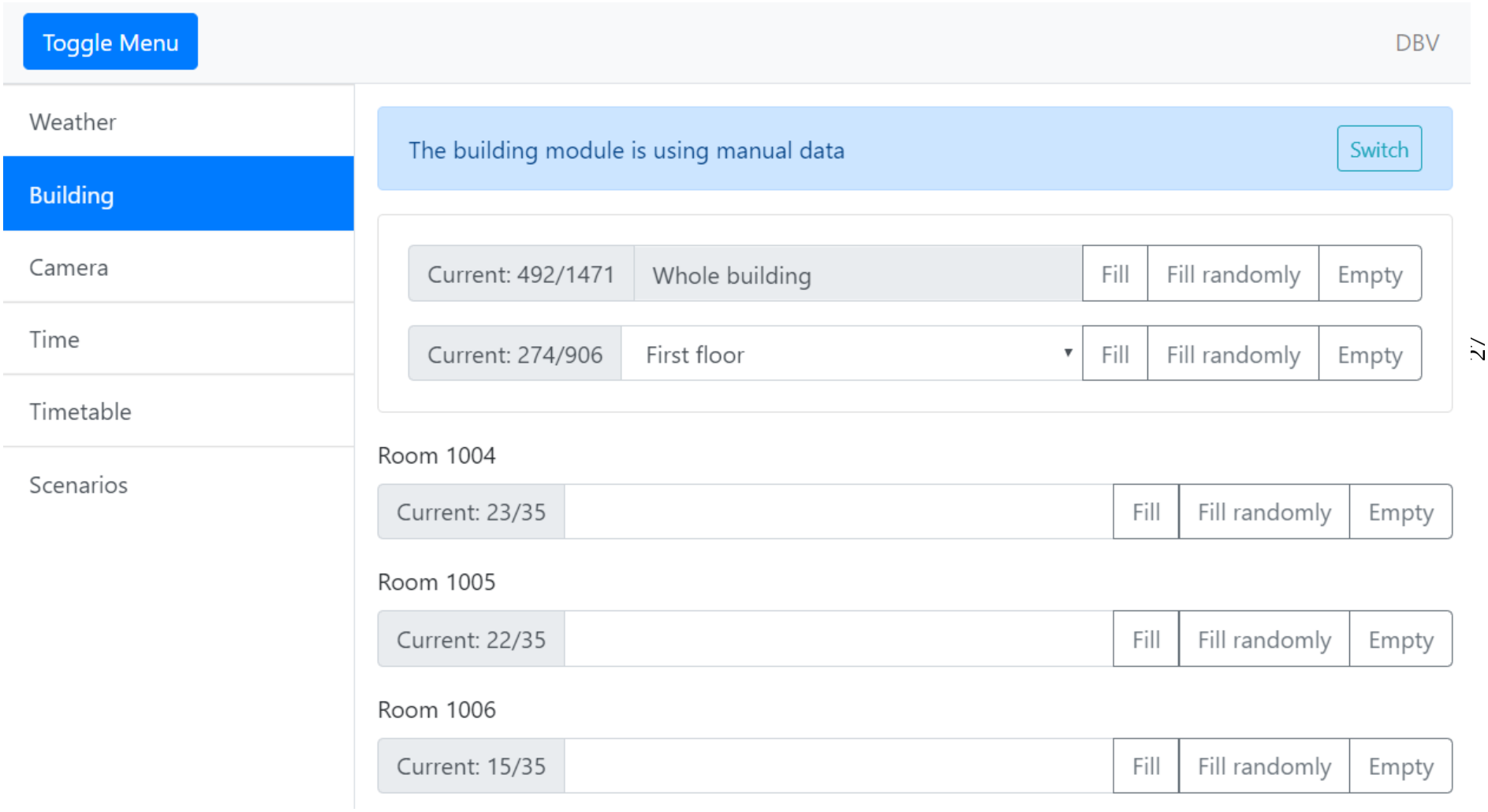
[8] Hall B. Beej's Guide to Network Programming, 2016 http://beej.us/guide/bgnet/pdf/bgnet_A4.pdf (08.05.2019)

