Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Øyvind Kanestrøm Sæbø

Client-Side Computing in General-Purpose Digital Shadow Applications

Master's thesis in Engineering and ICT
Supervisor: Associate professor Bjørn Haugen
July 2020

Norwegian University of
Science and Technology

Øyvind Kanestrøm Sæbø

Client-Side Computing in General-Purpose Digital Shadow Applications

Master's thesis in Engineering and ICT
Supervisor: Associate professor Bjørn Haugen
July 2020

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

With the rapid development of IoT technology and having more data available digitally, we have seen an emergence of new digital twin uses-cases. Historically, the definition of the digital twin has emphasized that there be a bidirectional connection between the physical entity and the digital twin, making it possible for the digital twin to both monitor and control the physical entity. However, some of the more recent, less industrial use-cases, like the Digital Twin of the Organization (DTO), as coined by Gartner, seem to focus more on the digital twin's monitoring capabilities, thus adhering more to the concept of a digital shadow.

Aiming to fill a gap in this emerging market, this thesis presents the idea of a general-purpose digital shadow application, an application which removes the need to develop digital shadows from scratch, by letting the user define the behavior and virtual representation of an entity through a frontend code interface. Since digital shadows (as opposed to digital twins), only have a one-way data flow, from the physical entity to the virtual entity, they are in a unique position to run nearly all of their logic on the frontend, rather than the backend, making it feasible to utilize a Backend as a Service (BaaS) in lieu of having to set up custom backend infrastructure.

This thesis specifically aims to show that web technologies have evolved so much in recent years that user-submitted transformation and aggregation logic can be handled in internet browsers, even if the logic is supplied in a domain-specific language not native to the browser and has to be interpreted by an interpreter written in JavaScript.

For testing purposes, a prototype of a general-purpose digital shadow application running all of its logic in the browser was implemented, providing the user with a code interface to write digital shadow logic, which maps API-data into live widgets. To facilitate running untrusted user-submitted code safely in the browser, an interpreter for a simple, domain-specific programming language was created. In addition to serving as a proof of concept, the interpreter was used to reason about how performance is affected by running user-submitted

logic in a domain-specific language not native to the browser, rather than using the browser's JavaScript engine directly.

Experimenting with using the application to create various digital shadows suggests that for basic aggregations and data transformations, it is feasible to calculate the necessary derived values in the user's browser, even when having to parse and evaluate the code in a domain-specific language not native to the browser.

The thesis also concludes that the main concern with moving logic from the backend to the frontend in digital shadows is not so much the decreased performance, as it is the digital shadow limiting itself to being an end destination for data, unable to be utilized by other modules.

Sammendrag

Den raske utviklingen av IoT-teknologi og det faktum at stadig mer data blir tilgjengelig digitalt har ført til et økt antall bruksområder for digitale tvillinger. Historisk har definisjonen av en digital tvilling forlanget at det er toveis datakommunikasjon mellom en fysisk enhet og den digitale tvillingen, slik at den digitale tvillingen kan brukes til både monitorering og styring av den fysiske enheten. Noen av de nye, mindre industrielle bruksområdene for digitale tvillinger, som for eksempel digitale tvillinger av organisasjoner, introdusert av Gartner, fokuserer hovedsakelig på den digitale tvillingens monitoreringsegenskaper. Denne typen system har historisk blitt klassifisert som en digital skygge.

Denne oppgaven tar sikte på å fylle et hull i dette nye markedet, og presenterer ideen om en universell plattform for utvikling av digital skygger som fjerner behovet for å utvikle dem fra bunnen av. Dette oppnår den ved å la brukeren definere atferd og virtuell representasjon av en enhet gjennom et frontend-kodegrensesnitt. Fordi digitale skygger (i motsetning til digitale tvillinger) bare har enveis datakommunikasjon, fra en fysisk enhet til en virtuell enhet, er de i en unik posisjon til å kjøre nesten all sin logikk i applikasjonens frontend, heller enn i applikasjonens backend, noe som gjør det mulig å bruke en generell Backend as a Service istedenfor å måtte sette opp egen backend-infrastruktur.

Denne oppgaven tar spesielt sikte på å vise at web-teknologi har utviklet seg så mye de siste årene at brukerdefinert transformasjons- og aggregeringslogikk kan håndteres i brukerens nettleser, selv om logikken er skrevet i et domenespesifikt språk som i utgangspunktet ikke er støttet av nettleseren, men må tolkes av en fortolker skrevet i JavaScript.

For testformål ble det utviklet en prototyp av en universell plattform for utvikling av digitale skygger, som kjører all sin logikk i nettleseren og gir brukeren et kodegrensesnitt for å definere atferd og visualiseringslogikk, for å mappe API-data til kontinuerlig oppdaterte widgets. For å gjøre det lettere å trygt kjøre potensielt ondsinnet brukerdefinert kode i nettleseren ble det utviklet en fortolker for et enkelt, domenespesifikt programmeringsspråk. I tillegg til å utgjøre en viktig del av prototypens functionalitet ble fortolkeren brukt til å utføre

eksperimenter for å finne ut hvor mye tregere evaluering av et domenespesifikt språk som må tolkes av en fortolker skrevet i JavaScript er i forhold til å evaluere JavaScript i nettleserens JavaScript-motor direkte.

Eksperimentering med å bruke prototypen til å lage digitale skygger rettet mot ulike bruksområder antyder at det er overkommelig å utføre grunnleggende aggregering og datatransformasjon i brukerens nettleser heller enn på en dedikert server, selv om logikken er skrevet i et domenespesifikt språk som i utgangspunktet ikke er støttet av nettleseren, men må tolkes av en fortolker skrevet i JavaScript.

Oppgaven konkluderer med at den viktigste konsekvensen av å flytte logikk fra server til klient i digitale skygger ikke er den noe reduserte ytelsen, men det at den digitale skyggen blir en sluttdestinasjon for data, og ikke kan brukes av andre moduler.

Preface

This master's thesis was written at the Department of Mechanical and Industrial Engineering (MTP) as part of the study programme Engineering and ICT (MTING) at the Norwegian University of Science and Technology (NTNU) in Trondheim.

The project was carried out in the spring of 2020 as a continuation of a specialization project conducted during the autumn of 2019. The specialization project focused on using WebGL to create a React component for creating and visualizing space frame structures in digital twin platforms.

An overarching theme of the specialization project was to experiment with moving tasks which have historically been reserved for native applications and centralized servers, to the web browser. Following the same theme, although not directly building on the results from specialization project, this master's thesis explores the feasibility of safely performing arbitrary user-defined computations in the browser in digital shadow applications.

I would like to thank my supervisor Bjørn Haugen for giving me the freedom to approach this project from an angle I found interesting. I have appreciated his positive attitude and great feedback during our weekly video calls.

Table of contents

Abstract 1

Sammendrag 3

Preface 5

Table of contents 7

Abbreviations 10

1 Introduction 12
1.1 Overview 12
1.2 Background and motivation 12
1.3 Thesis statement 16
1.4 Research questions 17
1.5 Objectives and scope 17
1.6 Project deliverables 18
1.7 Limitations 18
1.8 Structure of the thesis 19

2 Theoretical background 22
2.1 Overview 22
2.2 The digital twin 22
2.2.1 The origin of the digital twin 22
2.2.2 Definition of the digital twin 22
2.2.3 Later definitions of the digital twin 23
2.2.4 Misconceptions 24
2.3 The digital shadow 25
2.3.1 Digital shadow use-cases 26
2.4 Existing digital twin and digital shadow software 29
2.4.1 AWS IoT Device Shadow service 29
2.4.2 Eclipse Ditto 29
2.4.3 Microsoft Azure IoT Hub Device Twins 29
2.5 REST API 30
2.6 WebSocket 30
2.7 Single-page applications 30
2.8 Persistent storage in client-side applications 31
2.9 Domain-specific languages 31
2.10 Running user-submitted code in the browser 32
2.10.1 Cross-site scripting 32
2.10.2 XSS in user-submitted content 32

2.10.3 Deliberately running user-submitted code in the browser 33
2.10.4 Approaches to safely run user-submitted code in the browser 34
2.11 Lisp (programming language) 35

3 Requirements and reasoning 37
3.1 Overview 37
3.2 Functional requirements 37
3.3 Non-functional requirements 40

4 Method 42
4.1 Overview 42
4.2 Deciding to implement a pure client-side prototype 42
4.3 Using the prototype to evaluate the thesis statement 45

5 Implementation 46
5.1 Overview 46
5.2 Making a single-page application 46
5.3 Routing 47
5.4 State-based UI 49
5.4.1 The compose function 50
5.4.2 The If function 51
5.4.3 The Each function 51
5.5 Persistent storage 52
5.6 Fetching API data 53
5.7 Supporting dashboards 54
5.8 Widgets 54
5.8.1 The widget schema 56
5.8.2 Scaling and positioning visualizations 57
5.8.3 Distinguishing between 2D or 3D visualization widgets 60
5.8.4 Choice of coordinate system axis directions 61
5.9 Creating a basic 3D engine in JavaScript 62
5.10 Giving the user freedom through a code interface 65
5.11 Running user submitted code in the browser 66
5.11.1 Creating an interpreter for a domain-specific language 66
5.11.2 Parsing the program to an abstract syntax tree 67
5.11.3 Creating the parser 72
5.11.4 Evaluating the abstract syntax tree 73
5.11.5 Special forms 74
5.11.6 Core library 76
5.11.7 Converting the evaluated syntax tree to JavaScript 76
5.11.8 Making it impossible to escape interpreter scope 78
5.11.9 Preventing denial of service 79

5.12 Challenges with writing widgets code 82
5.13 Helping the user write valid widgets code 83
5.13.1 Optional values with sensible defaults 83
5.13.2 Helpful error messages 83
5.13.3 Displaying the resulting JavaScript structure 85
5.14 Sharing dashboards 86

6 Results 88
6.1 Overview 88
6.2 Walkthrough of the developed prototype 88
6.3 Performance 98

7 Evaluation and discussion 104
7.1 Overview 104
7.2 Assessment of non-functional requirements 104
7.2.1 Availability 104
7.2.2 Extensibility 104
7.2.3 Performance 104
7.2.4 Usability 105
7.2.5 Modifiability 107
7.3 Discussion related to research questions 107

8 Conclusion 112

9 Further work 113
9.1 Overview 113
9.2 Develop a general-purpose digital shadow SaaS solution 113
9.3 Develop a visual programming for user-submitted logic 114

10 Bibliography 115

11 Appendices 120
Appendix A - Dashboard code example from chapter 6 120
Appendix B - Running the prototype application 124
Appendix C - Digital Shadow Language examples 126

Abbreviations

API	Application Programming Interface
BaaS	Backend as a Service
DSL	Domain-Specific Language
DOM	Document Object Model
DTO	Digital Twin of an Organization
HTTP	Hypertext Transfer Protocol
MTING	Engineering and ICT
MTP	Department of Mechanical and Industrial Engineering
NTNU	Norwegian University of Science and Technology
SaaS	Software as a Service
SPA	Single Page Application
REST	Representational State Transfer
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UX	User Experience
XSS	Cross-Site Scripting

1 Introduction

1.1 Overview

This chapter presents the background and motivation for the thesis and the thesis statement to be tested, and raises some additional research questions the thesis should aim to answer.

1.2 Background and motivation

Over the past couple of years, master's theses performed by MTING students at MTP at NTNU have covered various aspects of cloud-based digital twins, with the overarching goal being to remove the need to install heavy enterprise software and remove the need to have access to powerful hardware to perform necessary digital twin calculations (1). Being able to run the digital twin software in the cloud rather than natively would increase the availability the software, since it could be accessed from any device with a browser and an internet connection.

In recent years, a number of such platforms, marketing themselves as Digital Twin as a Service, have started to emerge, along with a large number of IoT platforms which, while not explicitly marketing themselves as digital twin platforms, offer much of the same functionality. Despite the rich availability of performant platforms, a study (2) conducted by Cisco in 2017 revealed that 60% of IoT initiatives stall at the Proof of Concept stage. The initiatives often turned out to be much more difficult than anyone expected and were challenged by long time to completion, limited internal expertise, quality of data and budget overruns. While the quality of data can hardly be blamed on the IoT platforms used, all the other challenges could be attributed to IoT software being unable to provide immediate value, being difficult to use and expensive to set up.

One of the attractive capabilities of the digital twin is to present data in the context of a physical entity. While the established definition of a digital twin also requires that data should flow not only from the physical entity to the virtual entity, but also from the virtual entity to

the physical entity, systems which only deliver the first capability alone are commonly incorrectly referred to as digital twins. As the digital twin concept has evolved from the aeronautic and industrial field, to other fields, the digital twin definition has been diluted. At the same time, these other fields don't see the successful rise in use of digital twins which was predicted by research papers. This might be caused by a gap between the highly complex digital twin platforms available on the market, and common misconception that a digital twin is simply a virtual entity mirroring the behavior of a physical entity.

In many cases, an application which might suit these initiatives better is a digital shadow. A digital shadow makes it possible to combine sensor data from a physical object with our knowledge of how the physical object is constructed, together forming a live documentation of the object, enriched with live data. Knowing how the different parts of the object connect means that we can use the live data to derive other live data, and present the data in a way which can be more intuitively interpreted.

The distinction between the digital twin and the digital shadow has been blurred by inaccurate research, and an overly enthusiastic use of the Digital Twin buzz word. The digital shadow is very similar to a digital twin in that it models a virtual entity based on data from a real-life entity.

A digital twin has a two-way data connection to the physical entity it models, both receiving data and returning some processed data, thus being able to act as a regulator.

A digital shadow, however, only has a one-way data connection, from the physical entity to the virtual entity, its main purpose being to present a virtual model for monitoring purposes.

Since the data consumed by the digital shadow will not affect the physical entity, there is an opportunity to safely move computation to the browser, since the physical entity will not be affected, even if the code in the browser were to be modified with malicious intent.

With the recent improvements in browser performance, we have seen a seen a shift from server-side to client-side web applications, where some or all of the computational load is transferred to the browser (3).

Moving logic to the frontend comes with the following advantages:

Simpler backend architecture
The backend can focus on tasks like creating, reading, updating and deleting data, making it possible to utilize a Backend as a Service (BaaS) to handle tasks common to most web applications.

Distributed computing
Performing computations in the users' browsers means that work is distributed among a higher number of processors.

Fewer HTTP requests
Rather than fetching new, static HTML pages from a server on every user interaction, view changes can be performed with JavaScript.

Quicker access to derived values
Assuming that the values necessary to calculate some derived value are already available in the browser, calculating it directly in the browser is often faster than having to request it from the server, even if the calculations themselves take shorter time to compute on the server.

Rapid development
With more raw data available on the frontend, making changes to the application might just require changes to the frontend code, since it can simply use the data it already has available in new ways. This can reduce the time to value and make it much easier to quickly implement new features.

There are also some considerable disadvantages to moving logic to the frontend:

Less performant computing

In the browser, we don't have easy access to as performant languages as we have on the backend.

Blocking scripts

JavaScript is single-threaded, so while a script is running, the UI will be unresponsive until the script terminates.

The front-end needs much more data

Since the frontend has to fetch the data necessary to perform some calculation rather than just fetching the calculated results, the amount of data which needs to be fetched is higher than if only the result of the calculation was to be fetched from the server.

More data than necessary exposed to the front-end

The backend exposes more data to the frontend than is strictly needed, which might raise some privacy concerns. For instance, a company might be comfortable exposing the average salaries of their employees to the frontend, but they might not be as comfortable exposing a list of each individual salary, just to enable the front-end to calculate the average.

- The performance of the app becomes more dependent on the user's hardware.

- The more logic is moved to the frontend, the higher the risk of the application behaving differently in different browsers due to different browsers adopting new language features at different speeds.

A preliminary thesis conducted for this project in the fall of 2019 suggested that with the rapid development of web browsers' JavaScript engines, many tasks which historically have been more typically found in native applications, such as high quality rendering of 3D models and matrix operations, can now feasibly be performed in frontend web applications.

Previously, such tasks have been limited by the performance limitations of JavaScript being an interpreted, single-threaded language.

This thesis explores the concept of a general-purpose digital shadow application, a platform for creating digital shadows. The hope is that such a platform will be able to lower the threshold for setting up digital shadows, requiring less technical knowledge and providing shorter time to value, at a predictable cost.

In a general-purpose digital shadow application, any physical entity or abstract system should be visualizable as a function of the available data. This means that the application cannot limit itself to a set of predefined visualization templates. The user must have the freedom to calculate any value which can be derived from the available data and transform those to map to any visualization which best suits the user's needs. The application could benefit from giving the user access to a domain-specific language (DSL). This makes the question of moving logic to the browser particularly interesting for general-purpose digital shadow applications, since running user-submitted code in the browser comes with a number of security and performance concerns.

While the measures which need to be taken to run untrusted code on the server might cause some delay, that delay can be far more noticeable when caused by code running in the browser, because of JavaScript's single-threaded, blocking nature. Besides, since the context in which the user-submitted code runs in the browser cannot be as easily isolated, the measures necessary to run untrusted code user-submitted code in the browser may have significant performance implications.

1.3 Thesis statement

Web technologies have evolved so much in recent years that user-submitted transformation and aggregation logic can be handled in internet browsers, even if the logic is supplied in a DSL not native to the browser and has to be interpreted by an interpreter written in JavaScript.

This means that digital shadow SaaS solutions can be developed with minimal backend requirements, while still giving the user the ability to submit arbitrary code to map their data to derived values and visualizations.

1.4 Research questions

In addition to defend the thesis statement, I aim to answer the following research questions:

1. For digital shadows, what logic does it make sense to move to the frontend and what logic should remain on the backend?

2. What are the main digital shadow use-cases that will suffer from the disadvantages of handling more of their logic on the frontend?

3. How much slower is it to run user-submitted visualization code written in a language not native to the browser, in the browser, than it is to run similar JavaScript code using JavaScript's native eval function?

4. Can a general-purpose digital shadow platform work with no backend at all, and what are the limitations of doing this?

1.5 Objectives and scope

The objectives of this project are as follows:

1. Define a set of functional and non-functional requirements for a general-purpose digital shadow application.

2. Create a simple JSON format for 2D and 3D visualizations to serve as a declarative interface for the HTML canvas API.

3. Define a domain-specific language with a simple syntax, for which it is easy to write an interpreter.

4. Create an interpreter in JavaScript for the defined language, to make it possible to evaluate user-submitted code while restricting all access to the browser API, so as to enable users to safely submit and run widget logic.

5. Based on the points above, create a prototype of a general-purpose digital shadow application satisfying the functional requirements for the project.

6. Evaluate how well the prototype fulfills the functional and non-functional requirements for a general-purpose digital shadow application and use this to answer the research questions and discuss whether the thesis statement is strengthened or invalidated.

1.6 Project deliverables

The main deliverable for the project is this thesis. The compiled prototype of the general-purpose digital shadow application, along with its source code are delivered as supplementary attachments along with the thesis. It is also published on GitHub.

1.7 Limitations

The general-purpose digital shadow application developed as part of this project does not aim to be a commercial solution and has mainly been implemented for testing and illustrative purposes.

While the application works well in its current state, its feature set is limited to what was relevant for testing. For instance, the application lets users write logic in a non-JavaScript language to prevent XSS vulnerabilities which could be used to steal other users credentials. However, the application does in its current state not deal with user credentials, so while it served its purpose for testing how much slower code execution might be if one would have to consider the chance of XSS attacks, it is not as performant as its current set of features would allow it to be. A more exhaustive list of limitations of the implemented prototype are

described in *Chapter 5 - Implementation*, showcased in *Chapter 6 - Results* and discussed in *Chapter 7 - Evaluation and discussion*.

Furthermore, time has not been spent on performance optimizations for the implemented interpreter, as the thesis statement would be further strengthened if challenged not only with a highly optimized best-case scenario.

1.8 Structure of the thesis

Chapter 1 - Introduction
Presents the background and motivation for the thesis and the thesis statement to be tested, and raises some additional research questions the thesis should aim to answer.

Chapter 2 - Theoretical background
Summarizes the history of the digital twin and related concepts, with particular focus on the digital shadow. The chapter also explains core technologies, challenges and concepts which it is assumed that the reader is familiar with in the following chapters.

Chapter 3 - Requirements and reasoning
Translates the use-cases research papers have suggested for digital shadows, as defined in *Chapter 2 - Theoretical background*, into a set of functional requirements describing what capabilities a general-purpose digital shadow should have, as well as a set of non-functional requirements, against which general-purpose digital shadow applications can be evaluated.

Chapter 4 - Method
Explains how a prototype satisfying the functional requirements presented in *Chapter 3 - Requirements and reasoning* will be developed, assuming that the thesis statement presented in *Chapter 1 - Introduction* is valid. The chapter further explains how the implemented solution will be evaluated against the non-functional requirements which were also presented in *Chapter 3 - Requirements and reasoning* to determine whether there is a gap between the non-functional requirements and the operation of the implemented prototype and whether

those gaps are caused by the thesis statement in *Chapter 1 - Introduction* being invalid, or by other simplifications made during implementation.

Chapter 5 - Implementation

Describes how a prototype for a general-purpose digital shadow application was developed to satisfy the functional requirements presented in *Chapter 3 - Requirements and reasoning*. It presents the technologies used and explains why they were chosen. A particular focus has been put on the the development of the highly configurable 2D and 3D visualization widgets and the interpreter implemented to be able to safely evaluate user-submitted widget code. The chapter also covers challenges which presented themselves during implementation, how they were solved and what shortcuts were taken.

Chapter 6 - Results

Gives a detailed walkthrough of all of the features of the general-purpose digital shadow application prototype developed whose development was covered in *Chapter 5 - Implementation*, by creating a digital shadow of a bascule bridge, based on random data from the RANDOM.ORG HTTP interface.

Chapter 7 - Evaluation and discussion

Evaluates the implemented prototype against the non-functional requirements presented in *Chapter 3 - Requirements and reasoning* to determine whether there is a gap between the non-functional requirements and the operation of the implemented prototype. The chapter further aims to determine whether those gaps are caused by the thesis statement in *Chapter 1 - Introduction* being invalid, or by other simplifications made during implementation. Lastly, the chapter discusses any findings relevant to the research questions presented in *Chapter 1 - Introduction*.

Chapter 8 - Conclusion

Concludes, whether the results from *Chapter 6 - Results* and the evaluation of those in *Chapter 7 - Evaluation and discussion* reject or strengthen the thesis statement presented in *Chapter 1 - Introduction*. This chapter also summarizes the most relevant answers, if any to the research questions which were also presented in *Chapter 1 - Introduction*.