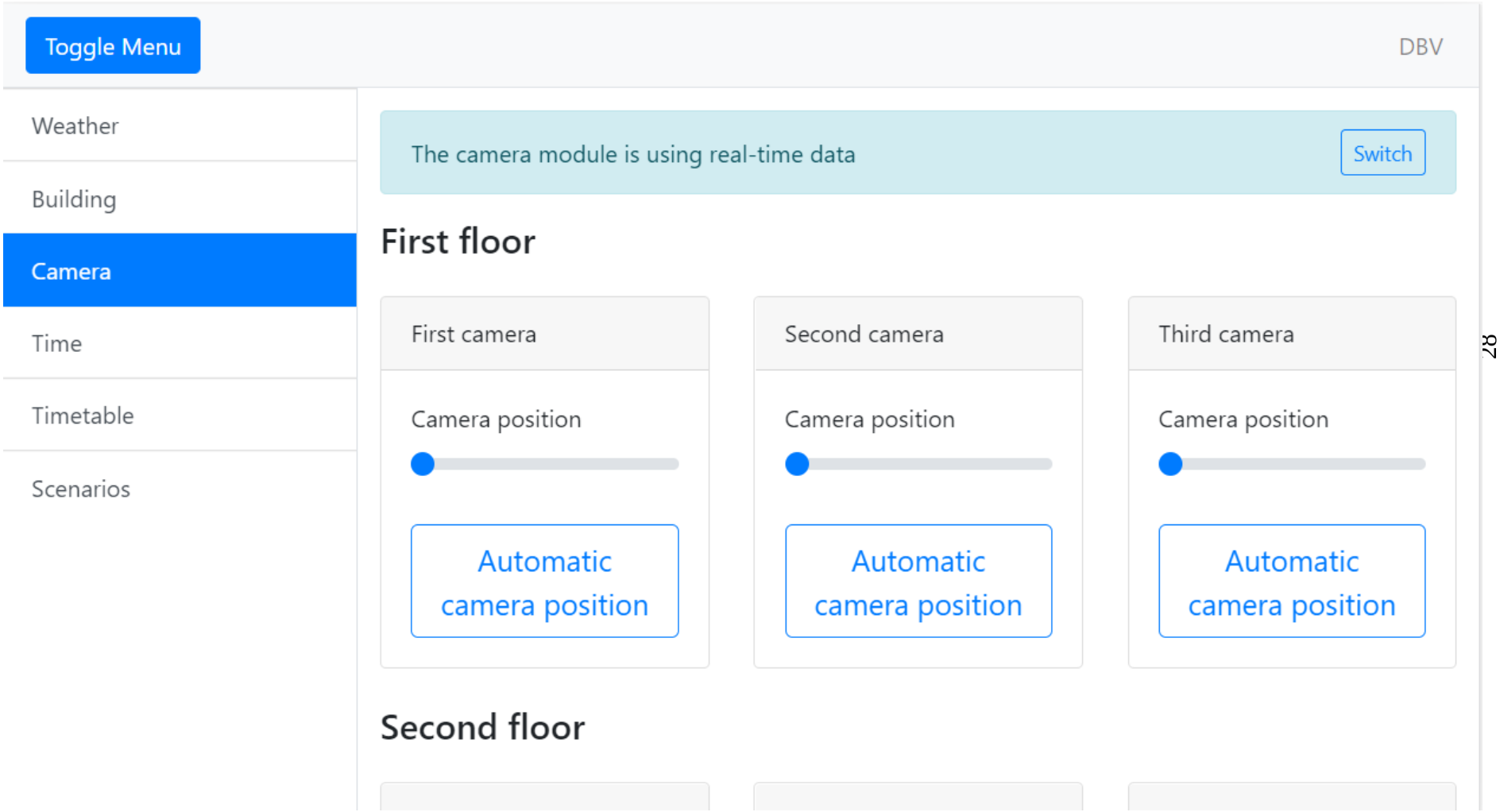
Appendix

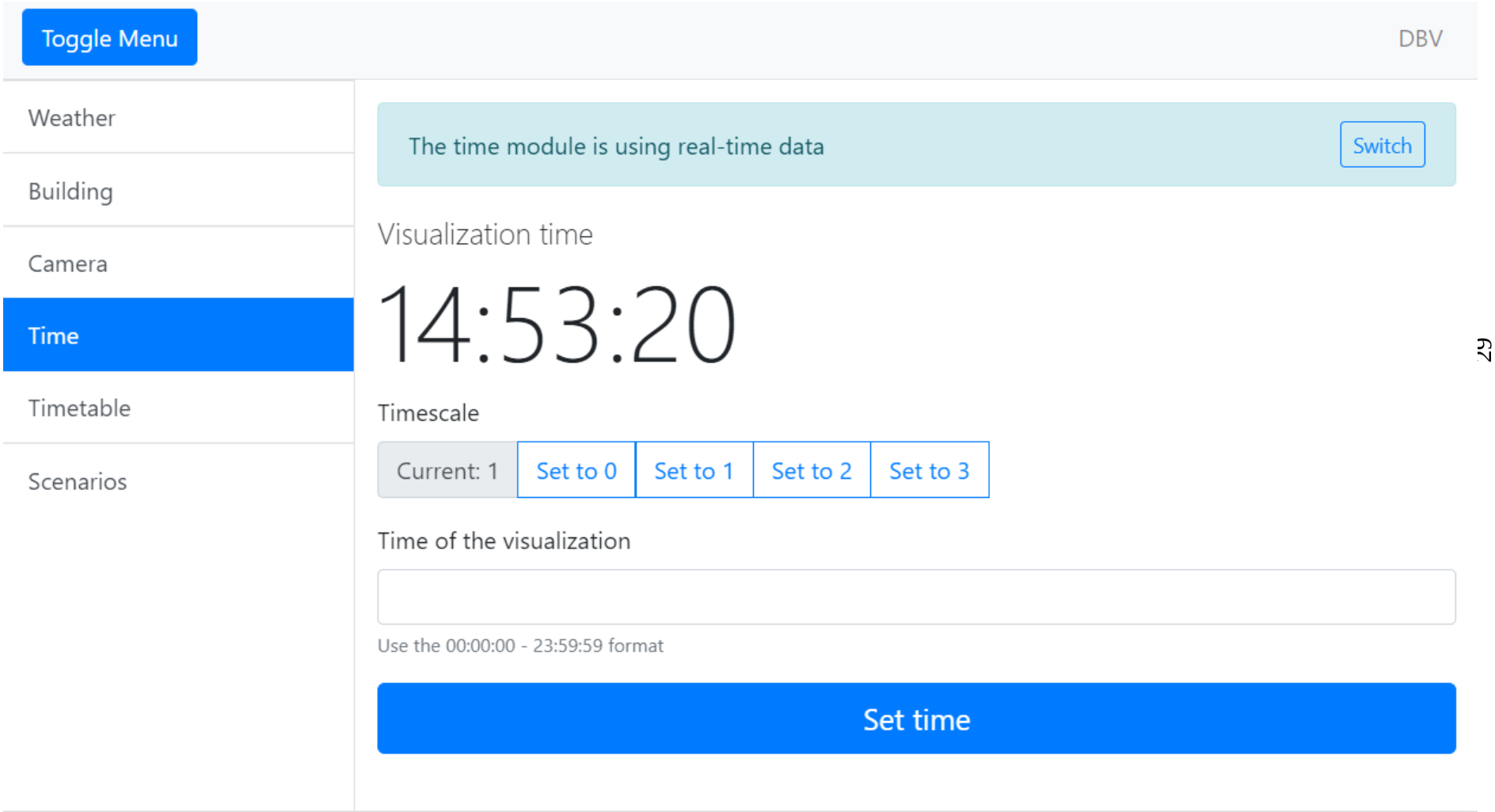
I. DBV Project Architecture Chart

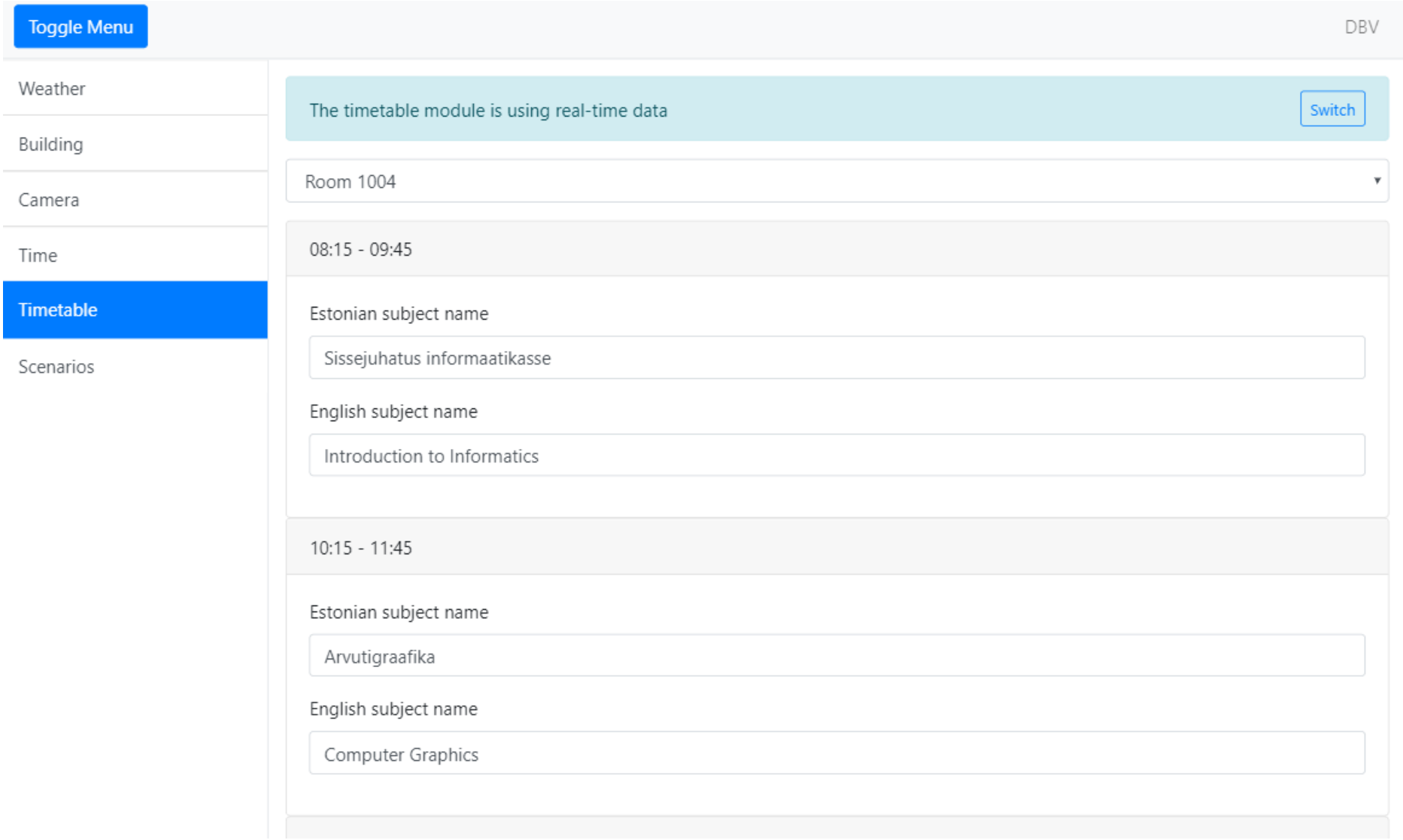


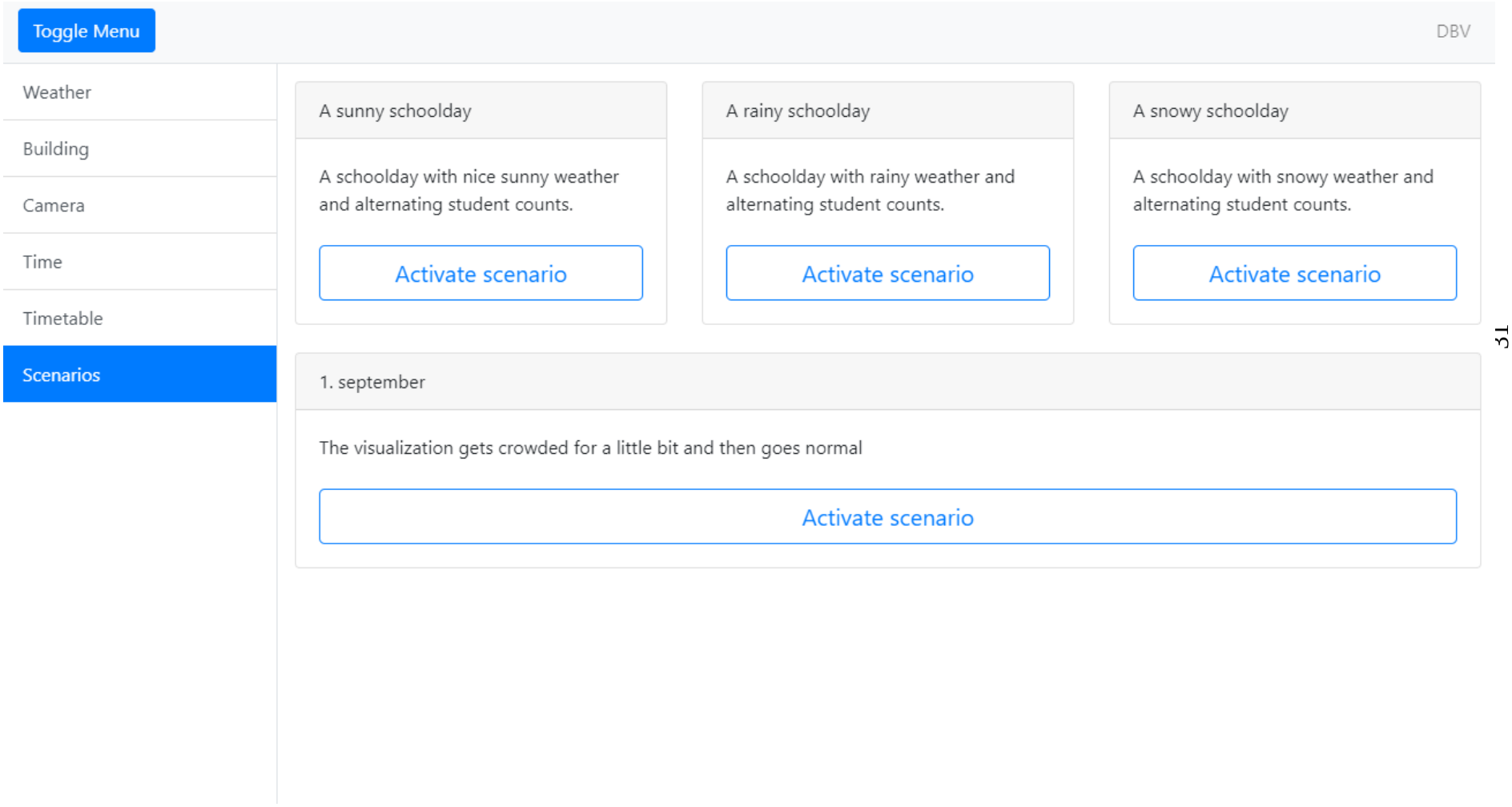
II. Admin Tool Tabs


Toggle Menu
DBV
Weather
Building
Camera
Time
Timetable
Scenarios
The building module is using manual data
Switch
Current: 492/1471
Whole building
Fill
Fill randomly
Empty
Current: 274/906
First floor
Fill
Fill randomly
Empty
Room 1004
Current: 23/35
Fill
Fill randomly
Empty
Room 1005
Current: 22/35
Fill
Fill randomly
Empty
Room 1006
Current: 15/35
Fill
Fill randomly
Empty


Toggle Menu
DBV
Weather
Building
Camera
Time
Timetable
Scenarios