Chapter 9 - Further work

Suggests further research based on which of the research questions presented in chapter 1 were not sufficiently answered by this project, as well as new research questions which have appeared along the way.

Chapter 10 - Bibliography

Lists all sources which are cited or referred to in the thesis.

Chapter 11 - Appendices

Contains additional material which may be of relevance:

Appendix A - Dashboard Code example from chapter 6

Contains the whole user-submitted code used in the example use-case in *Chapter 6 - Results*.

Appendix B - Running the prototype application

Provides information about how to run the prototype application developed as part of this project and delivered alongside the thesis.

Appendix C - Digital Shadow Language examples

Contains examples of code written in Digital Shadow Language, the Lisp-like domain-specific programming language designed as part of this project. Each function or special form will not be explained in detail, but the examples aim to be simple enough to be intuitively understandable and make the reader familiar with the syntax of the language.

2 Theoretical background

2.1 Overview

This chapter summarizes the history of the digital twin and related concepts, with particular focus on digital shadows. The chapter also explains core technologies, challenges and concepts which it is assumed that the reader is familiar with in the remaining chapters.

2.2 The digital twin

The following paragraphs aim to build an understanding of the origin of the concept of the digital twin, definitions of the digital twin which have surfaced in research papers since then, and similar types of systems which, as a result of misconceptions, have wrongly been identified as digital twins.

2.2.1 The origin of the digital twin

The term "Digital Twin" was introduced in the book Virtually Perfect: Driving Innovative and Lean Products through Product Lifecycle Management (4) in 2011 by Michael Grieves, who in turn attributed it to John Vickers of NASA with whom he had worked. However, the concept which would evolve to be known as the digital twin was initially introduced by Grieves already in 2003, in a course on product lifecycle management at the University of Michigan. (5)

2.2.2 Definition of the digital twin

In his Digital Twin White Paper (5), Grieves defines a digital twin as a virtual, digital equivalent to a physical product, consisting of three main parts: *"Physical products in Real Space"*, *"virtual products in Virtual Space"*, and *"the connections of data and information that ties the virtual and real products together"*. (5)

Grieves further describes a set of Digital Twin Fulfillment Requirements, which, among others require a two-way connection between the virtual and the physical product, with data

flowing from the physical product to the virtual product and information and processes flowing from the virtual product to the physical product. (6)

2.2.3 Later definitions of the digital twin

With the emergence of the idea of Industry 4.0 the idea of the digital twin grew in popularity and companies began using the term for marketing purposes, not always consistently (7). Research and advisory company Gartner in particular has played a strong role in popularizing the term, although not always adhering to the definition and requirements presented by Grieves (8). Over the years, numerous definitions of the digital twin have surfaced in research papers, some of which are presented here (9):

> *A Digital Twin is an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin.* (10)

> *A digital twin is a computerized model of a physical device or system that represents all functional features and links with the working elements.* (11)

> *The digital twin is actually a living mode of the physical asset or system, which continuously adapts to operational changes based on the collected online data and information, and can forecast the future of the corresponding physical counterpart.* (12)

> *A digital twin is a set of virtual information that fully describes a potential or actual physical production from the micro atomic level to the macro geometrical level.* (13)

> *A digital twin is a digital representation of a physical item or assembly using integrated simulations and service data. The digital representation holds information from multiple sources across the product life cycle. This information is continuously updated and is visualized in a variety of ways to predict current and conditions, in both design and operational environments, to enhance decision making.* (14)

A Digital Twin is a virtual instance of a physical system (twin) that is continually updated with the latter's performance, maintenance, and health status data throughout the physical system's life cycle. (15)

2.2.4 Misconceptions

In their paper Digital Twin: Enabling Technologies, Challenges and Open Research (9), Fuller, Fan and Day describe three main types of systems which are typically referred to as digital twins, only one of which adheres to the definition established by Grieves. They are as follows:

Digital Model

A digital model is described as "*a digital version of a preexisting or planned physical object.*" (9) It is further described as having "*no form of automatic data exchange between the physical system and digital model.*" (9) In other words, a change made to the state of the physical object is not automatically reflected in the digital model, and a change made to the state of the digital model is not automatically reflected in the physical model.

Digital Shadow

The paper further continues by describing a digital shadow as "*a digital representation of an object that has a one-way flow between the physical and digital object.*" (9) Here, a change made to the state of the physical object will automatically be reflected in the digital object, but a change in the state of the digital object will not automatically be reflected in the physical object.

Digital Twin

Lastly, the paper concludes that "*If the data flows between an existing physical object and a digital object, and they are fully integrated in both directions, this constituted the reference "Digital Twin". A change made to the physical object automatically leads to a change in the digital object and vice versa."* (9)

2.3 The digital shadow

In the majority of cases where the term "Digital Twin" is incorrectly used to describe a system which is not a digital twin, the system is in reality a digital shadow. In (9), the authors categorically reviewed 26 papers on digital twins and revealed that eleven of the systems described as digital twins were in reality just digital shadows, with no data connection from the virtual entity to the physical. Of the remaining, eleven did indeed adhere to Grieves' definition of the digital twin, while two were categorized as digital models. The remaining two papers did not provide a description of the digital twin and could not be classified. (9)

A major contributor to the digital twin term being used to describe digital shadows is global research and advisory company Gartner, who have introduced their own definition of the digital twin, relaxing the digital twin requirements specified by Grieves:

> *Gartner defines a digital twin as a software design pattern that represents a physical object with the objective of understanding the asset's state, responding to changes, improving business operations and adding value.* (16)

They also provide another definition:

> *A digital twin is a digital representation of a real-world entity or system. The implementation of a digital twin is an encapsulated software object or model that mirrors a unique physical object, process, organization, person or other abstraction. Data from multiple digital twins can be aggregated for a composite view across a number of real-world entities, such as a power plant or a city, and their related processes.* (8)

While the term digital twin may be most popularly associated with digital representations of physical entities, their value proposition can in many cases be extended to entail digital representations of non-physical systems as well, like company structures or application infrastructure.

For instance, Gartner has also introduced the concept of a "Digital Twin of an Organization" (DTO) for which they provide the following definition:

> *A digital twin of an organization (DTO) is a dynamic software model of any organization that relies on operational and/or other data to understand how an organization operationalizes its business model, connects with its current state, responds to changes, deploys resources and delivers exceptional customer value.* (17)

These definitions only describe a one-way data connection, from the real-world entity to the virtual entity. Since there is no data flowing from the virtual entity to the real-world entity, these systems do not adhere to the digital twin definition established by Grieves, but can better be categorized as digital shadows.

Based on the definition of a digital shadow, a digital shadow has much in common with any other application providing data-driven visualizations. While the digital shadow is not as clearly defined in research as the digital twin, its proposed use-cases suggest that a digital shadow should be able to not only display data, but also be able to present the data in the context of a model, which simulates the known dynamics of the entity from which the data originates. As such a digital shadow can also be used to present derived data, and do so not only by using a set of predefined graphs, but by providing 2D or 3D visualizations which closely resemble the real-life entity.

2.3.1 Digital shadow use-cases

Use-cases which have been proposed for the digital shadow largely overlap with those which have been proposed for the digital twin. In the Digital Twin White Paper (5), Grieves lists three categories of use-cases for the digital twin, which are largely transferable to the digital shadow:

> **Conceptualization**
> *The capability of the digital twin lets us directly see the situation and eliminate the inefficient and counterproductive mental steps of decreasing the information and*

translating it from visual information to symbolic information and back to visually conceptual information. (5)

With the digital twin to build a common perspective, we can directly see both the physical product information and the virtual product information, simultaneously. (5)

Comparison

With the digital twin model, we can view the ideal characteristic, the tolerance corridor around that ideal measurement, and our actual trend line to determine for a range of products whether we are where we want to be. Tolerance corridors are the positive and negative deviations we can allow before we deem a result unacceptable. (5)

Collaboration

The digital twin capability with its conceptualization, comparison, and collaboration capability frees us from the physical realm where humans operate relatively inefficiently. We can now move to virtual realm where physical location is irrelevant, and humans from across the globe can have common visualization, engage in comparisons identifying the difference between what is and what should be, and collaborating together. (5)

Below, some more specific digital shadow use-cases are listed. These will be revisited in chapter 3, when specifying the functional requirements for a general-purpose digital shadow.

Live visual models

Digital shadows makes it possible to bring life to visual models, either in 2D or 3D, making sure the state of the model matches the physical entity. Such state may for instance represent the position and orientation of an entity. An example of this is a car navigation system, where the position of the car is continuously visualized on a map.

Live reports

Continuously mirroring a physical entity, digital shadows are well suited for creating live documentation, where non-static properties are continuously updated to match the physical counterpart being documented. Additionally, digital shadows make it possible to create reports whose charts and numbers are continuously updated.

Present derived data

Digital shadows can be used to present derived data which is continuously updated to use new data in their calculations. This can be used to present the data in a more digestible form, or to predict future estimates based on the current and past data. For instance, in a production process, a digital shadow can be used to map operating parameters to expected outcomes, converting the information from the technical domain to a more business-oriented domain, which is more actionable (18).

Generate what-if scenarios

A digital shadow can be used to evaluate different scenarios by provide alternative values to the live data. This can be used to optimize parameters, or test worst case scenarios (18).

Predictive maintenance

Rather than performing maintenance on equipment at regular intervals, digital shadows can be used to perform predictive, or condition-based maintenance. This means that maintenance can be limited to when the data provided from the physical entity suggests that a part is about to fail. This can be predicted based on historical data, or when a specific condition is met.

Sanity checking

In cases where the value of one sensor can be derived from the value of another sensor, given what we know about the physical entity, the values can be compared. Large divergences may indicate that a sensor is faulty, or that our understanding of the physical entity is no longer correct, which might be a sign of fatigue.

2.4 Existing digital twin and digital shadow software

2.4.1 AWS IoT Device Shadow service

The aim of AWS IoT Device Shadow service is to make the state of a device available to apps and other services. The shadow acts as an interface to both monitor and request changes to the device's state. As such, it satisfies the digital twin requirement of a bidirectional data flow between the physical device and its virtual representation, thus being able to provide both digital shadow and digital twin capabilities. (19)

2.4.2 Eclipse Ditto

Eclipse DItto is a backend IoT solution which aims to facilitate setting up digital twins by making it easy to expose connected devices, or "things" as web services, removing the need for a custom backend solution. It does so by routing requests between application and hardware, maintaining last reported state of hardware for when it's not connected, as well as providing notifications about changes. (20)

Eclipse Ditto is not a complete digital twin solution, but a module which can be responsible for some of the backend responsibilities in a bigger system. As such, it can be used for both digital twins and digital shadows. (20)

2.4.3 Microsoft Azure IoT Hub Device Twins

Device twins in Microsoft Azure IoT Hub are JSON documents that store information about a device's state. For each device connected to Azure IoT Hub, a device twin is implicitly created. A device twins includes read-only data about a device, reported properties and desired properties. These can be used by a digital twin application to mirror the state of the device, and make updates to the device, thus enabling bidirectional data flow between the physical and virtual device. Just like the AWS IoT Device Shadow service and Eclipse Ditto, Microsoft Azure IoT Hub Device Twins can be used for both digital twins and digital shadows. (21)

2.5 REST API

REST is an acronym for **RE**presentational **S**tate **T**ransfer, and is a set of constraints which must be satisfied for an interface to be considered RESTful. It was introduced in year 2000 by Roy Thomas Fielding and has become a de facto standard way for offering web services. RESTful interfaces rely solely on URIs for resource detection and interaction, and typically HTTP for message transfer. The predefined HTTP verbs (GET, DELETE, etc.) are used to define the operation to be performed on the selected resource. (22)

2.6 WebSocket

The WebSocket protocol is a protocol which enables two-way communication between the user's browser and a server. Both HTTP and the WebSocket protocol utilize a TCP connection, but while HTTP requires that a new connection be opened for each new message, the WebSocket protocol is able to open and maintain a single TCP connection over which the client can send messages to the server and receive event-driven responses without having to poll the server for a reply. (23)

2.7 Single-page applications

A single page application (SPA) is a web application consisting of a single HTML document, where navigation and interactivity is achieved by modifying all or some of the web page using JavaScript, rather than navigating between different, more or less static HTML documents which need to be fetched from a server on navigation. (24)

Compared to traditional websites where the client's sole responsibility is to display the HTML provided by the server, SPAs run comparably more code on the client side, handling both business logic and view logic, much like a native application. This comes with advantages such as more responsive UI, and in many cases the possibility to utilize simpler, more general-purpose backend solutions. (24)

Even if SPAs remove the need of fetching additional pages after initially being loaded, they may still be reliant on subsequent calls to the server. However, rather than fetching a new page from the server, individual pieces of data can be fetched from the server on an as-needed basis. This makes it possible to create large and complex SPAs without initially having to load all the data which the application might need at some point, but which is not be relevant for the current view. (24)

2.8 Persistent storage in client-side applications

With SPAs adapting a role similar to native applications, one might want to be able to store data locally. This can be useful if for instance an application should be able to persist data even without an internet connection or one does not have access to a remote database.

Before the introduction of HTML5, the only way to store data locally was in the form of cookies, which can only store a few kB of data and have the disadvantage of being included in every server request, making them non-ideal for storing sensitive data.

The Web Storage API, however, makes it possible to more securely store larger (several MB) amounts of data in the browser. It provides two objects for storing data on the client, namely localStorage or sessionStorage. Data written to localStorage does not expire. Data written to sessionStorage, on the other hand, expires at the end of the session. (25)

2.9 Domain-specific languages

A domain-specific language (DSL), as opposed to a general purpose language (GPL), is a computer language which is tailored to a particular domain or use-case. By sacrificing generality, a DSL can be much more expressive and easy to use than a GPL within a particular domain. This can increase productivity, reducing maintenance costs and the need for programming expertise, thus making the domain available to a larger group of developers than a GPL would. In other cases, a DSL might be characterized not by being more convenient for the programmer, but by being more suitable for a particular use-case, for instance by adhering to a stricter subset of a language. This makes it possible to give the developers access to a language with a syntax they might be familiar with from another

language, while for instance removing the possibility to perform side effects or write programs which never terminate. (26)

2.10 Running user-submitted code in the browser

2.10.1 Cross-site scripting

Before covering the challenges of running user-submitted code in the browser, the reader should be familiar with the concept of cross-site scripting (XSS).

In her paper *Security against cross site scripting (XSS) attacks: signature based model on server side.* (27), Sonali Nalamwar gives the following description of XSS:

> *In typical cross site scripting the target views a website which contains code inserted into the HTML which was not written by the website designer or administrator. This bypasses the document object model which was intended to protect domain specific cookies (sessions, settings, etc.). In most instances the target will sent a link to a website on the server which the target has a legitimate account and by viewing that website the attackers malicious code is executed (commonly JavaScript is used to sent the user's cookie to a third party server, in effect stealing their session and their account).* (27)

2.10.2 XSS in user-submitted content

A large fraction of XSS attacks are caused by dynamic, typically user-submitted content being included on a web page without being validated for malicious content. (28)

Supporting user-submitted content in the form of primitive data, e.g. text, numbers or boolean values typically do not carry any inherent risk as long as they are presented in their pure form, without being parsed as code. However, when user-submitted content is parsed or inserted as HTML, for instance, this presents an opportunity for hackers to inject malicious scripts which can be run in other users' browsers. (29) Use-cases where user-submitted content has to be parsed as HTML include rich-text editors which support a subset of HTML. (30)

HTML is the markup language used to define the Document Object Model (DOM), which can be further modified using JavaScript. JavaScript and HTML can exist within the same document. When JavaScript is included in an HTML document, it resides within <script> nodes. Initially, this makes it very easy to separate JavaScript from HTML when parsing user-submitted content. However, some of the attributes accepted by HTML elements are evaluated as JavaScript (31). This makes it challenging to display user-submitted HTML without the risk of running any user-submitted scripts. There are, however, well-tested libraries which remove most or all of the most common XSS attack vectors.

2.10.3 Deliberately running user-submitted code in the browser

Deliberately running user-submitted JavaScript in the browser is a very different problem. One might want to let the user write custom JavaScript calculations to perform data transformations to run in the browser, for instance. In this case we are interested in letting the user write JavaScript code which evaluates to some value. However, by directly evaluating the code as JavaScript, for instance using the native JavaScript eval function, we also let the user submit code which has access to the whole browser API, which means that it can access cookies and localStorage, send HTTP requests on behalf of the user, modify the DOM or navigate the user to other websites (32).

This is mainly a concern if the user-submitted JavaScript will be available to other users. Otherwise, the malicious user will only cause trouble for themselves. However, as a website provider it is desirable that a user is not able to submit a script which accidentally starts a never-ending loop, for instance, blocking all other script execution and rendering the page unusable for the user (33).

Evaluating user-submitted JavaScript in the client without the risk of accessing user credentials or perform actions on behalf of the user is a very difficult problem. There exists several approaches to sandbox JavaScript. While these approaches can do a good job protecting against many XSS attack vectors, defending against all of them is very difficult.

2.10.4 Approaches to safely run user-submitted code in the browser

Running untrusted code in sandboxed iframes

The HTML iframe tag enables a nested browsing context where each nested page's scope and DOM is isolated from their parent page. As of HTML5 iframes also have a sandbox attribute, which allows for fine-grained control of restrictions to impose on the content of the iframe. (34) Scripts from different origins cannot access each other, so by hosting the content of the iframe in a separate origin, the restrictions of the iframe's same-origin policy can be utilized to enforce that all communication happens via the Window.postMessage() method, which enables safe cross-origin communication between a page and the iframe embedded within it (35). This means that iframes can be used to safely evaluate JavaScript, or any other language for which there is an interpreter written in JavaScript. It is worth noting that iframes still share the same thread/process as the parent page, so sandboxing code in an iframe does not prevent the possibility of denial of service attacks, where non-terminating code blocks all other script execution, which in turn causes the page to freeze. Furthermore, sandboxing code in iframes can be cumbersome, because of the iframe content having to be hosted on a separate, safe domain and having to deal with setting up the low level cross-domain messaging (36).

Not using the JavaScript engine provided by the browser

Another way to prevent the user from performing unwanted side effects with JavaScript is to simply not interpret the code using the browser's native JavaScript interpreter. Instead, the code can be parsed to an abstract syntax tree. The abstract syntax tree can then be then be evaluated, and keeping track of a local scope, we can validate that no variable or property not defined inside the local scope can be accessed. The parser can either be written to accept a JavaScript-like language to let users take advantage of any familiarity they might have with JavaScript, or a simpler language, which is easier to parse.

Js.js (37) is a JavaScript library approaching the challenge of sandboxing JavaScript in this way. Instead of being written from scratch, it is created by compiling Mozilla's JavaScript runtime SpiderMonkey, which is written in C and C++, to LLVM, and then translating the result to JavaScript using emscripten. In Chrome, the resulting JavaScript interpreter is

around 100 to 200 times slower than when being evaluated using the browser's native JavaScript interpreter. (37)

Not having any credentials to steal nor users to act maliciously on behalf of

Being able to run and execute JavaScript in another user's browser is not necessarily a serious security concern in itself. After all, all the JavaScript a user runs in their browser is written by someone else, unless they are among the creators of the web page.

One way to circumvent the problem of user-submitted JavaScript stealing credentials or acting on behalf of other users is to simply remove the concept of users and authentication. Rather than sharing user-submitted JavaScript from one logged in user to another via the platform, with the risk of the code acting on behalf of the other user or stealing the other user's credentials, the website can simply be exported with the user-submitted JavaScript and sent as a file, or embedded on the creator's website.

In this case the creator of the code has taken on the role of the application developer. The only sensitive information available is the data provided by the developer, and it is in the developer's interest that it is not misused.

If the exported file containing user-submitted JavaScript does indeed contain sensitive data, the developer might want to encrypt it, to only make it available to specific people. It might be easy for the developer to execute code in the file recipient's browser to steal their decryption key when decrypting the content, but this is similar to how it is possible for web developers in general to steal and misuse their users' credentials. While possible, this is not desirable for the developers since it is in the developers' interest to that their users' accounts stay safe.

2.11 Lisp (programming language)

Lisp is a family of programming languages, originally defined in 1958 by John McCarthy as part of his work at Massachusetts Institute of Technology (MIT) (38). It was described in his paper *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part*

I (39) in 1960, showing that with a few essential operators and anonymous functions similar to those in lambda calculus, a Turing-complete language could be defined.

One of Lisp's major data structures are linked lists and the name Lisp comes from its focus on list processing. One of the innovative features of Lisp was that the source code was itself made up of lists, making it possible for Lisp code to treat source code as data. This is the foundation for Lisp's macro system, which makes it possible to extend the syntax of the language. (40)

The syntax of Lisp is easily recognizable, consisting only of expressions and heavy use of parentheses. Unlike in most other languages, there is no concept of a statement. Function calls are written as lists, where the first element is an expression evaluating to a function, and the remaining elements are expressions whose value will be passed as parameters to the function. (39) The simple syntax and expression-oriented structure makes Lisp languages particularly simple to parse, compared to other languages with more complex syntax.

Lisp has evolved into many dialects which, despite having evolved in different directions are still easily recognizable, in part thanks to their heavy use of parentheses and prefix notation. Some major Lisp dialects include Common Lisp, Scheme and Clojure (41).

3 Requirements and reasoning

3.1 Overview

This chapter translates the use-cases research papers have suggested for digital shadows into a set of functional requirements describing what capabilities a general-purpose digital shadow should have, as well as a set of non-functional requirements, against which general-purpose digital shadow applications can be evaluated.

3.2 Functional requirements

The functional requirements of a system describe its intended core functionality, i.e. what the application should enable the user to do. Based on the use-cases listed in chapter 2.3.1, the following functional requirements were derived:

It should be possible to create dashboards
All of the listed digital shadow use-cases revolve around being able to present live data of an entity, be it through visualizations, text or derived values. A dashboard enables getting an overview of important data in a single interface.

It should be possible to create numerical widgets as functions of live data
The value proposition of a digital shadow lies in being able to present data in a context in which the data makes more sense. In some cases this might be best achieved by simply presenting key numerical values derived from the available data.

It should be possible to visualize any model, both in 2D and 3D
In many cases, the state of an entity may most intuitively be represented by a 2D or 3D visualization.

It should be possible to create visualization widgets as functions of live data

Being able to define visualizations as a function of continuously or frequently updated data is essential for the visualizations to be able to serve as a digital shadows, rather than just digital models.

It should be possible to use live data from REST APIs and WebSocket APIs

Common for most of the endpoints relevant to the use-cases the digital shadow platform targets, is that they expose REST-APIs intended for sporadic or periodic HTTP requests. For some of the more dynamic use-cases, where it is desirable to visualize continuous streams of data, having support for WebSocket APIs can be beneficial.

It should be possible to use data from multiple API endpoints

It cannot be expected that all data needed for a particular dashboard or visualization can be accessed from a single endpoint. One of the digital shadow's value propositions is to be able to gather data from multiple data sources and use it to give a holistic overview of the entity being monitored.

It should be possible to select the frequency with which new data is fetched

Since the update frequency of the data sources that the digital shadow is based on can vary a lot from use-case to use-case. Some data sources are continuously updated, others may be updated with new data every ten seconds. For use-cases like the DTO, there may be several days between each time data is updated (17). Because of this, it does not make sense that the digital shadow application enforces a predefined update frequency. A too slow rate might leave the user annoyed or lower their trust in the data and a too high rate might cause many unnecessary API calls. If the user's dashboards depend on heavy calculations, fetching data too often can cause unnecessary strain on the user's computer, which in the worst case might freeze the UI for a moment or cause noise from increased computer fan speed. Additionally, the frequency at which data updates can vary a lot from data source to data source, so it should be possible to set the frequency at which data from each individual data source is fetched.

For variable amounts of data, it should be possible to let the amount of widgets depend on the amount of data

There may be cases where a chosen data-source does not have a one-to-one correspondence to a particular value or entity. An example of this would be the Oslo City Bike Realtime data API, which exposes data about bike dock availability at city bike stations in Oslo (42). In this particular case the amount of bike stations might be known by the user creating a dashboard, but having to define a static amount of widgets would require that the dashboard be manually updated every time a new bike station was added. An essential capability of digital shadows is to be able to mirror real-world entities and systems, and requiring the user to manually update their dashboards to reflect new data would limit the digital shadow's ability to do this.

It should be possible to create documents as functions of live data

Documenting entities which are dynamic in nature quickly leads to outdated documentation, which cannot be trusted. While visual and numerical widgets might work well for giving an overview of an entity, the document format is more suitable for extensive, more detailed documentation.

It should be possible to share created digital shadows

One of the three overarching use-cases described by Michael Grieves in his Digital Twin white paper was collaboration. He described that "*humans from across the globe can have common visualization, engage in comparisons identifying the difference between what is and what should be, and collaborating together.*" (5) This is equally applicable to the digital shadow, and as such, being able to share digital shadows is of the essence.

It should be possible to define calculations to calculate derived values

Being able to define and present derived values not only makes it possible to convert values into more business-oriented insights. It is also fundamental for being able to combine data from different sources and define visualizations.

It should be possible for the user-defined calculations to have access to both current and past data

Use-cases like predictive maintenance or any other use-case depending on performing estimations about the future are dependent on having access to past data, either to extrapolate past data into future data, or to simply compare current data to past data.

3.3 Non-functional requirements

The non-functional requirements of the system describe the intended qualities or characteristics of the system. Based on the defined digital shadow use-cases, the following non-functional requirements were defined, describing the qualities by which the functionality of the system should be evaluated:

Availability

- It should be possible to run the application without having to install anything, assuming that any major web browser is installed.

- It should be possible to run the application using any operating system in which it is possible to install any major web browser.

Extensibility

- The application should be highly modular, making it easy to use as a module in a bigger system.

Performance

- The application should have a low bundle size to be able to load quickly.

- User-submitted calculations performed in the frontend should appear to be instant, without any noticeable freeze of the UI.

- All animated visualizations should have a high enough frame rate to not appear laggy.

Usability

- The widget creation interface should not make limiting assumptions about the user's visualization requirements.

- It should be easy to define visualization widgets.

- It should be easy to define numerical widgets.

Modifiability

- It should be easy to continuously deliver new versions of the application, without the user having to do anything to update to the latest version.

4 Method

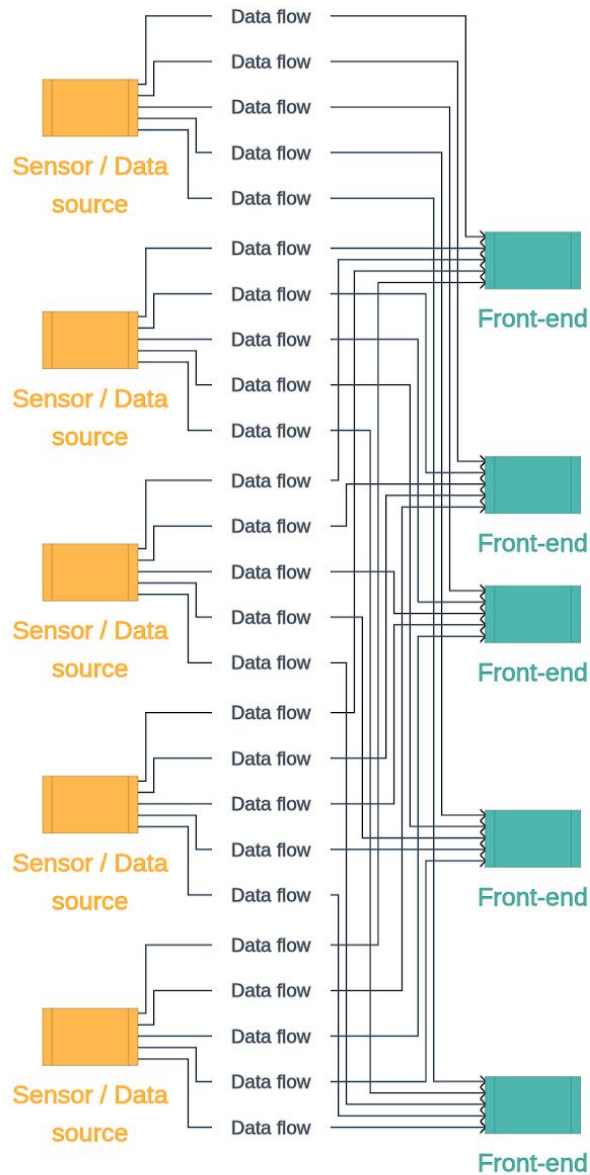
4.1 Overview

This chapter explains how a prototype satisfying the functional requirements presented in *Chapter 3 - Requirements and reasoning* will be developed, assuming that the thesis statement presented in *Chapter 1 - Introduction* is valid. The chapter further explains how the implemented solution will be evaluated against the non-functional requirements which were also presented in *Chapter 3 - Requirements and reasoning* to determine whether there is a gap between the non-functional requirements and the operation of the implemented prototype and whether those gaps are caused by the thesis statement in *Chapter 1 - Introduction* being invalid, or by other simplifications made during implementation.

4.2 Deciding to implement a pure client-side prototype

Being able to run user-submitted calculations on the frontend opens up the opportunity for creating a pure frontend general-purpose digital shadow system, a general-purpose digital shadow application which can run with no backend at at all.

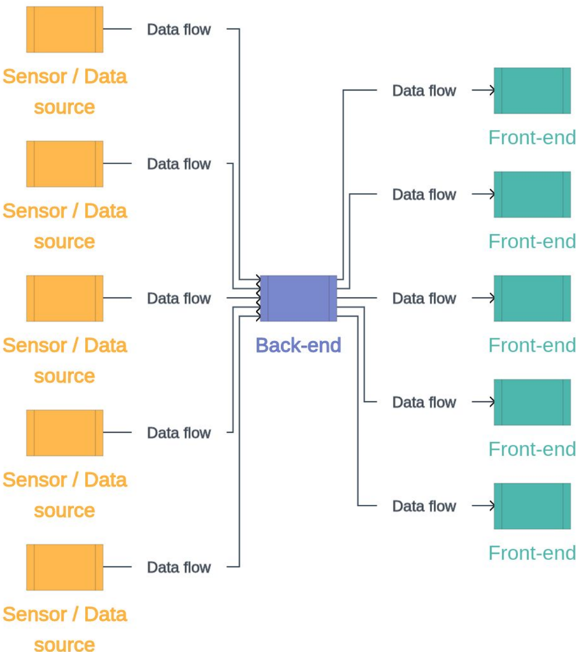
A pure frontend application has the advantages of avoiding server/license costs, and can be used without any form of user authentication.



The image above illustrates the architecture of a "backendless" digital shadow application, where data collection happens directly from the frontend.

A general-purpose digital shadow application utilizing a backend, however, for instance in the form of a SaaS application, comes with several advantages. For instance, fewer calls are made to each data source API. This is particularly important if the data source endpoints are rate-limited. Then it is preferable if data is requested from these APIs at predictable intervals, rather than at intervals which depend on the amount of concurrent users. Another advantage is that data can be stored on the server, making it possible to present historical values and perform aggregations over time. Furthermore, dashboards can be shared without giving direct access to the data source APIs they are based on. The API keys stay safely hidden on the

server and are not exposed in the frontend code. Furthermore, the client only needs to fetch data from one endpoint. Storing project data centrally also means that projects can be accessed from any device with an internet connection. Lastly, a backend service does not expose its source code to the user, and as such it is more sellable.



The image above illustrates the architecture of a digital shadow application utilizing a backend for data collection.

While a pure frontend and a hosted/SaaS digital shadow solution both can solve many of the use-cases the digital shadow aims to solve, it is evident that the hosted/SaaS solution has several capabilities that are missing from the pure client-side application.

However, since the purpose of the prototype being developed for this project is to evaluate the feasibility of running user-submitted code in the browser, it does not matter whether the application connects to a backend, or is developed as a pure frontend application.

To not add unnecessary complexity to the experiment, and also to make it easier to answer the research question about whether a general-purpose digital shadow platform can be created with no backend, the prototype was implemented as a pure frontend general-purpose digital shadow application.

4.3 Using the prototype to evaluate the thesis statement

The aim of the prototype was to satisfy the functional requirements defined for a general-purpose digital shadow application, was implemented under the assumption that

> *Web technologies have evolved so much in recent years that user-submitted transformation and aggregation logic can be handled in internet browsers, even if the logic is supplied in a DSL not native to the browser and has to be interpreted by an interpreter written in JavaScript.*

as defined in the thesis statement.

Once implemented, the prototype was evaluated against the non-functional requirements which were defined for a digital shadow application in *Chapter 3 - Requirements and reasoning*. If the prototype were to satisfy the non-functional requirements, that would strengthen the thesis statement. If some non-functional requirements were not satisfied by the prototype, I would reason about whether this was caused by the thesis statement being incorrect, whether it was only applicable to specific cases of digital shadow use-cases or whether it was caused by other invalid assumptions or simplifications made while developing the prototype.

5 Implementation

5.1 Overview

This chapter describes how a prototype for a general-purpose digital shadow application was developed to satisfy the functional requirements presented in *Chapter 3 - Requirements and reasoning*. It presents the technologies used and explains why they were chosen. A particular focus has been put on the the development of the highly configurable 2D and 3D visualization widgets and the interpreter implemented to be able to safely evaluate user-submitted widget code. The chapter also covers challenges which presented themselves during implementation, how they were solved and what shortcuts were taken.

While the reason for implementing a safe way to run untrusted user-submitted code on the frontend is to facilitate running code from other users, effort was not spent implementing any form of user management or centralized solution to store user-submitted code. The goal of the prototype was not to end up as a sellable solution, but to serve as a platform for testing various approaches to safely share live documentation which depend on running untrusted user-submitted code. The prototype would serve this purpose by enforcing that dashboards be written in a safe domain-specific language, and through exporting a read-only subset of the application which could be shared as a single HTML document, without any user management requirements.

The tools chosen to develop the user interface of the prototype will only be described briefly, as they are of little relevance to the challenges of interpreting user-submitted calculations in the browser. The implementation of the interpreter for a domain-specific language, however, will be explained in greater detail.

5.2 Making a single-page application

The prototype for the general-purpose digital shadow application was developed as a single-page application. Since the main goal of the prototype was to evaluate the feasibility of

running user-submitted calculations in the browser, it was given that the prototype would need to be a web application, as opposed to a native application. There were several reasons for deciding to implement the prototype as a single page application instead of a multi-page application:

- Navigation between views would be instant, with no need to fetch a new page from the server.
- Working with a single page would make it easier to maintain state between views.
- The need to duplicate logic which is used by several views would be avoided.
- The prototype could be compiled to a single HTML file, making it easy to embed into other web pages and to run locally or share.
- Having the whole application in a single file would make it easier to export a read-only subset of the application with the purpose of sharing live dashboards, and maintain the application's routing capabilities. In the end, the exported read-only subset of the application did not end up using routing after all, since it would only export a single dashboard view.

5.3 Routing

Routing is the logic which makes it possible to render different content based on the URL in the browser's address bar. Since single-page applications consist of a single page and it is the frontend which is responsible for rendering the correct content, the same page has to be returned from the server, regardless of the URL entered in the address bar. This requires that the server that the application is requested from is configured to do so. While this is the normal approach for modern single-page applications, accessing routes through the address bar will not work properly when running the application locally as an HTML document.

```
https://example.com/project/0/dashboard
```

URL-based routing in single-page applications requires that the server be configured to deliver the same file, even if resources from different locations are requested.

To avoid the need for server configurations, and the need for a server at all for that matter, the prototype was instead developed using to use hash-based routing. Hash-based routing is routing which utilizes the anchor part of the URL to simulate different paths.

```
https://example.com/#/project/0/dashboard
```

Hash-based routing in single-page applications does not require special server configurations. The anchor part of the URL is only available to the frontend.

For example, in the case of the URL https://example.com/#/projects/0/dashboard, https://example.com/ is the resource which will be requested from the server. The anchor part of the URI (#/projects/0/dashboard) is only available to the frontend. By using hash-based routing, the prototype did not need to depend on correctly configured servers, and could be run as a single HTML file with working routing, even when run locally.

For the prototype, a simple hash-based router was created to make it easy to specify parameterized paths.

```
const router = isExported
  ? new Router({
      '/': 'Dashboard',
    })
  : new Router({
      '/': 'Projects',
      '/projects/<projectId:string>': 'Data sources',
      '/projects/<projectId:string>/data-sources': 'Data
sources',
      '/projects/<projectId:string>/values': 'Values',
      '/projects/<projectId:string>/values/edit': 'Edit values',
      '/projects/<projectId:string>/dashboard': 'Dashboard',
      '/projects/<projectId:string>/dashboard-editor': 'Edit
dashboard',

'/projects/<projectId:string>/dashboard-editor/widgets-preview':
        'Edit dashboard',
      '/projects/<projectId:string>/dashboard-editor/raw-output':
        'Edit dashboard',
      '/projects/<projectId:string>/dashboard-editor/problems':
        'Edit dashboard',
    });
```

The code above shows how different routes map to different views in the prototype. The router maintains two variables: currentRoute and params. The URL https://example.com/#/project/0/dashboard would result in currentRoute being set to '/projects/<projectId:string>/dashboard' and params being set to { projectId: '0' }. An event listener listens for hash changes, i.e. when the anchor part of the URL changes, and when a hash change occurs, the UI components depending on these values will be updated with new properties. Note that if the application has been exported, only one route is available. This will be further covered in 5.14 Sharing dashboards.

5.4 State-based UI

The application developed as part of the preliminary thesis for this project was created using React, a popular JavaScript library for creating reactive frontend interfaces. It enables the developer to create state-based UIs, where the UI automatically updates based on state changes, removing the need to manually specify imperative DOM operations. The digital shadow application prototype created for this project, was initially intended to have a very simple user interface, so I underestimated the convenience of using a third-party UI library.

During the course of the project, it became apparent that development could indeed benefit from being able to write declarative code which would translate to imperative DOM operations, as offered by the major JavaScript libraries for reactive interfaces.

To make it easy to compose and modify HTML structures, three utility functions were created. They will introduced briefly in the next sections.

5.4.1 The `compose` function

A utility called compose was created to make it easy to define HTML structures as JavaScript functions. The compose function takes three arguments: The tag name of the HTML element it should evaluate to, an object containing HTML properties to be set on the element, and an array of HTML nodes to be added as children of the created element.

For instance, the following JavaScript

```
compose('ul', {}, [
  compose('li', { innerText: 'List item 1' }, []),
  compose('li', { innerText: 'List item 2' }, []),
])
```

evaluate to the following HTML structure:

```
<ul>
  <li>List item 1</li>
  <li>List item 2</li>
</ul>
```

Alternatively, if the HTML properties should be dynamic, and update on application state changes, a function returning HTML properties can be passed as the second argument instead. In this case updated properties will be set every time the element’s update method is called.

```
compose('ul', {}, [
  compose('li', () => ({ innerText: getTextOfFirstItem() }), []),
   compose('li',  ()  =>  ({  innerText:  getTextOfSecondItem()  }),
[]),
])
```

The JavaScript above will evaluate to an unordered list where the innerText property of the list items will be updated every time the unordered list element's update method is called.

5.4.2 The `If` function

In some cases it is desirable to render some UI elements conditionally. For those cases a utility function called If was created. It takes two or three arguments: A function returning a condition, a function returning an HTML element to use if the condition is truthy, and optionally a function returning an HTML element to use if the condition is falsy. If the element's update method is called, the condition function is reevaluated, and the element is updated if necessary.

```
If(
  () => getProps().is3d,
  () => [Canvas3dWidget(getProps)],
  () => [Canvas2dWidget(getProps)]
)
```

The JavaScript above evaluates to a 3D widget element if the first parameter function returns a truthy value, and to a 2D widget element otherwise. If the element's update method is called, the condition function is reevaluated, and the element is replaced if necessary.

5.4.3 The `Each` function

There are also some cases where it is desirable to render a variable amount of HTML elements. For those cases a utility function called Each was created. It takes two arguments: A function returning an array of data which will be mapped to HTML elements, and a function returning a function used to map the data from the array returned from the first argument to HTML elements. If the element's update method is called, the current HTML elements are updated, superfluous elements are removed, and new elements are added.

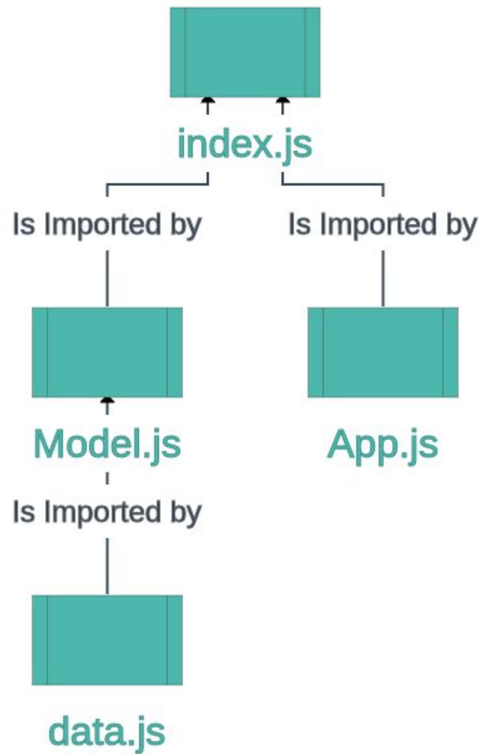
```
Each(
  () => getProps().state.widgets,
  (getCurrentValue) => [Widget(getCurrentValue)]
)
```

The JavaScript above maps a dynamic amount of widget data to a dynamic amount of widget elements which will be updated every time the element's update method is called.

Together, these three utility functions make it easy to create complex user interfaces without the need for a third-party library.

5.5 Persistent storage

Since the general-purpose digital shadow application was developed as a pure client-side application, it could not rely on storing data on a server. Instead, the browser's localStorage was used. The prototype only uses localStorage for storing digital shadow project data, i.e. names of created projects, URLs that the projects depends on, how frequently new data should be fetched, as well as user-submitted code. Having access to this data locally, rather than having to fetch it from a remote server will only have performance benefits for the time it takes to initially load the application and the time it takes to save updates to a project. It will not affect the performance of running user-submitted code in the client. Not storing project data centrally comes with the disadvantage of not being able to access projects created on other devices, but also comes with the benefit of users not having to authenticate themselves.



In the developed prototype, data.js exposes an interface to interact with localStorage. The application can easily be rewritten to store data remotely by modifying the functions in data.js.

5.6 Fetching API data

The functional requirements defined in Chapter 3 - Requirements and reasoning specify that it should be possible to fetch live data from REST APIs and ideally also WebSocket APIs. They also specify that it should be possible to use data from multiple API endpoints. In the developed prototype, it is only possible to fetch data using HTTP requests, making it possible to fetch data from REST APIs delivering data in JSON format. The prototype does support connecting to WebSocket APIs. Another limitation which conflicts with the functional requirements is that the prototype only lets the user specify a single API endpoint to connect to per project. The intention of the prototype was to be as simple as possible, while still implementing the features necessary to validate the feasibility of running user-submitted calculations in a DSL directly in the frontend, and as such, I considered these limitations to be acceptable.

The fact that the prototype was developed as a pure frontend application means that it is not well suited for storing or aggregating historical data. If data storage were to happen in the browser, the continuity of the stored data would depend on how continuously or frequently the application was used, and would vary between users. The size limitation of the browser's localStorage might also be a problem. This means that creating digital shadows which depend on having access to historical data would need to depend on the API endpoints they connect to, to provide such data.

5.7 Supporting dashboards

The functional requirements introduced in *Chapter 3 - Requirements and reasoning* specified that the general purpose digital shadow application should support the creation of live documentation, both in the form of dashboards and documents. However, to test the feasibility of running user-submitted code in the browser, only one of the use-cases had to be implemented. I prioritized letting the prototype enable the creation of live documentation in the form of dashboards. Live dashboards and live documents have much in common. They can both present visual and textual information, and as such, the distinction between them might not be clear. For this prototype, the main consequence of supporting live dashboards rather than live documents was that I would not add support for providing documentation in the form of lengthy text. Instead, dashboards would support displaying widgets containing short text, numerical values, 2D visualization and 3D visualizations. While this would not satisfy all the functional requirements defined in *Chapter 3 - Requirements and reasoning*, it would enable experiments with running multiple user-submitted calculations for different types of widgets in the same view.

5.8 Widgets

The preliminary thesis for this project covered the design and implementation of a sophisticated frontend module for creating and visualizing space frame structures. This module depends on a library called Three.js which utilizes WebGL for rendering. While it works well and is easy to use, it is limited to visualizing struts and nodes, and does not support creating surfaces, drawing arbitrary shapes or choosing colors. Since digital shadow use-cases can include both physical entities and abstract systems, I concluded that the space

frame visualization module would be too use-case-specific for a general purpose digital shadow application.

Since the application developed in this thesis aims to be less use-case-specific, I decided to implement a simpler, but more versatile visualization module using the HTML canvas API. This module would support drawing lines and surfaces of arbitrary color in both 2D and 3D.