The camera module is using real-time data
Switch
First floor
First camera
Camera position
Automatic camera position
Second camera
Camera position
Automatic camera position
Third camera
Camera position
Automatic camera position
Second floor


Toggle Menu
DBV
Weather
Building
Camera
Time
Timetable
Scenarios
The time module is using real-time data
Switch
Visualization time
14:53:20
Timescale
Current: 1
Set to 0
Set to 1
Set to 2
Set to 3
Time of the visualization
Use the 00:00:00 - 23:59:59 format
Set time


Toggle Menu
DBV
Weather
Building
Camera
Time
Timetable
Scenarios
The timetable module is using real-time data
Switch
Room 1004
08:15 - 09:45
Estonian subject name
Sissejuhatus informaatikasse
English subject name
Introduction to Informatics
10:15 - 11:45
Estonian subject name
Arvutigraafika
English subject name
Computer Graphics


Toggle Menu
DBV
Weather
Building
Camera
Time
Timetable
Scenarios
A sunny schoolday
A schoolday with nice sunny weather and alternating student counts.
Activate scenario
A rainy schoolday
A schoolday with rainy weather and alternating student counts.
Activate scenario
A snowy schoolday
A schoolday with snowy weather and alternating student counts.
Activate scenario
1. september
The visualization gets crowded for a little bit and then goes normal
Activate scenario

III. Accompanying Files

The latest version of the source code for this project can be accessed form a Gitlab repository[19].

In the accompanying files, a build of the Client (Unity application) and of the Server and Admin Tool (ASP.NET Core + Aurelia application) can be found along with the source code that produced these builds.

[19] https://gitlab.com/UT-CGVR-Projects/DeltaBuildingVisualization

IV. License

Non-exclusive license to reproduce thesis and make thesis public

I, **Daniel Kütt**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
Delta Building Visualization – Admin Tool,

supervised by Raimond-Hendrik Tunnel, MSc.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tartu, **10.05.2019**