One of the things which separates a general-purpose digital shadow application from many other data visualization and IoT applications is that it should not limit the way an entity can be presented to a predefined set of visualization and chart templates. To give users the freedom to visualize any entity for which one could possibly want to create a digital shadow, the implemented prototype gives users low-level control by letting them define widgets as lines and surfaces in either 2D or 3D space. This was achieved by implementing a simple, but versatile visualization module which, provided a data structure containing information about the lines and surfaces, would translate this into imperative drawing operations, using the methods provided by the HTML canvas API.

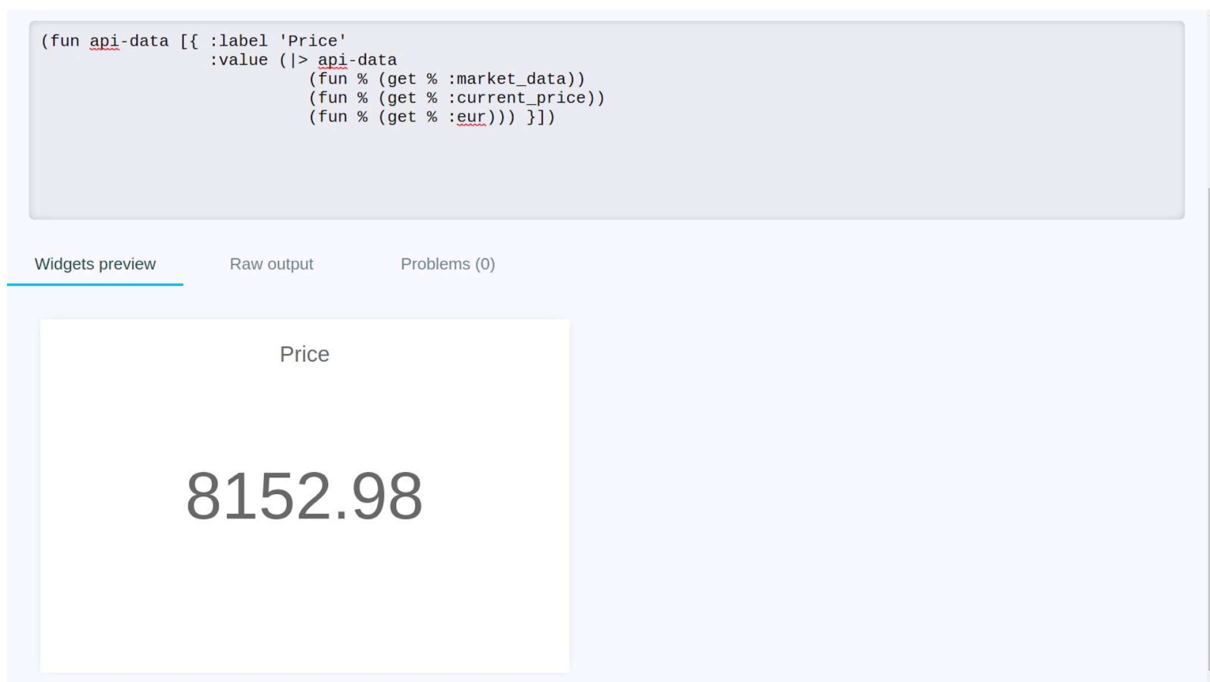
5.8.1 The widget schema

The visualization module supports drawing lines of arbitrary color and thickness, as well as surfaces of arbitrary color, in both 2D and 3D. The lines are defined by their start point and endpoint, and the surfaces are defined by the area enclosed by an arbitrary number of points. Each point is defined as an array of x, y and optionally z coordinates which can either be constants or values derived from continuously updated API data. When new data is fetched from the api, the visualizations will be rerendered to reflect the updated data.

```
type Point = number[] | {
  x: number;
  y: number;
  z?: number;
}

type Surface = {
  color?: string;
  points: Point[];
}

type Line = {
  color?: string;
  width?: string;
  points: Point[2];
}

type Widget = {
  label?: string;
  value?: number | string;
  is3d?: boolean;
  surfaces?: Surface[];
  lines?: Line[];
  center?: Point;
}
```

The above code shows the type definition for dashboard widgets as they would be written in TypeScript. Notice that a widget does not have to be either a value widget or a visualization widget. It is perfectly possible to display a value on top of a visualization. This comes with the advantage of the user not having to specify the widget type, or keep track of different schemas for different widget types.

While the goal of the dashboard widgets is to not impose limiting assumptions on the user, I recognized the need for being able to easily display important numerical values. The HTML canvas API does provide methods for rendering text as part of a visualization, by specifying text, color, font, font size, and position. However, while the visualization capabilities would benefit from exposing these methods as attributes to the user, I made the simplification of giving the user access to a property called value. Any non-nullish value supplied as the value property will be displayed nicely formatted in the middle of the widget. A label property is exposed as well. Any non-nullish value specified as the label property will be displayed at the top middle of the widget. While this makes some limiting assumptions about how the user might want to display numerical or textual values, it makes it possible to create nice-looking value widgets by simply providing a label and a numerical value, both of which can be defined either as constants or as a value calculated from API data.



The image above shows how a numerical widget can be created by assigning a numerical value to the widget object's value property.

5.8.2 Scaling and positioning visualizations

The user needs to somehow be able to decide where each point in a visualization will be rendered on the widget. A decision had to be made about whether the origin in the Euclidean

space would correspond to the center of the widget or the bottom left corner of the widget, for instance. Furthermore, there should be a predictable way to determine how much of the coordinate system would end up being rendered within the bounds of the widget.

In order to require as little configuration as possible from the user, the initial idea was to let the average of all of the coordinates used in the structure define the center point of the visualization, and position the visualization so that this point would appear at the center of the widget. Afterwards, the visualization would be scaled to just fit within the borders of the widget. This was the approach used to position and scale 3D structures in the space frame visualization module developed in the preliminary thesis for this project. While making it very easy to create visualizations without having to think about layout, automatic positioning and scaling turned out to be problematic for widgets whose points' average coordinate would vary as updated data arrived. Bar charts, for instance, whose height would change as new data arrived, would appear to have their width change, rather than their height, because they would be zoomed in and moved upwards when their height decreased.

To prevent the position of the visualization from changing in an uncontrollable manner upon arrival on new data, I decided to rather let the user specify a point in the Euclidean space which would correspond to the center of the widget. Depending on whether the user defines this point using static values or values derived from API data, the center can either be static or moved (albeit controllably) as new data arrives. If the user does not specify a point in the Euclidean space to appear in the center of the widget, (0, 0, 0) will be used by default.

The reason for letting the user define the center point of the visualization, rather than for instance the bottom left corner of the visualization is that for 3D widgets, there is no fixed point corresponding to the bottom left corner. What coordinate in the Euclidean space corresponds to a particular non-center point in the visualization depends on the "camera's" azimuth, polar angle, distance from the coordinate and focal length. The only point we can reliably define something around is the center point, which the "camera" in a 3D widget orbits around, and which will always be at the center of the widget, regardless of the recently mentioned "camera" parameters. As such, it makes sense to define which coordinates in the Euclidean space should correspond to the center of the visualization.

With regard to determining how much of the coordinate system would end up being rendered within the bounds of the widget, one approach was letting all coordinate values be percentages of the widget's height and width. This would give an intuitive understanding of where a particular point would end up. However, since the widgets were not developed to be square, a certain height interval would appear shorter than a width interval of the defined by the same size, which might make seemingly simple tasks like drawing a square challenging, especially if the aspect ratio of the widget was not known to the user.

Another approach was to define the size unit in all directions as one percent of just the widget width. This would make it easy to understand where a coordinate would end up horizontally, and would not result in stretched visualizations. However, the user would not intuitively be able to tell whether a certain height coordinate would fall outside of the widget, without knowing the widget's aspect ratio.

Yet another approach, which was the approach I decided to go for was to let the size units in the coordinate system correspond to one pixel in the widget. This approach requires that the user knows both the width and the height of the widget, which might seem like a step in the wrong direction compared to the previously mentioned approach where at least the horizontal placement of coordinates were intuitive. However, an advantage of letting one size unit correspond to one pixel is that it is made clear to the user just how precise details will appear once rendered. The user will not have to wonder whether a distance of one size unit will be enough to achieve a noticeable distance between two elements. Additionally, I had already decided that the most intuitive way to specify line width would be in pixels. As such, it would be convenient if a line of width 2 and length 2 ended up as wide as it was long. For 3D widgets, this is not entirely valid. The "camera" is not orthographic, meaning that the widget will not be isometric. This means that distances will appear smaller if they are located further away from the "camera". In these cases the line width is scaled similarly, but since I have not implemented any mechanism for gradually changing the width of lines, the width of the line is determined based on the average distance of its two endpoints from the camera.

Projects / Bascule bridge   Export

Data sources   Dashboard editor   Dashboard

One unit in the Euclidean space corresponds to 1px in the widget (for 3D widgets the resulting unit size will depend on the camera distance).

- Widget width: 480px
- Widget height: 320px

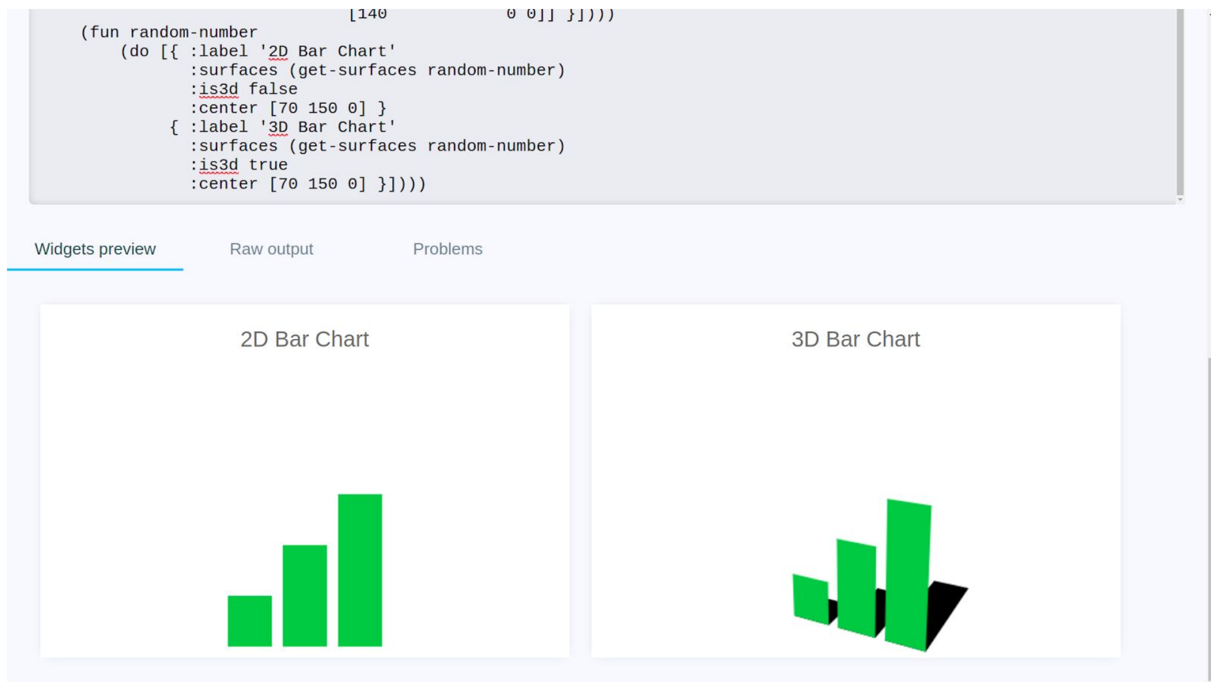
```
(do (define surfaces [{ :color 'rgba(0, 0, 255, 0.5)'
                        :points [[-25 0 -75] [25 0 -75] [25 0 75] [-25 0 75]] }
                      { :color 'rgba(0, 255, 0, 0.5)'
                        :points [[-25 0 75] [-25 0 -75] [-75 5 -75] [-75 5 75]] }
                      { :color 'rgba(0, 255, 0, 0.5)'
                        :points [[25 0 75] [25 0 -75] [75 5 -75] [75 5 75]] }])
    (define get-lines
            (fun rotation
                 (do [{ :color 'brown' :points [[-25 0 -5] [-35 0 -5]] }
                      { :color 'brown' :points [[-25 0  5] [-35 0  5]] }
                      { :color 'brown' :points [[-25 0 -5] [-25 0  5]] }
                      { :color 'brown' :points [[-35 0 -5] [-35 0  5]] }

                      { :color 'brown' :points [[-25 5 -5] [-35 5 -5]] }
                      { :color 'brown' :points [[-25 5  5] [-35 5  5]] }
                      { :color 'brown' :points [[-25 5 -5] [-25 5  5]] }
                      { :color 'brown' :points [[-35 5 -5] [-35 5  5]] }

                      { :color 'brown' :points [[-35 0 -5] [-35 5 -5]] }
                      { :color 'brown' :points [[-35 0  5] [-35 5  5]] }
                      { :color 'brown' :points [[-25 0 -5] [-25 5 -5]] }
                      { :color 'brown' :points [[-25 0  5] [-25 5  5]] }
```

To account for the issue of the user having to know the dimensions of the widget, the widget height and widget width were listed above the widget editor.

5.8.3 Distinguishing between 2D or 3D visualization widgets

The implemented prototype makes it possible to create dashboards with both 2D and 3D widgets. The same set of lines and surfaces can be visualized either as a 2D visualization widget, or a 3D visualization widget. The difference between 2D widgets and 3D widgets is that 3D widgets offer "camera" controls, so the view can be zoomed and rotated to see the visualized entity from different angles. To limit the amount of properties the user would have to set manually, the initial plan was to consider a widget to be a 3D visualization widget if at least one of the lines or surfaces defining the visualization had coordinates in three dimensions. This idea was abandoned, as it would make it cumbersome to show 3D data in 2D, since the depth information would have to be removed from the data in the function transforming the API response to an array of widget objects. While this is not particularly difficult, it limits the ability to reuse code across widgets with a different number of dimensions. Also, keeping the extra third dimension data in 2D widgets could be convenient if one would wish to change the visualization to 3D in the future.

Because of this, I ended up letting the dimensions of the widget be decided based on an is3d property in the widget object. Setting this to true will enable orbit controls and zoom, and setting it to false or not setting it at all will result in a 2D widget.



The image above shows two widgets created using the same surface data. The only difference (besides the labels) between the widgets is that the 2D Bar Chart widget has a falsy is3d property, while the 3D Bar Chart widget has a truthy is3d property.

5.8.4 Choice of coordinate system axis directions

In order to make it easier to utilize the same data for 2D and 3D visualizations, the z-axis was set to be perpendicular to the screen. This means that any 3D widget can easily be converted to a 2D widget by setting the is3d property to false, and any 2D widget can easily be converted to a 3D widget by setting the is3D property to true, without having to make changes to the order of the coordinates. Since coordinates on the axis perpendicular to the screen are optional, having the z-axis as the axis perpendicular to the screen also means that we adhere to good programming practice by placing optional parameters last.

5.9 Creating a basic 3D engine in JavaScript

In the created prototype, dashboards widgets are defined as functions of the data structure representing the widget. 3D widgets differ from 2D widgets in that they maintain internal "camera" state. This state keeps track of the following:

- The target of the widget, i.e. what point in 3D space the "camera" points at.
- The horizontal rotation (azimuth angle) of the camera.
- The vertical rotation (polar angle) of the camera.
- The distance of the camera from the defined center point in 3D space.
- The focal length of the camera.

The user can rotate the "camera" around entities visualized in 3D widgets by clicking the widget and dragging their mouse around. The user can move the "camera" closer or further away by scrolling while their cursor is over the widget.

Based on this "camera" state, all line and surface coordinates are transformed using the following function:

```
const transformPoint = pipe(
  toCenterOfStructure(center),
  rotateHorizontally(azimuthAngle),
  rotateVertically(polarAngle),
  toPerspective(d, focalLength),
  toCenterOfWidget(canvasWidth, canvasHeight)
);
```

Each of the functions the coordinate is piped through are presented below

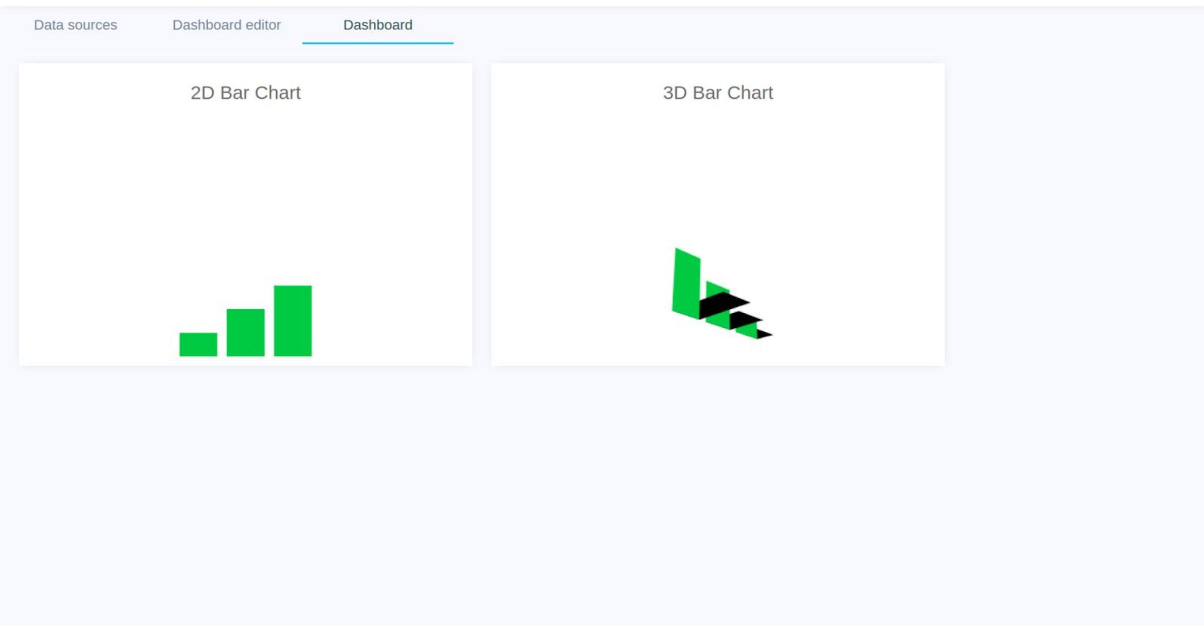
```
const toCenterOfStructure = ([centerX, centerY, centerZ]) => ([x,
y, z]) => [
  x - centerX,
  y - centerY,
  z - centerZ,
];
```

```
const rotateHorizontally = (azimuthAngle) => ([x, y, z]) => [
  x * Math.cos(azimuthAngle) + z * Math.sin(azimuthAngle),
  y,
  z * Math.cos(azimuthAngle) - x * Math.sin(azimuthAngle),
];
```

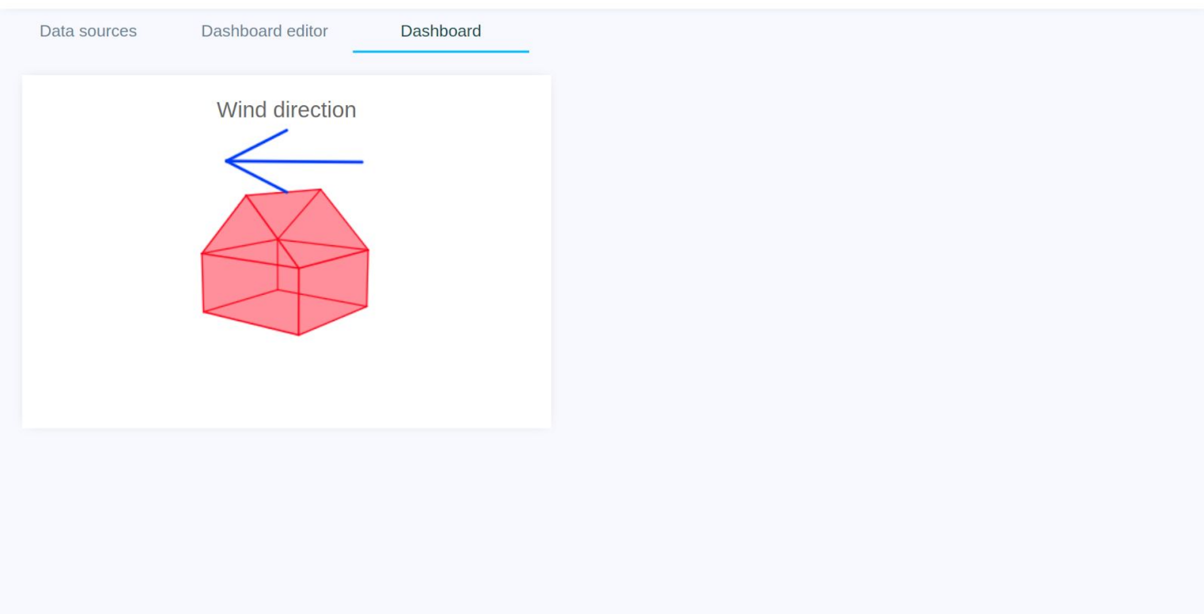
```
const rotateVertically = (polarAngle) => ([x, y, z]) => [
  x,
  y * Math.cos(polarAngle) - z * Math.sin(polarAngle),
  y * Math.sin(polarAngle) + z * Math.cos(polarAngle),
];
```

```
const toPerspective = (d, focalLength) => ([x, y, z]) => {
  if (!focalLength) return [x, y, z];
  return [
    x * (focalLength / (d + focalLength + z)),
    y * (focalLength / (d + focalLength + z)),
    z,
  ];
};
```

```
const toCenterOfWidget = (canvasWidth, canvasHeight) => ([x, y,
z]) => [
  x + canvasWidth / 2,
  y - canvasHeight / 2,
  z,
];
```



Effort has not been spent on determining which edges and surfaces lie in front of one another. All edges are drawn in front of all surfaces, edges are drawn in the order they are defined in the widget object and so are surfaces.



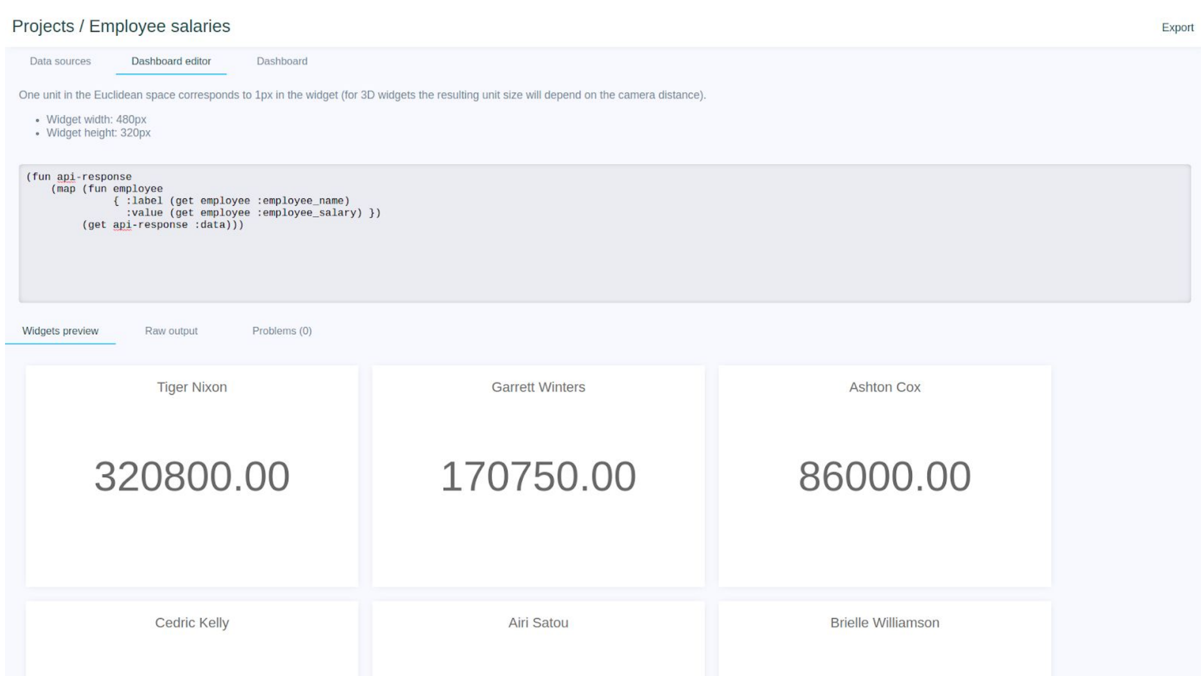
One can work around the issue of all lines being drawn in front of all surfaces and surfaces being drawn in the order they are defined by giving edges and surfaces the same hue, but letting the surfaces be transparent. Then the rendering order will not matter.

5.10 Giving the user freedom through a code interface

For the platform to be useful, it should enable the user to monitor anything which can be described by the data from an API. This means that the platform should be able to present any value which could be derived from the API data, as well as visualize it in a way which communicates the data intuitively.

In the developed prototype, this was accomplished by giving the user access to a code interface, residing in a view called *Edit Dashboard*. It could potentially also have been accomplished through a complex user-interface, resembling a visual programming language. For the prototype this was not prioritized, as it would be time-consuming to implement properly. Furthermore, the main purpose of the prototype was to test the feasibility of safely running user-submitted logic on the frontend a domain-specific language not native to the browser.

One of the functional requirements for the general-purpose digital shadow application was that it should be possible to let the number of widgets in a dashboard depend on the data fetched from the API. This has been achieved by giving the user access to only one text area for code input per dashboard, as opposed to giving access to one text area for code input per widget. In the prototype, the user creates a dashboard by defining a single function which returns an array of widget objects, thereby making it possible to let the amount of widgets in a dashboard be derived from the API data by writing a function which returns an array whose length is dependent on the widget data in some way.



Having to define all widgets in a dashboard as a single function might result in less readable code than if each widget were defined by their own isolated function. It does, however, enable the user to map dynamic data to an equally dynamic set of widgets.

5.11 Running user submitted code in the browser

5.11.1 Creating an interpreter for a domain-specific language

To make it possible to evaluate user-submitted code without giving it access to the browser API, an interpreter for a Lisp-like programming language was created. I will refer to this language as Digital Shadow Language.

The implementation of the interpreter is largely inspired by the interpreter for the programming language Egg, described in the book Eloquent JavaScript (43) by Marijn Haverbeke. Digital Shadow Language does, however, have a different syntax, supports more data structures, and implements a more extensive evaluator to make sure that a program written in Digital Shadow Language cannot access anything not defined within its own scope.

The syntax of Digital Shadow Language is very simple, consisting only of expressions and no statements. Each expression is either a value, or consist of multiple expressions, either as

parameters in a function application or as expressions in an array or object structure. This recursive structure of expressions makes it possible to parse and evaluate the language with a simple, similarly recursive interpreter. In addition to being easy to interpret, the Lisp-like syntax of Digital Shadow Language makes it easy to read and learn for people familiar with other Lisp dialects, like Common Lisp, Scheme or Clojure.

5.11.2 Parsing the program to an abstract syntax tree

The implemented interpreter for Digital Shadow Language consists of two modules: A parser and an evaluator. The parser is a function which takes the program as an argument in the form of a string, and returns an abstract syntax tree, representing the structure of the program.

For instance, consider the following program written in Digital Shadow Language:

```
(+ 4 (- 2 3))
```

This program will be parsed to the following abstract syntax tree:

```
{
  tokenType: 'apply',
  operator: { tokenType: 'word', word: '+' },
  args: [
    { tokenType: 'value', dataType: 'number', value: 4 },
    {
      tokenType: 'apply',
      operator: { tokenType: 'word', word: '-' },
      args: [
        { tokenType: 'value', dataType: 'number', value: 2 },
        { tokenType: 'value', dataType: 'number', value: 3 }
      ]
    }
  ]
}
```

Recall that Digital Shadow Language only consists of expressions. An expression can be a value, the name of a binding, or the application of a function or a special form like *if* or *define*.

A few features have been sacrificed in the pursuit of simplicity. For instance, any sequence of non-single-quote characters wrapped in single quotes, is considered a string, even if the single quotes are escaped with a backslash. Furthermore, in Digital Shadow Language there is no such thing as comments. Neither is there a concept of a null value.

```
const isString = (expression) => {
  if (expression.length < 2) return false;
  if (expression.charAt(0) !== "'") return false;
  if (expression.slice(-1) !== "'") return false;
  return true;
};
```

Any sequence of non-single-quote characters wrapped in single quotes, is considered a string in Digital Shadow Language.

```
const isNumber = (expression) => !isNaN(Number(expression));
```

In Digital Shadow Language, a number is any sequence of characters which can be interpreted as a number by JavaScript.

```
const isBoolean = (expression) =>
  expression === 'true' || expression === 'false';
```

In Digital Shadow Language, a boolean is any sequence of characters forming the words true or false.

```
const isArray = (expression) => {
  if (expression.length < 2) return false;
  if (expression.charAt(0) !== '[') return false;
  if (expression.slice(-1) !== ']') return false;
  return true;
};
```

In Digital Shadow Language, an array is any sequence of characters wrapped in square brackets. For an array to be valid, the sequence of characters wrapped in square brackets must consist only of valid expressions, separated by whitespaces.

```
const isKeyword = (token) =>
  token.charAt(0) === ':' && !token.slice(1).includes(':');
```

In Digital Shadow Language, a keyword is any sequence of characters that is not whitespace, does not have special meaning in the syntax and that starts with a colon.

```
const isObject = (expression) => {
  if (expression.length < 2) return false;
  if (expression.charAt(0) !== '{') return false;
  if (expression.slice(-1) !== '}') return false;
  return true;
};
```

In Digital Shadow Language, an object is any sequence of characters which are wrapped in curly brackets. For an object to be valid, the characters wrapped in the curly brackets must consist only of valid expressions. The number of expressions must be even. Furthermore, every expression at an even index (assuming that the first index is 0) must evaluate to a keyword.

```
const isWord = (expression) => true;
```

In Digital Shadow Language, a word (a binding name) is any sequence of characters that is not whitespace and does not have special meaning in the syntax.

```
const isApplication = (expression) => {
  if (expression.length < 2) return false;
  if (expression.charAt(0) !== '(') return false;
  if (expression.slice(-1) !== ')') return false;
  return true;
};
```

In Digital Shadow Language, an application is any sequence of characters which is wrapped in parentheses. For an application to be valid, the characters wrapped in the parentheses must consist only of valid expressions. The first expression must be a special construct, or evaluate to a function.

The syntax tree, which the parser returns, consists of a nested structure of expression objects. Each expression object has a tokenType property, as well as other properties which are specific to expression objects of that tokenType.

Expression objects whose tokenType is 'value' represent strings, numbers or booleans. They have a value property that contains their value, and a dataType property which indicates what the type of the value is. The dataType can be 'string', 'number' or 'boolean'.

```
{ tokenType: 'value', dataType: 'number', value: 42 }
```

The expression object above represents the number 42.

Expression objects whose tokenType is 'word' represent identifiers. They have a word property that contains the identifier's name as a string.

```
{ tokenType: 'word', word: 'digital-shadow-language' }
```

The expression object above represents the word digital-shadow-language.

Expression objects whose tokenType is 'keyword' represent keywords. They have a value property that contains the string representation of the keyword.

```
{ tokenType: 'keyword', value: ':digital-shadow-language' }
```

The expression object above represents the keyword :digital-shadow-language.

Expression objects whose tokenType is 'array' represent arrays. They have a values property that contains an array of expression objects.

```
{
  tokenType: 'array',
  values: [
    { tokenType: 'value', dataType: 'number', value: 1 },
    { tokenType: 'value', dataType: 'number', value: 2 },
    { tokenType: 'value', dataType: 'number', value: 3 },
    { tokenType: 'value', dataType: 'number', value: 4 }
  ]
}
```

The expression object above represents the array [1 2 3 4].

Expression objects whose tokenType is 'object' represent objects. They have a keywords property, that contains an array of keyword expression objects, and a values property, that contains an array of expression objects. As such, objects in Digital Shadow Language are not hashmaps, meaning that value access happens in linear time, rather than constant time.

```
{
  tokenType: 'object',
  keywords: [
    { tokenType: 'keyword', value: ':a' },
    { tokenType: 'keyword', value: ':b' },
    { tokenType: 'keyword', value: ':c' }
  ],
  values: [
    { tokenType: 'value', dataType: 'number', value: 1 },
    { tokenType: 'value', dataType: 'number', value: 2 },
    { tokenType: 'value', dataType: 'number', value: 3 }
  ]
}
```

The expression object above represents the object { :a 1 :b 2 :c 3 }.

Lastly, expression objects whose tokenType is 'apply' represent applications of functions or special forms. They have an operator property that contains an expression object which evaluates to a function or a special form. They also have an arguments property that contains an array of expression objects which will be passed as arguments.

```
{
  tokenType: 'apply',
  operator: { tokenType: 'word', word: '+' },
  args: [
    { tokenType: 'value', dataType: 'number', value: 1 },
    { tokenType: 'value', dataType: 'number', value: 2 }
  ]
}
```

The expression object above represents the function application (+ 1 2).

5.11.3 Creating the parser

Since Digital Shadow Language is recursive in its nature, consisting of expressions which in turn may be applications of other expressions, the parser could be kept relatively simple by adapting a similarly recursive nature.

A function called parseExpression was created to parse expressions. An expression is either an application, an array, an object, a string, a number, a boolean, a keyword, or a word.

If the expression is an application, the content wrapped in the parentheses are tokenized using a function called getTokens. The getTokens function takes a string as an argument which contains one or more expressions, and returns an array of the these expressions.

Each of the expressions, or tokens returned from the getTokens function are then parsed to expression objects by recursively using the parseExpression function. An expression object with 'apply' as the tokenType property, the first expression object as the operator property and the remaining expression objects as the arguments property is returned.

If the expression is an array, the content wrapped in the square brackets is tokenized using the getTokens function and parsed to expression objects by recursively using the parseExpression

function. An expression object with 'array' as the tokenType property and the expression objects as the values property is returned.

If the expression is an object, the content wrapped in the curly brackets are tokenized using the getTokens function and parsed to expression objects by recursively using the parseExpression function. An expression object with 'object' as the tokenType property, the expression objects with even indices (assuming that the first index is 0) as the keywords property and the expression objects with the odd indices as the values property is returned.

If the expression is a string, an expression object with 'value' as the tokenType property, 'string' as the dataType property, and the expression string as the value property is returned.

If the expression is a number, an expression object with 'value' as the tokenType property, 'number' as the dataType property, and the expression string parsed as a number using the JavaScript Number function as the value property is returned.

If the expression is a boolean, an expression object with 'value' as the tokenType property, 'boolean' as the dataType property, and true or false depending on whether the expression is 'true' or 'false' respectively as the value property is returned.

5.11.4 Evaluating the abstract syntax tree

Once the Digital Shadow Language code has been parsed to an abstract syntax tree, it can be evaluated. During evaluation, words will be evaluated to the value they are bound to and functions and special forms will be applied to the arguments they are called with. This is done recursively until no expression objects with tokenType 'application' are left.

To evaluate the syntax tree, a function called evaluateSyntaxTree was defined. It has separate evaluation logic for each of the different expression object tokenTypes:

'value'

Calling evaluateSyntaxTree with an expression object whose tokenType is 'value' or 'keyword' simply returns the expression object.

'array'

Calling evaluateSyntaxTree with an expression object whose tokenType is 'array' or 'object' returns the expression object, but with any keywords or values properties recursively evaluated using the evaluateSyntaxTree function.

'word'

Calling evaluateSyntaxTree with an expression object whose tokenType is 'word' will return the expression object in the scope bound to that word, assuming it exists in the scope. Otherwise, it will throw an error.

'apply'

The operator in an application expression object can either evaluate to a function or a special form. If it evaluates to a function, the parameters are evaluated using the evaluateSyntaxTree function and the evaluated operator is applied to the arguments, which are first evaluated using the evaluateSyntaxTree function. If, however, the expression object evaluates to a special form, the arguments should not be evaluated before applying the special form. In the case of the special form *if*, for instance, only one of the arguments should be evaluated. Which argument should be evaluated depends on what the condition argument evaluates to. Another example is the special form *fun* which is used to create functions. The body of the function should not be evaluated until the function is called. Otherwise, the function body might reference parameters which are not yet bound to values. Because of this, special forms should be called with unevaluated arguments, as well as the scope.

5.11.5 Special forms

Special forms in Digital Shadow Language are expressions which are processed by their own rules and not similar to how regular functions are processed. The special forms implemented in Digital Shadow Language are *define*, *do*, *fun* and *if*.

define

define is a special form used to create bindings. It does so by assigning word expression objects to the local scope. *define* takes two parameters: A word and an expression that will be bound to that word in the local scope. When *define* is applied, it evaluates to the expression being bound to the word.

do

do is a special form which makes it possible to evaluate expressions consecutively within the same local scope. Without the *do* function, the *define* function would be of little value, since it would not be possible to access defined words, since the scope where they were defined would not be accessible for other expressions. Thanks to *do*, it is possible to define bindings in a local scope and access those in consecutive expressions. When *do* is applied, it evaluates to the last expression being passed to it.

```
(do (define a 2)
    (define b (+ a 2))
     b)
```

The code above defines a binding called a and assigns the value 2 to it. Next, a binding called b is defined, and assigned the value a + 2, which is 4. Finally, b is the final argument, and therefore it is b's value which will be returned by the do function.

fun

fun is a special form which makes it possible to create functions. All arguments passed to *fun* except for the last, are words which will act as function parameters. The last argument passed to *fun* will be considered the function body. The function body will not be evaluated until the function is applied, at which point the arguments the function is called with will be assigned to the word parameters. The function body has its own local scope, and the function parameters will be assigned to this local scope, making them accessible from within the function body.

```
((fun a (* 2 a)) 8)
```

In the code above, an anonymous function which returns the argument it is called with multiplied by 2, is called with the value 8. The whole expression evaluates to 16.

```
(do (define sum (fun a b (+ a b)))
    (sum 1 2))
```

In the code above, a function which adds two values together is assigned to the word sum. Next, the sum function is applied to the numbers 1 and 2. The whole expression evaluates to 3.

if

if is a special form which expects three arguments. The first argument must evaluate to an expression object whose value is either true or false. Otherwise, an error will be thrown. If the first argument evaluates to an expression object whose value is true, the second argument will be evaluated and returned. If the first argument evaluates to false, the third argument will be evaluated and returned.

```
(do (define count-to-ten
           (fun x
               (if (< x 10)
                   (count-to-ten (+ x 1))
                    x)))
    (count-to-ten 0))
```

In the code above, the if function is used to create a recursive function which returns its argument if it is not less than 10, and calls itself with the argument + 1 if it is less than 10. The function is assigned to the word count-to-ten, and then called with the number 0. The whole expression evaluates to 10.

5.11.6 Core library

To facilitate fundamental operations, an extensive core library was developed, constituting the initial scope of the script evaluation. The functions will not be explained in detail, but code examples showcasing both special forms and core functions in Digital Shadow Language can be found in *Appendix B - Digital Shadow Language Examples*.

5.11.7 Converting the evaluated syntax tree to JavaScript

The result from the evaluateSyntaxTree function is an expression object for the resulting non-application expression object. The reason why it evaluates the syntax tree to an

expression object, rather than directly to a JavaScript value is because of JavaScript's object-oriented nature.

Digital Shadow Language supports non-primitive data types, such as objects and arrays, as well as a function called *get* which is used to access properties of these objects. For Digital Shadow Language to not reintroduce all the security vulnerabilities of JavaScript, it is important to not translate expressions to their JavaScript equivalents too early in the evaluation process. One issue which may arise is if a sub-syntax-tree which evaluates to a function is evaluated to a JavaScript object before its (nonexistent) values are attempted accessed with the get function. In JavaScript, functions are first-class objects, so from the *get* function's perspective, it is difficult to know if the object it is accessing values from should indeed be considered an object in Digital Shadow Language, or if it should rather be considered a function. This means that unless we restrict accessing certain properties, the *get* function could be exploited to access the constructor property of a JavaScript function, for instance, which makes it possible to create a JavaScript function by providing a function body as a string to the constructor. This function would have access to the global scope of the page. Allowing this would defeat Digital Shadow Language's main purpose, which is to prevent user-submitted code from accessing the global scope. To prevent this, the program is first parsed and evaluated into an expression object and then converted to a JavaScript value. This ensures that no function application happens after the expression objects have been evaluated to JavaScript, which in turn ensures that only properties defined within the program being interpreted can be accessed.

To convert the expression object which is returned from the evaluateSyntaxTree function to a JavaScript value, I created a function called convertToJavaScriptValue. It takes an expression object as an argument and returns a string, a number, a boolean, an array or an object.

Calling convertToJavaScriptValue with an expression object whose tokenType is 'value' returns the value of the expression object's value property.

Calling convertToJavaScriptValue with an expression object whose tokenType is 'array' or 'object' returns the expression object, but with any keywords or values properties recursively evaluated using the convertToJavaScriptValue function.

Calling convertToJavaScriptValue with an expression object whose tokenType is 'keyword' returns the keyword as a string, including the colon.

```
const evaluate = (expression) =>
 convertToJavaScriptValue(
   evaluateSyntaxTree(parseExpression(expression), { ...core })
 );
```

A function called evaluate was created to parse a program to a syntax tree, and evaluate the syntax tree to a value, with access to the functions included in the core library.

5.11.8 Making it impossible to escape interpreter scope

The previous section has already covered the importance of not converting expression objects to JavaScript before all function and special form applications are done, as it could potentially enable access to unwanted JavaScript properties. A similar problem as having evaluated code to JavaScript too soon occurs when we wish to apply a function written in Digital Shadow Language to API data in the general-purpose digital shadow prototype. In this case the API data is in the form of a JavaScript expression, typically an object. The user-submitted code should only be able to access properties containing applicable data. For instance, the object's toString function should not be accessible. To accomplish this, a function called callWithJavaScriptArguments was created. The callWithJavaScript function creates an application expression object where the operator is the evaluated syntax tree of the user-submitted code, and the arguments is an array consisting of the evaluated syntax tree of the API data, obtained by calling the convertFromJavaScriptValue function with the data from the API. In Digital Shadow Language, it is only possible to access a particular property of an object if the key is listed in the keys property in the object's expression object. The convertFromJavaScriptValue function ensures that the syntax tree created from the data passed to the Digital Shadow Language function only adapts properties with non-applicable values. This application expression object is then evaluated using the evaluateSyntaxTree

function and then converted to JavaScript using the convertToJavaScript function. The convertToJavaScript function also ensures that only non-callable values or structures, and not functions are returned, since widgets only need non-callable properties. Together, these measures ensure that only keywords, defined in the user-submitted code, or which are primitive values on an object the user-submitted code is applied to can be accessed by the user-submitted code.

5.11.9 Preventing denial of service

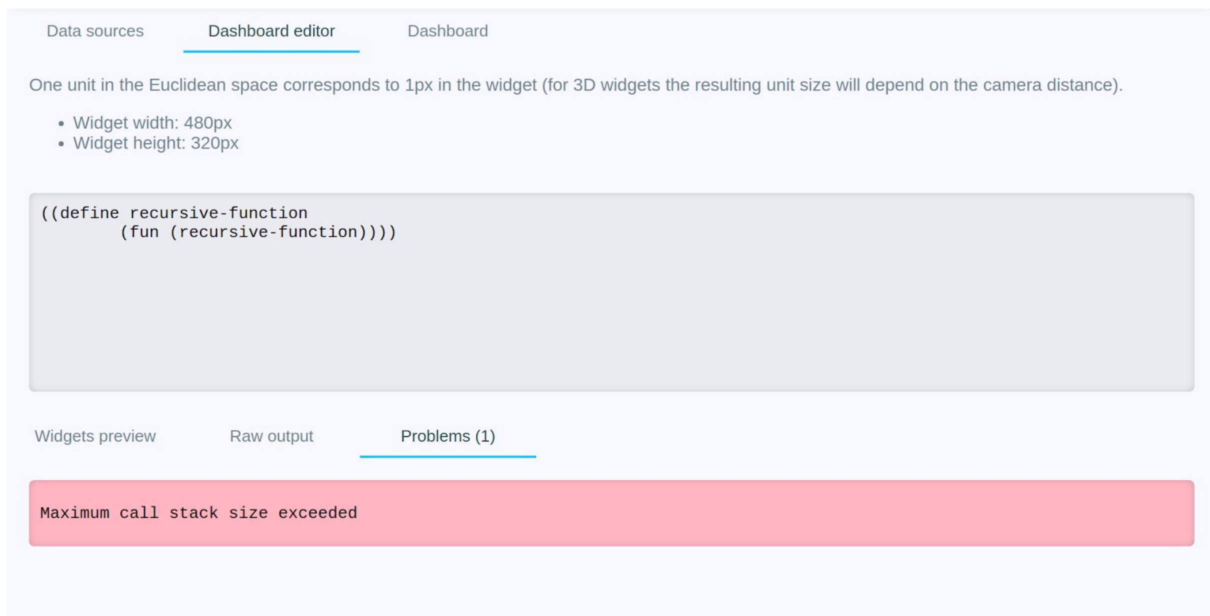
Because of JavaScript's single-threaded, blocking nature, long-lasting or never-ending user-submitted calculations pose a denial of service risk. While waiting for a script to terminate, all other user interaction with the page, will be blocked, temporarily rendering the page unusable. If dashboards were stored centrally and could be shared with other users within the application, this could be used as an attack vector to freeze other users' application. The Digital Shadow Language interpreter should account for this. It should also account for users who accidentally write long-lasting or never-ending scripts.

In Digital Shadow Language, there is no special form for creating loops, so the only way to cause a never-ending script is through recursion, i.e. a function which calls itself infinitely.

```
((define recursive-function
      (fun (recursive-function))))
```

The code above calls a recursive function which calls itself, causing an infinite loop.

Infinite recursion itself is not necessarily a problem, as JavaScript automatically aborts script execution if the maximum call stack size is exceeded.



The infinite recursion script is quickly aborted as soon as it exceeds the maximum call stack size. In this case, the infinite recursion does not result in denial of service.

Denial of service can still be accomplished by writing long-running scripts which don't quickly exceed their maximum call stack size.

```
(do (define get-nth-fibonacci
          (fun n
              (if (= n 1)
                  0
                  (if (= n 2)
                      1
                      (+ (get-nth-fibonacci (- n 1))
                         (get-nth-fibonacci (- n 2)))))))
    (get-nth-fibonacci 45))
```

The code above, which calculates the 45th Fibonacci number would block all other script execution in the application for about 10 000 seconds.

To prevent scripts from running for an unreasonable amount of time, the *evaluateSyntaxTree* function was extended to accept a *maximumNumberOfApplications* parameter. It also maintains internal state of how many applications have been performed. If the number of performed applications exceeds the the value of *maximumNumberOfApplications*, the evaluation is aborted by throwing an error.

The UI of the prototype does not let the user manually set the maximum number of applications. Instead, all user-submitted code which has not terminated after 200 000 applications will be aborted. Based on empirical observations, this corresponds to between one and two seconds, depending on the computer. If the user's use-case requires heavy calculations that the user is willing to wait for, it would be desirable to be let the user manually specify the maximum number of applications, or perhaps the maximum time the script would be allowed to run before being aborted.

Projects / Fibonacci   Export

Data sources   Dashboard editor   Dashboard

One unit in the Euclidean space corresponds to 1px in the widget (for 3D widgets the resulting unit size will depend on the camera distance).

- Widget width: 480px
- Widget height: 320px

```
(do (define get-nth-fibonacci
            (fun n
                 (if (= n 1)
                     0
                     (if (= n 2)
                         1
                         (+ (get-nth-fibonacci (- n 1))
                            (get-nth-fibonacci (- n 2)))))))
    (get-nth-fibonacci 45))
```

Widgets preview   Raw output   Problems (1)

```
Script ran for too long. Aborted to prevent denial of service.
```

User-submitted scripts are aborted if they have not terminated after 200 000 function applications.

5.12 Challenges with writing widgets code

During initial testing, a number of challenges with writing valid widgets code presented themselves, often resulting in errors.

Errors which are difficult to discover even when familiar with the widget schema

Even when familiar with the expected schema, it was difficult to discover mistakes, like having forgotten to specify an empty array for lines when the visualization should only contain surfaces, or not remembering to specify the width of a line.

Non-error which still result in nothing being rendered

Another issue was not defining a center point for the visualized entity, or setting it to incorrect values. This could lead to the visualization not being rendered within the canvas, causing nothing to be rendered at all. This is an example of an issue which could arise even when following the widget schema perfectly.

Errors caused by schema not matching user intuition

Even if users are familiar with the overall required widgets schema, they might make the mistake of specifying a point as an object with x, y and z properties instead of an array with an x, y and z values.

Errors cause by syntax errors in the widgets code

Errors which occur during parsing or evaluation of the submitted code are particularly difficult to debug, as a single error will cause no widgets to be displayed, since all the widgets in a single dashboard are defined by the same function.

The above challenges were largely approached by making the widgets' schema more flexible with optional values and alternative ways to specify values. Additionally, a UI component was created to show errors thrown during parsing or evaluation, as well as helpful messages to help the user understand why the widget might not display as expected.

5.13 Helping the user write valid widgets code

The challenges mentioned in *5.12 Challenges with writing widgets code* are challenges which existed during the early stages of the project and were since improved upon. The following paragraphs explain how the final prototype supports the user in the process of writing valid widgets code.

5.13.1 Optional values with sensible defaults

For a widget to display a visualization in 3D, it is not required that a z-coordinate is specified for all of the points used to define the lines and surfaces of the visualization. Omitted coordinates will default to zero.

While lines should be specified as an array of line objects, it is not necessary to specify an empty array of lines if the visualization does not contain any lines. Similarly, it is not necessary to specify an empty array of surfaces if the visualization does not contain any surfaces.

5.13.2 Helpful error messages

If an error occurs while interpreting the user-submitted code, for instance if the syntax is invalid, or the code attempts to access variables which are not defined, these errors will be displayed in the UI in a separate *Problems* tab in the dashboard editor. The error messages of the errors thrown by the interpreter are designed to be easily understandable. Even if the user-submitted code can be interpreted successfully, there might be other mistakes which result in no widgets showing. The user-submitted function defining the dashboard may, for instance, return a widget object instead of an array of widget objects. This mistake will also be explained in the *Problems* tab.

```
                    { :color 'brown' :points [[32.5 30 -2.5] [75 5 -2.5]] }
                    { :color 'brown' :points [[32.5 30  2.5] [75 5  2.5]] }
                    { :color 'brown' :points [[53.75 17.5 -2.5] [53.75 5 -2.5]] }
                    { :color 'brown' :points [[53.75 17.5  2.5] [53.75 5  2.5]] }
                    { :color 'brown' :points [[64.375 11.25 -2.5] [64.375 5 -2.5]] }
                    { :color 'brown' :points [[64.375 11.25  2.5] [64.375 5  2.5]] }
                    { :color 'brown' :points [[43.125 23.75 -2.5] [43.125 5 -2.5]] }
                    { :color 'brown' :points [[43.125 23.75  2.5] [43.125 5  2.5]] }

                    { :color 'brown' :points [[-32.5 35 -2.5] [32.5 35 -2.5]] }
                    { :color 'brown' :points [[-32.5 35  2.5] [32.5 35  2.5]] }
                    { :color 'brown' :points [[-32.5 30 -2.5] [32.5 30 -2.5]] }
                    { :color 'brown' :points [[-32.5 30  2.5] [32.5 30  2.5]] }
                ])))
  (define center [0 25 0])
  (fun rotation-deg (do (define rotation-rad (* rotation-deg (/ 3.14 180)))
                      [{ :label 'Bascule bridge (3D)'
                         :is3d true
                         :surfaces surfaces
                         :lines (get-lines rotation-rad)
                         :center center }
                       { :label 'Span angle (degrees)'
                         :value rotation-deg }
                       { :label 'Span distance (m)'
                         :value (* 2 (* 25 (- 1 (cos rotationRad)))) }])))
```

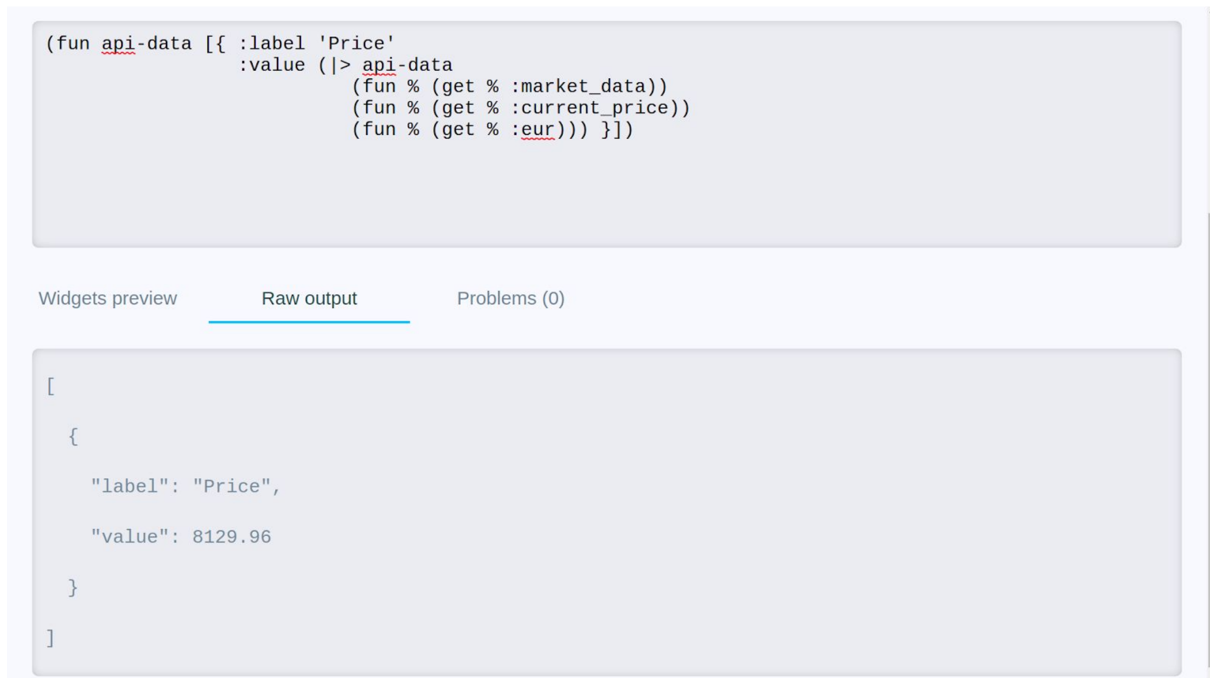
Widgets preview   Raw output   Problems (1)

```
rotationRad is not defined.
```

If the user-submitted cannot be successfully evaluated, the user will be informed about what is wrong, making it easier to fix mistakes.

5.13.3 Displaying the resulting JavaScript structure

If the calculation is complex, being able to see a stringified JSON version of the calculation can make small mistakes much more evident, so the user does not have to analyze the code meticulously to be able to figure out what's wrong.



In the Raw output tab in the Edit dashboard view, the result from applying the user-submitted function to the data from the API can be visualized, making it easier for users to validate if their function actually returns the data they think it does.

5.14 Sharing dashboards

The user should be able to easily share dashboards created with the application, with others. For instance, the user might want to create a dashboard which serves as living documentation of some entity, and embed it on their website, or they might want to send the dashboard to a coworker, or store it as a document.

If the application was not developed as a pure frontend application, but was rather hosted centrally, for instance in the form of a SaaS application, logic could be implemented to make it possible to share dashboards with other users of the same application, within the application.

One of the research questions defined in *Chapter 1 - Introduction* is about whether a general-purpose digital shadow platform can work with no backend at all, and what the limitations of doing this would be. Since sharing data between users typically depends on a backend to store the shared data, satisfying the functional requirement of being able to share created dashboards is particularly interesting.

The implemented prototype solves this problem by making it possible to export a created dashboard to a single HTML file which, when running in a browser, will continue to fetch data from the selected API endpoint at the specified intervals. This means that live, exported dashboards can be embedded within a web page or shared as a file, by email, for instance.

An exported dashboard would need to reuse much of the logic from the frontend application in which it was created. To be able to reuse the logic, dashboard exports are handled by exporting a modified version of the application's source code, keeping the API and visualization logic, but modifying the code to use different functions for accessing project data. Because the creation of the exported dashboard will happen at runtime rather than at compile-time, exporting a modified version of the application's source code could be challenging if the prototype was created using a framework or library written in languages

which are not valid JavaScript until being compiled, or if the source code would be minified during compilation.

To be easy to run, the prototype was already configured to be compiled to a single HTML file with all scripts and styles inlined in the same document, as opposed to being bundled as several files which would have to be fetched and inlined at runtime. Since the main application already had all the JavaScript needed to render the page residing within the document's HEAD-tags, the code could easily be accessed as a string through document.head.innerHTML.

In the source code of the prototype, a constant called isExported was defined and is set to false in the unexported application.

```
const isExported = false;
```

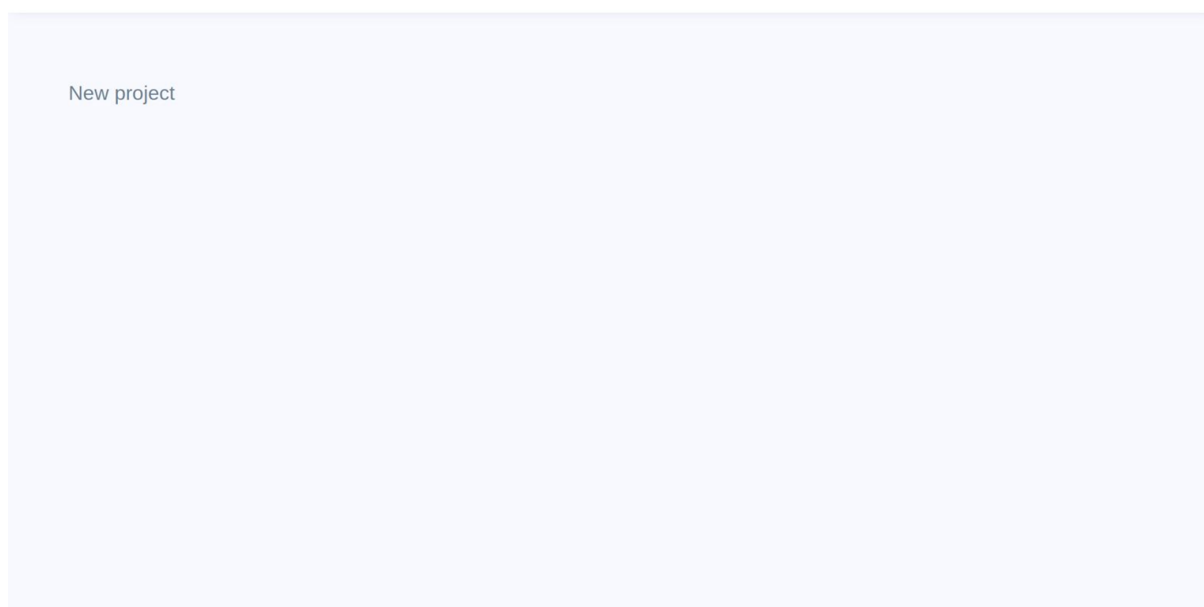
When exporting a dashboard, in the string representing the page's source code, const isExported = false; is replaced with const isExported = true;. Additionally, a function called getExportedProject which returns the current project state is inserted into the string representation of the code and then exported to an HTML file. When opening the exported application, the isExported flag indicates that data should not be loaded from localStorage, but rather from the getExportedProject function. When isExported is true, routing will be restricted as well, as mentioned in *5.3 Routing*.

6 Results

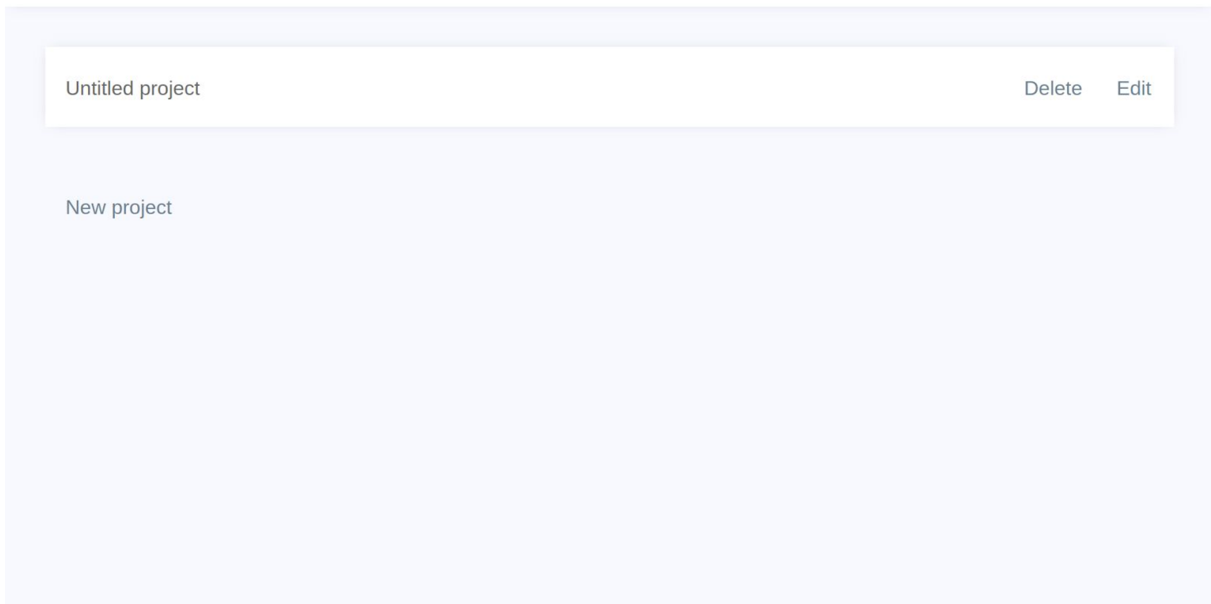
6.1 Overview

This chapter gives a detailed walkthrough of all of the features of the general-purpose digital shadow application prototype whose development was covered in *Chapter 5 - Implementation*, by creating a digital shadow of a bascule bridge, based on random data from the RANDOM.ORG HTTP interface.

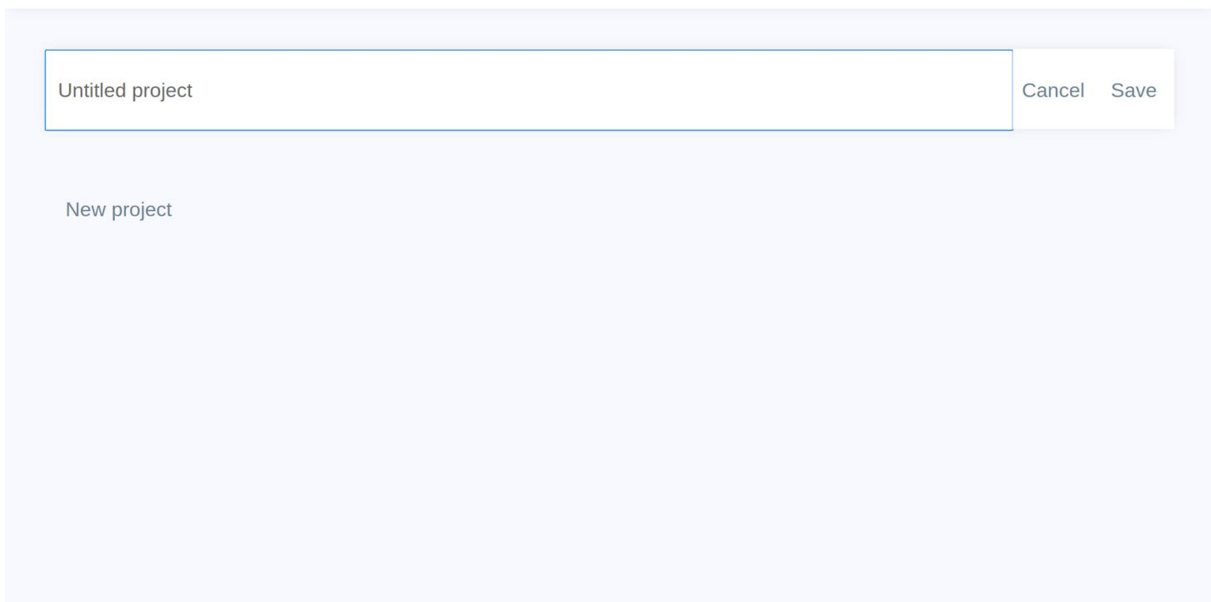
6.2 Walkthrough of the developed prototype



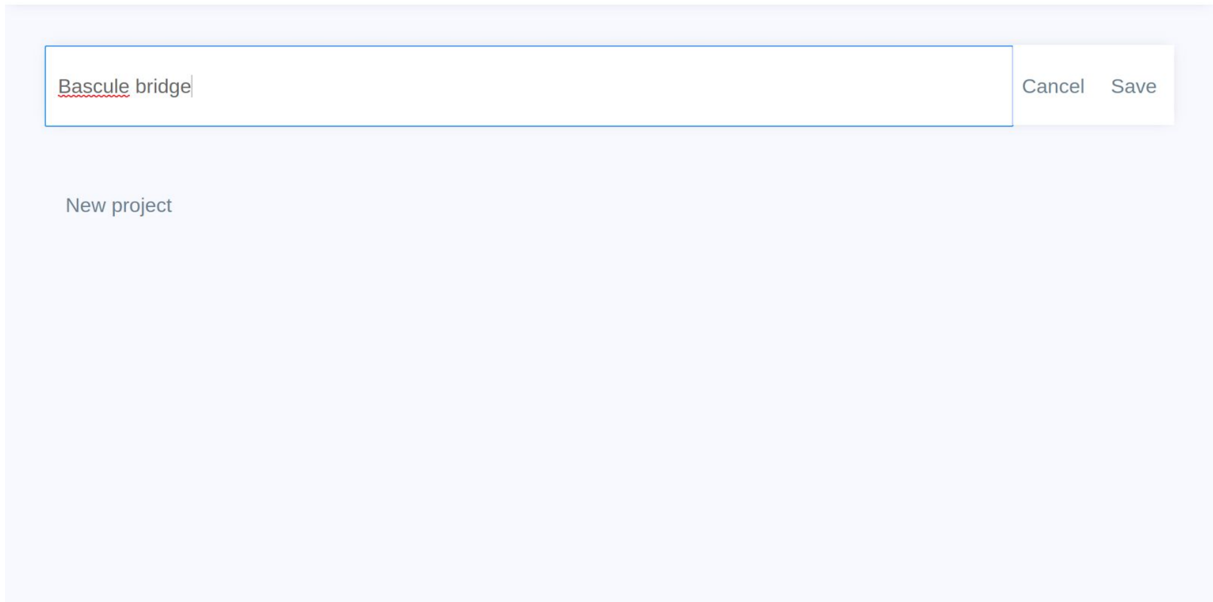
*This is the Projects overview before any projects have been created. To create a new project, we can click **New project**.*



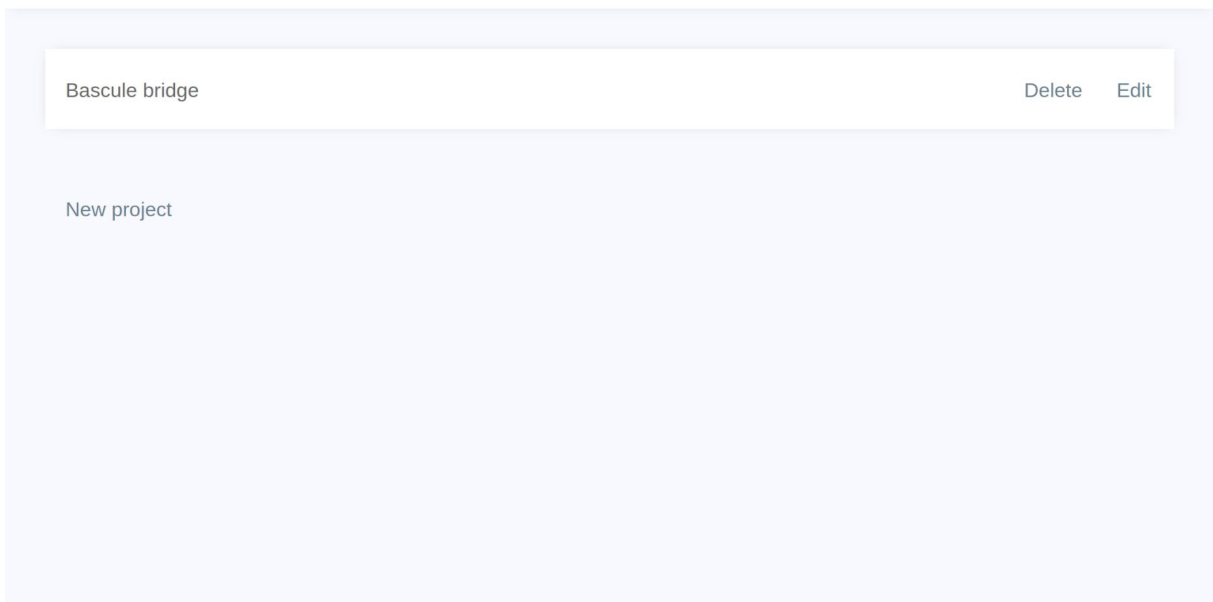
*Clicking **New project** will create a new project called **Untitled project**. To edit the name of the project, we can click **Edit**.*



*After clicking **Edit**, a new project name can be entered.*



In this example we wish to create a model of a bascule bridge, so we rename the project to ***Bascule bridge*** *and click* ***save****.*



The project now has a new name. To open the project, we can click the white box representing the project.

Projects / Bascule bridge

Export

Data sources

Dashboard editor

Dashboard

API URL

Test API URL

Fetch interval (s) 0

*Opening the project will take us to the **Data sources** view of the project. The **Data sources** view is divided into three sections: An input field for entering an API URL, a preview section for previewing data from the API and an input field for entering a fetch interval to determine how often new data should be fetched from the selected data source.*

Projects / Bascule bridge

Export

Data sources

Dashboard editor

Dashboard

API URL :tps://www.random.org/integers/?num=1&min=1&max=90&col=1&base=10&format=plain&rnd=new

Test API URL

Fetch interval (s) 0

In this example, we enter the URL https://www.random.org/integers/?num=1&min=1&max=90&col=1&base=10&format=plai

n&rnd=new, using the RANDOM.ORG HTTP Interface to get a random number in the range 1 up to and including 90, which will be used to represent the angle of the spans of the bascule bridge.

Projects / Bascule bridge
Export

Data sources | Dashboard editor | Dashboard

API URL `https://www.random.org/integers/?num=1&min=1&max=90&col=1&base=10&format=plain&rnd=ne`

Test API URL

Fetch interval (s) `10`

I set the fetch Interval to 10 seconds, meaning that a new random value will be fetched every 10 seconds.

Projects / Bascule bridge
Export

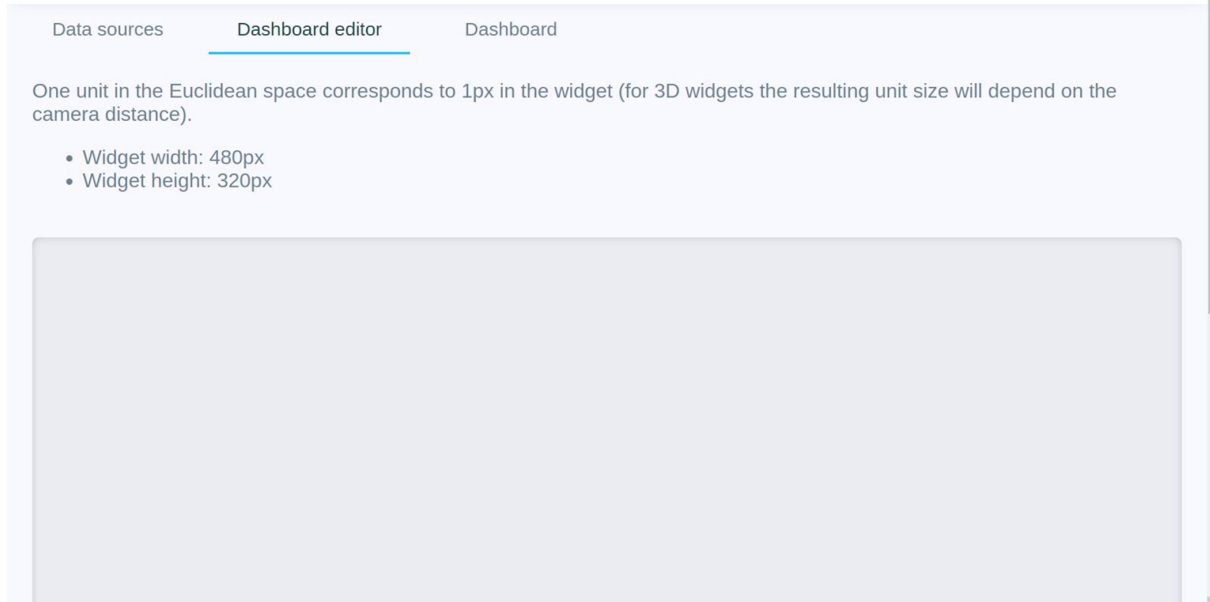
Data sources | Dashboard editor | Dashboard

API URL `https://www.random.org/integers/?num=1&min=1&max=90&col=1&base=10&format=plain&rnd=ne`

Test API URL 26

Fetch interval (s) `10`

*Clicking **Test API URL** shows that the API returns the number 20, which is as expected.*



*Once we have configured the data sources, we can navigate to the **Dashboard editor tab** where we have access to a text area where we can write widgets code.*

For this example, we will create a 3D visualization of the bascule bridge for which we are making the digital shadow. we will also create a numerical value widget showing the span angle, as returned from the API. Lastly we will create one widget showing the distance between the two spans of the bascule bridge. This is a value derived from the value fetched from the API, assuming that we know that each span is 25 meters long. The code used to define the widgets is a function which, when called with the bridge span angle, returns three widget objects. The complete function can be found in *Appendix A - Dashboard Code Example from chapter 6*.

Projects / Bascule bridge

Export

Data sources | Dashboard editor | Dashboard

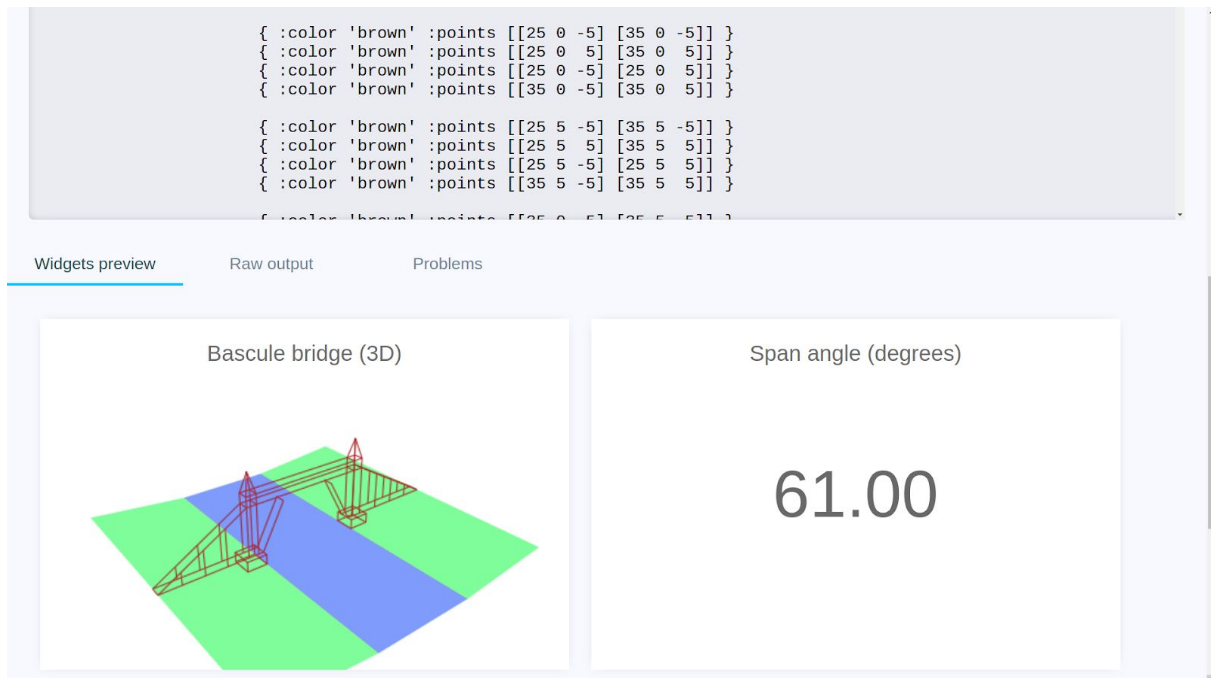
One unit in the Euclidean space corresponds to 1px in the widget (for 3D widgets the resulting unit size will depend on the camera distance).

- Widget width: 480px
- Widget height: 320px

```
(do (define surfaces [{ :color 'rgba(0, 0, 255, 0.5)'
                        :points [[-25 0 -75] [25 0 -75] [25 0 75] [-25 0 75]] }
                      { :color 'rgba(0, 255, 0, 0.5)'
                        :points [[-25 0 75] [-25 0 -75] [-75 5 -75] [-75 5 75]] }
                      { :color 'rgba(0, 255, 0, 0.5)'
                        :points [[25 0 75] [25 0 -75] [75 5 -75] [75 5 75]] }])
    (define get-lines
        (fun rotation
            (do [{ :color 'brown' :points [[-25 0 -5] [-35 0 -5]] }
                 { :color 'brown' :points [[-25 0  5] [-35 0  5]] }
                 { :color 'brown' :points [[-25 0 -5] [-25 0  5]] }
                 { :color 'brown' :points [[-35 0 -5] [-35 0  5]] }

                 { :color 'brown' :points [[-25 5 -5] [-35 5 -5]] }
                 { :color 'brown' :points [[-25 5  5] [-35 5  5]] }
                 { :color 'brown' :points [[-25 5 -5] [-25 5  5]] }
                 { :color 'brown' :points [[-35 5 -5] [-35 5  5]] }

                 { :color 'brown' :points [[-35 0 -5] [-35 5 -5]] }
                 { :color 'brown' :points [[-35 0  5] [-35 5  5]] }
                 { :color 'brown' :points [[-25 0 -5] [-25 5 -5]] }
                 { :color 'brown' :points [[-25 0  5] [-25 5  5]] }
```

Here, the code has been inserted into the text area.

Below the widgets code text area, the widgets corresponding to the widget objects returned by the user-submitted code when called with API data are displayed. Changes to the code or new data being fetched from the API will immediately be reflected in the widgets.



All 3D visualization widgets are tiltable and zoomable by default.

```
{ :color 'brown' :points [[25 0 -5] [35 0 -5]] }
{ :color 'brown' :points [[25 0  5] [35 0  5]] }
{ :color 'brown' :points [[25 0 -5] [25 0  5]] }
{ :color 'brown' :points [[35 0 -5] [35 0  5]] }

{ :color 'brown' :points [[25 5 -5] [35 5 -5]] }
{ :color 'brown' :points [[25 5  5] [35 5  5]] }
{ :color 'brown' :points [[25 5 -5] [25 5  5]] }
{ :color 'brown' :points [[35 5 -5] [35 5  5]] }
```

Widgets preview | Raw output | Problems


```
[
  {
    "label": "Bascule bridge (3D)",
    "is3d": true,
    "surfaces": [
      {
        "color": "rgba(0, 0, 255, 0.5)",
```

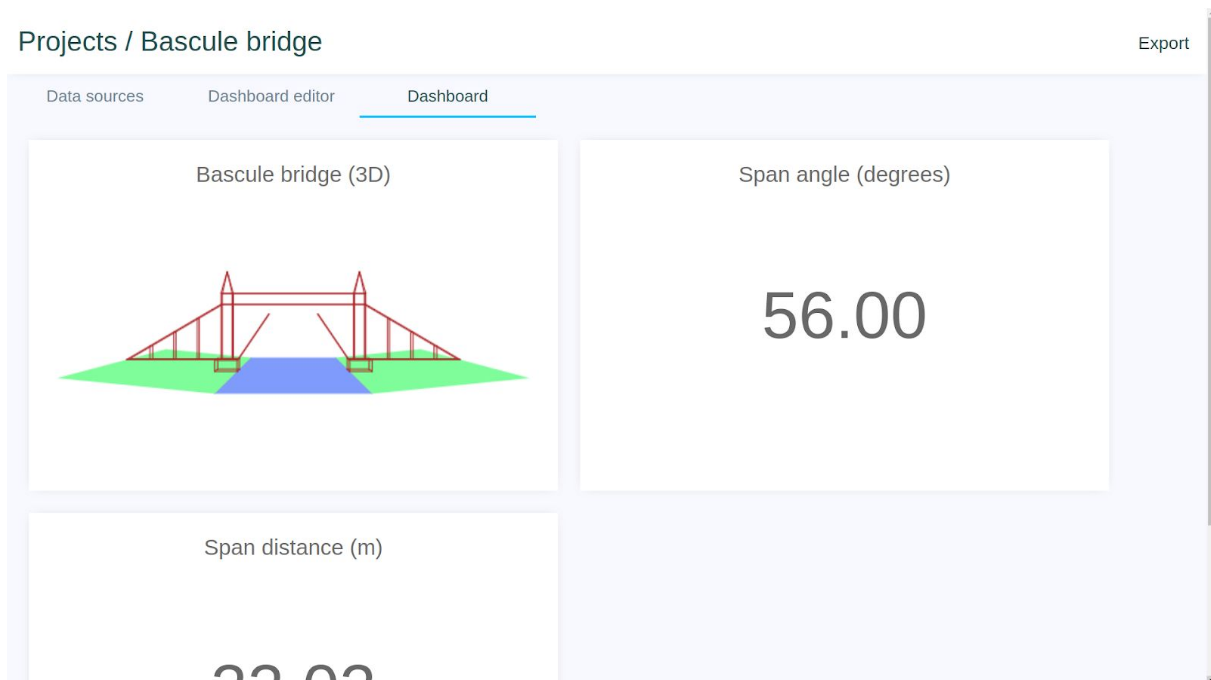

*Also below the widgets code text area is a tab called **Raw output**. Here we can preview the data which is returned from the user-submitted function called with the data from the API endpoint we specified in the **Data sources** view.*

```
                   { :color 'brown' :points [[-64.375 11.25 -2.5] [-64.375 5 -2.5]] }
                   { :color 'brown' :points [[-64.375 11.25  2.5] [-64.375 5  2.5]] }
                   { :color 'brown' :points [[-43.125 23.75 -2.5] [-43.125 5 -2.5]] }
                   { :color 'brown' :points [[-43.125 23.75  2.5] [-43.125 5  2.5]] }

                   { :color 'brown' :points [[32.5 30 -2.5] [75 5 -2.5]] }
                   { :color 'brown' :points [[32.5 30  2.5] [75 5  2.5]] }
                   { :color 'brown' :points [[53.75 17.5 -2.5] [53.75 5 -2.5]] }
                   { :color 'brown' :points [[53.75 17.5  2.5] [53.75 5  2.5]] }
                   { :color 'brown' :points [[64.375 11.25 -2.5] [64.375 5 -2.5]] }
                   { :color 'brown' :points [[64.375 11.25  2.5] [64.375 5  2.5]] }
                   { :color 'brown' :points [[43.125 23.75 -2.5] [43.125 5 -2.5]] }
                   { :color 'brown' :points [[43.125 23.75  2.5] [43.125 5  2.5]] }

                   { :color 'brown' :points [[-32.5 35 -2.5] [32.5 35 -2.5]] }
                   { :color 'brown' :points [[-32.5 35  2.5] [32.5 35  2.5]] }
                   { :color 'brown' :points [[-32.5 30 -2.5] [32.5 30 -2.5]] }
                   { :color 'brown' :points [[-32.5 30  2.5] [32.5 30  2.5]] }
               ])))
    (define center [0 25 0])
    (fun rotation-deg (do (define rotation-rad (* rotation-deg (/ 3.14 180)))
                          [{ :label 'Bascule bridge (3D)'
                             :is3d true
                             :surfaces surfaces
                             :lines (get-lines rotation-rad)
                             :center center }
                           { :label 'Span angle (degrees)'
                             :value rotation-deg }
```

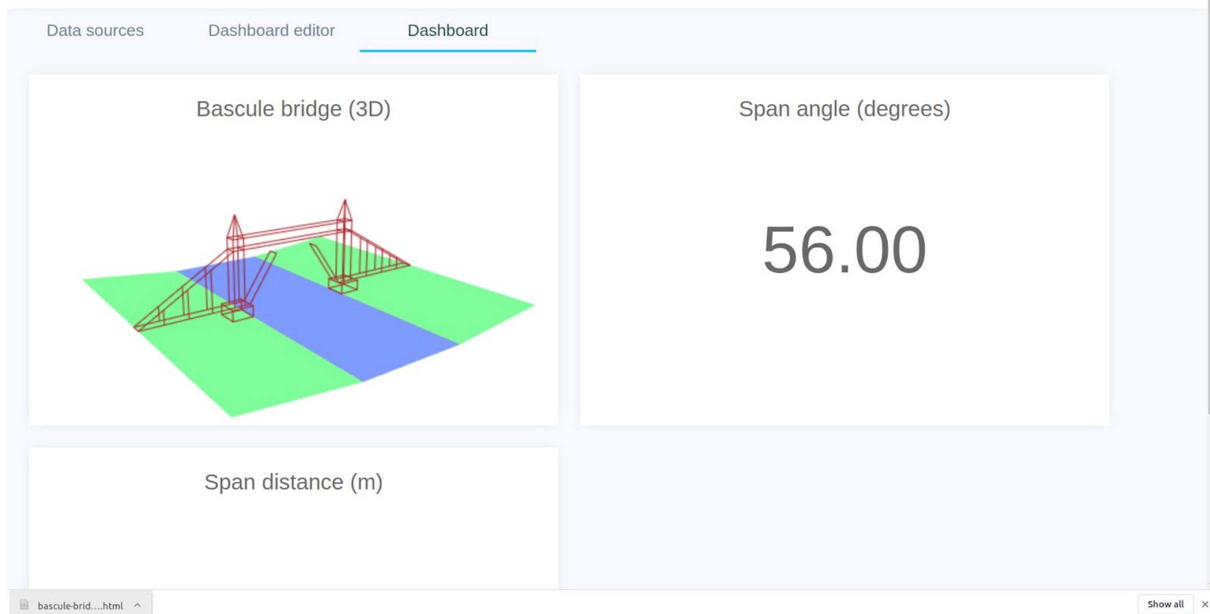
Widgets preview | Raw output | Problems (0)

No problems were detected.

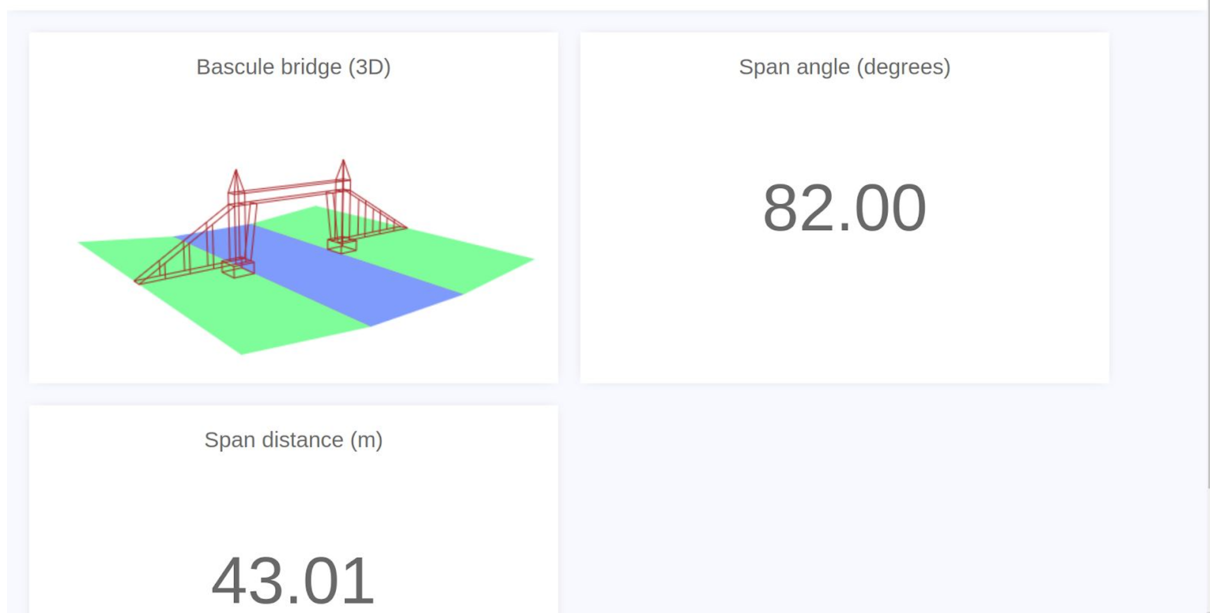
There is also a tab called ***Problems*** *where runtime errors will be displayed to assist users in explaining why their code is failing. In this example there are no problems.*



Once we are satisfied with editing the dashboard widgets code, we can navigate to the ***Dashboard*** *tab. Here, only the dashboard widgets will be displayed, being re-rendered with updated values at the selected fetch intervals.*



*In the top right corner of the screen there is an **Export** button. Clicking it exports a subset of the application, resembling the current **Dashboard** view to a single HTML file.*



Opening the exported HTML file in the browser reveals a read-only version of the dashboard we just created. Just like the unexported dashboard, the exported dashboard will continue to fetch data at the specified interval, thus enabling easy sharing of continuously updated documentation.

6.3 Performance

The Digital Shadow Language interpreter developed as part of this project was created as an alternative to the not so safe JavaScript eval function. The Digital Shadow Language interpreter is itself implemented in JavaScript, adding a performance overhead compared to evaluating the code directly using the native JavaScript eval function. We are therefore interested in comparing the performance of the Digital Shadow Language interpreter to the JavaScript eval function.

How one programming language's performance compares to another can depend a lot on the program's complexity. To give an impression of how the execution time of a program written in Digital Shadow Language compares to the execution time of a similar program written in JavaScript, the execution time of calculating the Nth Fibonacci number recursively was measured for both languages, for values of n from 1 up to and including 45.

What makes calculating Fibonacci numbers recursively great for testing is that the time complexity of the algorithm is $O(2^n)$, where n is the ordinality of the calculated fibonacci number. This means that even for very efficient programming languages, the execution time will increase exponentially, meaning that n does not have to be very large before the slowness caused by the inefficiency of the algorithm becomes significantly greater than the overhead of setting up the profiling tools, or minor variations in computer performance. The complexity of the calculations increases as n increases. The execution time of the recursive Fibonacci algorithm implemented in both JavaScript and Digital Shadow Language for increasing values of n were measured. The times where then plotted and compared to determine by what factor the Digital Shadow Language interpreter is slower than the JavaScript eval function for programs of varying complexity.

The language introduced in this thesis is a functional programming language where everything is an expression, while JavaScript is a multi-paradigm language which consists of both expressions and statements. For the comparison to be as fair as possible, the recursive Fibonacci implementation was written as similarly as possible in the two languages, utilizing

a more functional style of JavaScript, resulting in code which is slightly less readable than what could have been achieved using statements. Furthermore, rather than directly running the JavaScript code, it was evaluated as a text string using the native JavaScript eval function.

```
// Recursive Fibonacci (Digital Shadow Language)
evaluate(`
   (do (define get-nth-fibonacci
                (fun n
                     (if (= n 1)
                         0
                         (if (= n 2)
                             1
                             (+ (get-nth-fibonacci (- n 1))
                                (get-nth-fibonacci (- n 2)))))))
        (get-nth-fibonacci 20))
`);
```

This is the function used to calculate a given Fibonacci number recursively in Digital Shadow Language, evaluated using the Digital Shadow Language interpreter. (get-nth-fibonacci 20) produces the 20th Fibonacci number.

```
// Recursive Fibonacci (JavaScript)
eval(`
  const getNthFibonacci = (n) =>
    n === 1
      ? 0
      : n === 2
      ? 1
      : getNthFibonacci(n - 1) + getNthFibonacci(n - 2);

  getNthFibonacci(20);
`);
```

This is the function used to calculate a given Fibonacci number recursively in JavaScript, evaluated using the native JavaScript eval function. getNthFibonacci(20) produces the 20th Fibonacci number.

The measurements of the execution time of the functions called with different values of n were performed in Node.js. Node.js is a JavaScript runtime built on Google's V8 JavaScript engine, which is utilized in the Chrome browser (44).

A popular way to measure execution time in JavaScript in is using the native JavaScript Date class. It is intuitive to use and works in both Node.js and in browsers, but has the disadvantage of not measuring time more granularly than milliseconds.

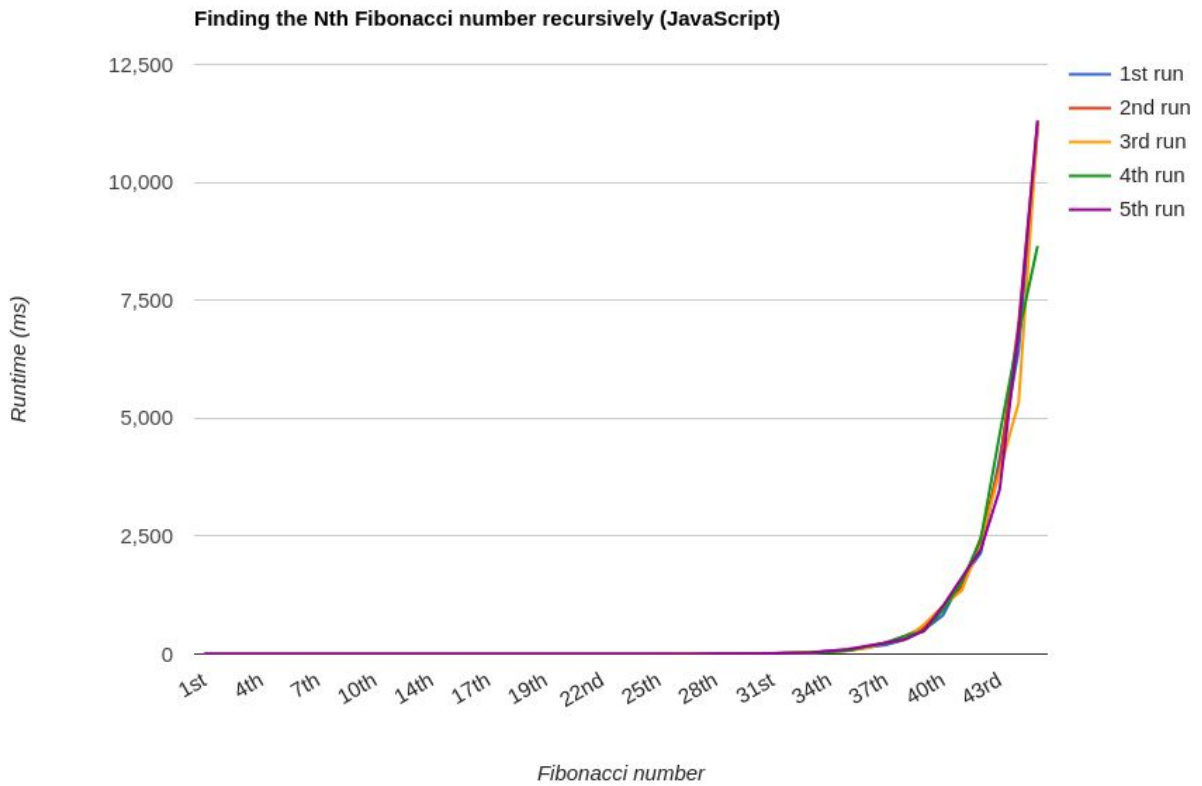
```
// Measuring execution time using the Date class
const startTime = new Date();

// Some operation whose time should be measured

const endTime = new Date();

const executionTime = endTime - startTime;
```

The executionTime constant will be assigned a number corresponding to the amount of milliseconds from startTime to endTime.

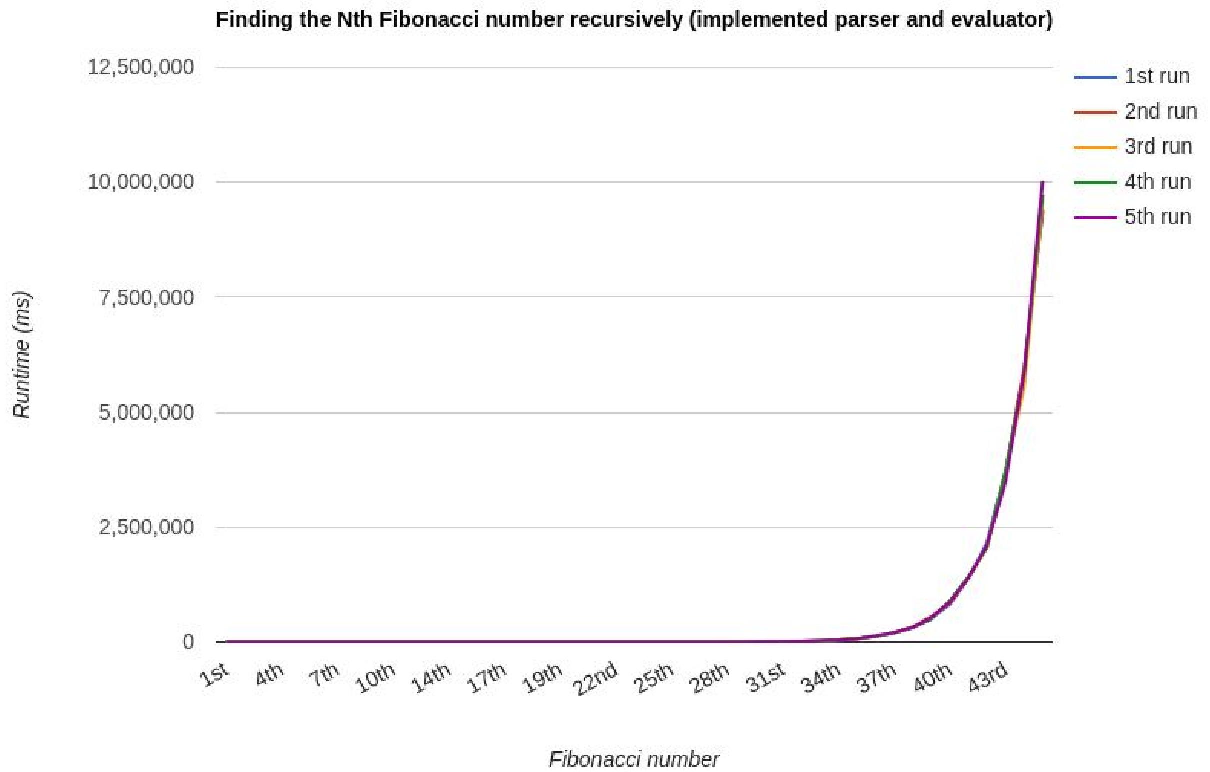
Node.js comes with an additional function, process.hrtime() to measure execution time. It returns the time in a [seconds, nanoseconds] array.

```
// Measuring execution time
const startTime = process.hrtime();

// Some operation whose time should be measured

const executionTime = process.hrtime(startTime);
```

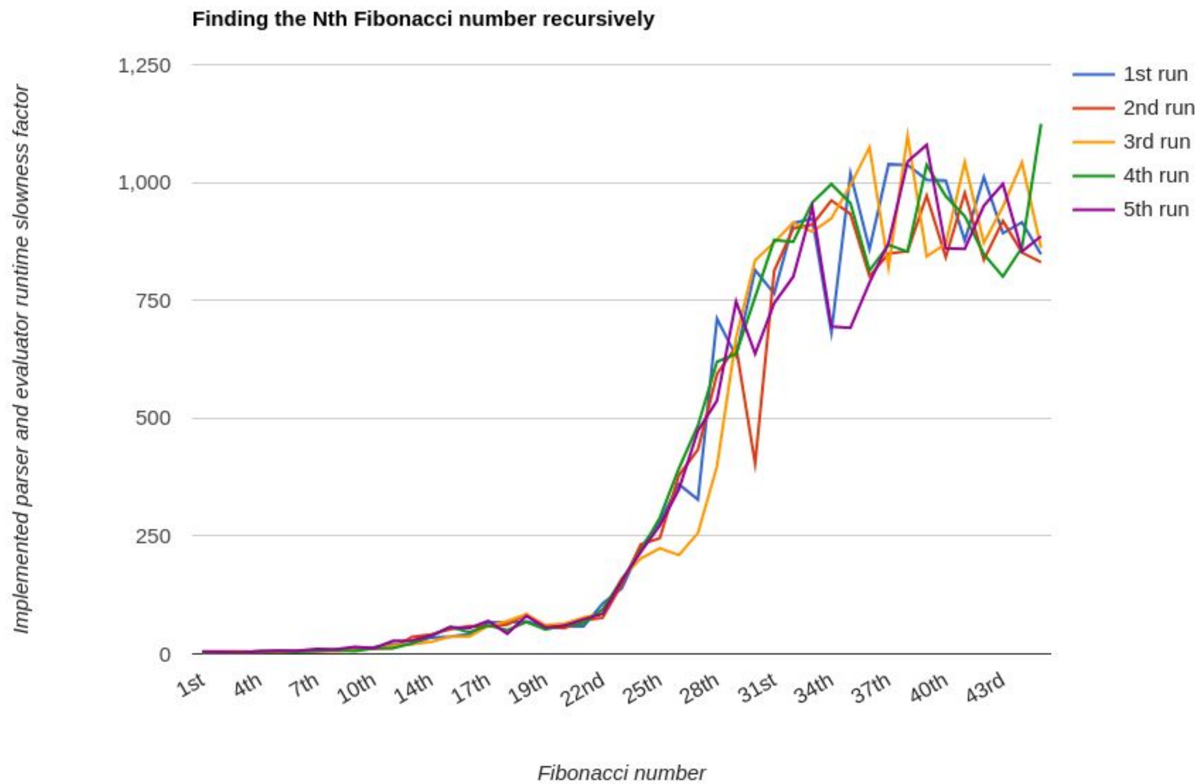
The executionTime constant will be assigned an array containing two values. The first value is a number corresponding to the amount of whole seconds have passed since startTime, and the other value is the amount of additional nanoseconds that have passed since the last whole second.



The above graph shows how the execution time of the recursive Fibonacci algorithm implemented in JavaScript increases as n increases. Calculating the 45th Fibonacci number took averagely 10.71 seconds.



The above graph shows how the execution time of the recursive Fibonacci algorithm implemented in Digital Shadow Language increases as n increases. Calculating the 45th Fibonacci took averagely 9633 seconds.



Looking at how the ratio develops as n increases further, we can observe that Digital Shadow Language evaluated using the interpreter developed as part of this project converges towards being around 800 to 1000 times slower than JavaScript evaluated using the native JavaScript eval function. However, the programs where this slowness factor occurs are very heavy, so for most basic derived value calculations, the slowness factor will be significantly lower. Empirical testing revealed that for basic data transformations and calculating simple derived values, Digital Shadow Language introduces negligibly low latency, with calculations appearing to be instantaneous.

7 Evaluation and discussion

7.1 Overview

This chapter evaluates the implemented prototype against the non-functional requirements presented in *Chapter 3 - Requirements and reasoning* to determine whether there is a gap between the non-functional requirements and the operation of the implemented prototype. The chapter further aims to determine whether those gaps are caused by the thesis statement in *Chapter 1 - Introduction* being invalid, or by other simplifications made during implementation. Lastly, the chapter discusses any findings relevant to the research questions presented in *Chapter 1 - Introduction*.

7.2 Assessment of non-functional requirements

7.2.1 Availability

Being a web application, the developed prototype can be run on any operating system, where any major web browser capable of running JavaScript can be installed. Apart from a web browser, there is also no need for the user to install anything in order to use the application.

7.2.2 Extensibility

Since the user-defined functions in the digital shadow application are applied to the data from the specified data sources on the frontend, rather than the backend, the results of these calculations cannot be exposed through an API endpoint as they could if they were applied on the backend. The digital shadow application can still be a module in a larger system, utilizing data from other modules, but other modules cannot utilize data from the digital shadow application. This limits the digital shadow application's potential to be used by other modules.

7.2.3 Performance

Bundle size and load times

The bundle size of a frontend application can vary a lot depending on the frameworks and libraries being used, as well as the complexity of the user interface. Therefore, rather than

evaluating the bundle size of the implemented prototype, I will evaluate how much of the bundle size is caused by having to include the interpreter for the Digital Shadow Language, in which user-submitted code will be written. The size of the Digital Shadow Language interpreter implemented as part of this project is 26.4 kB. The total application bundle size is 112.9 kB. In comparison, the compiled version of the js.js JavaScript interpreter is 3 MB (37).

Performance of user-submitted code

The Digital Shadow Language interpreter developed as part of this project converges towards being around 800 to 1000 times slower than the native JavaScript eval function when applied to equivalent JavaScript code. However, the programs where this slowness factor occurs are very heavy, so heavy that the JavaScript equivalent takes from 9 to 75 seconds to run. For most basic derived value calculations, the slowness factor will be significantly lower. Empirical testing revealed that for basic data transformations and calculating simple derived values, Digital Shadow Language introduces negligibly low latency, with calculations appearing to be instantaneous.

Visualization performance

Changes to visualizations caused by new data arriving are delayed by the time it takes to apply the user-submitted functions to the new visualization data. The time it takes to actually render the visualization is negligibly low, with no noticeable lag when zooming or tilting the visualizations. It is also worth mentioning that the time it takes to render the visualizations would not be different if the user-submitted calculations were rather run on the backend. When tilting and zooming in 3D visualizations, only the render logic is re-run, not the user-submitted code.

7.2.4 Usability

One of the capabilities separating a general-purpose digital shadow application for any other data visualization application is its ability to not only visualize data in a predefined set of charts, but to enable the user to visualize any physical or non-physical entity.

In the prototype developed for this project, this is accomplished by exposing a subset of the imperative operations of the HTML canvas API as a declarative interface. This gives the user

the ability to draw anything in 2D or 3D space which can be defined as an arbitrary amount of surfaces of arbitrary color, and an arbitrary amount of lines of arbitrary color and width. As such, given enough time and effort, it should be possible to visualize anything the user can imagine. However, some things are more difficult to visualize than they could have been. For instance, the HTML canvas API makes it possible to render text as part of the visualization. The methods for doing so are not exposed in the developed prototype, making it very difficult to render text of arbitrary color, size and position. There is nothing in the way of exposing the whole HTML canvas API to user-submitted widgets code, but it was not prioritized for the prototype.

The reason for this not being prioritized was partly because the prototype already accepted a value property on the widget objects returned when applying the user-submitted code to the data from the configured data sources. Any non-nullish value supplied as the value property will be displayed in the middle of the widget.

Giving the user access to a code interface is a simple way to give the user freedom to visualize nearly anything. It does, however, require that the user is familiar with the programming language used, as well as the schema of the widget objects the user-defined function needs to return when applied to the data from the API. It is unreasonable to expect that most users would be willing to learn to use a new programming language to use an application, so writing the code for the digital shadow visualizations would be a task reserved for the technically inclined.

Difficulty in defining widgets through a code interface is not a challenge specific to running user-submitted code in the frontend, but rather the idea of letting the user define widgets as code, instead of through a graphical interface.

To make it easier to define how the data from the selected data sources should be mapped to widgets, a visual programming language could be implemented. While someone familiar with Digital Shadow Language might be able to define widgets much faster than if they had to use a graphical user interface, giving the user an intuitive user interface, lowers the threshold for

being able to define digital shadows. In addition to being more intuitive to use, a GUI is better suited to prevent the user from doing something wrong.

For the prototype, implementing a visual programming language was considered. Since the implemented interpreter consists of a parser which parses the code to an abstract syntax tree, as well as an evaluator which evaluates the abstract syntax tree, a visual programming language could be implemented as a user interface whose state represents the abstract syntax tree. Then the code interface and the parser could be replaced with the visual programming language, and the current evaluator could still be used to evaluate the abstract syntax tree.

Even without being easy to use for non-technical users, a general-purpose digital shadow application depending on user-submitted code might still have a solid value proposition. The alternative to using a general-purpose digital shadow application would typically be to develop one from scratch, a task which would typically be handled by more technical people anyways.

7.2.5 Modifiability

Since the application is a web application which can be accessed as a website, updates can be delivered by the software provider by simply serving an updated application the next time the user requests the page. This is the case for any web application. For a general-purpose digital twin application specifically, the biggest advantage of being able to run digital shadow-specific logic in the browser is that one can omit a custom backend all together, meaning that changes can be implemented solely on the frontend, rather than frequently having to rely on changing both the frontend and backend. Backend as a Services like Firebase also typically come equipped with easy to use interfaces for defining database security rules or set up authentication.

7.3 Discussion related to research questions

In Chapter 1 - Introduction, six research questions were presented. In this section I will go through each of them and discuss them in context of the implemented prototype.

For digital shadows, what logic does it make sense to move to the frontend and what logic should remain on the backend?

This thesis has mainly been concerned with evaluating user-submitted code on the frontend. The user-submitted code will be used to transform API data into derived data or visualizations. Since the data has to be fetched with an HTTP request anyways, whether the data transformation is applied on the backend or the frontend will not have major impact on how instant the transformed data will be available on the frontend.

Being able to interpret user-submitted code in the client means that the data to be transformed could actually be fetched directly from the API endpoints providing them, without a detour through the digital shadow's backend. This would, however, require that the credentials needed to fetch data from the selected data source APIs reside in the client. This would make it difficult to share dashboards without also sharing the credentials, so fetching data from the APIs should remain the responsibility of the server.

From a UX point of view, it is only beneficial to move logic to the frontend if it results in quicker response to user-interaction. As we have already covered, running user-submitted code in the frontend rather than the backend does not reduce the time from the data is fetched from the API until the transformed data is available on the frontend.

However, running the data transformation code on the frontend means that the user instantly can see how the transformed data changes as they type in new transformation code. While this presents a slight advantage with regard to UX, a much bigger advantage can be accredited to the reduced complexity of the backend infrastructure. While the complexity avoided in the backend still has to be implemented in the frontend, the lower requirements for the backend means that a custom backend solution does not have to be created from scratch. Instead, a BaaS can be utilized to handle user authentication, and fetch and store data.

While moving logic to the frontend can have some advantages, it also comes with some caveats we need to be mindful of, such as blocking script execution, inconsistencies in language implementation and performance across browsers, differences in what browsers and hardware users use, and the fact that client-side computatuins cannot be trusted by the

backend. As such, whether it makes to handle some logic on the frontend rather than the backend is mainly a question of whether the desired backend solution requires it. What can feasibly be handled on the backend it makes sense to handle on the backend.

What are the main digital shadow use-cases that will suffer from the disadvantages of handling more of their logic on the frontend?

Aggregation will need to happen on the frontend each time new data arrives, instead of being able to for instance specify a standing query which continuously aggregates data on the backend as it arrives. From testing, the time it takes to for instance aggregate years of daily financial data to calculate the moving average is negligible, but for aggregations of higher runtime complexity, for instance time series prediction using pattern-matching, the user-submitted computations might take too long and be automatically aborted in order to prevent denial of service.

Assuming that aggregation of data is handled on the frontend rather than the backend, that means that the frontend will need access to all the data which will be aggregated. This will be unacceptable for use-cases where the data to be aggregated is considered private information.

How much slower is it to run user-submitted visualization code written in a language not native to the browser, in the browser, than it is to run similar JavaScript code using JavaScript's native eval function?

The performance tests conducted as part of this project show that the Digital Shadow Language interpreter converges towards being between 800 and 1000 times as slow as interpreting similar programs using the native JavaScript eval function. To prevent only being representative for a best-case scenario, little effort was put into performance optimization of the Digital Shadow Language interpreter.

More sophisticated interpreters, such as js.js which is a JavaScript interpreter written in JavaScript have been measured to only be about 200 times slower than the browser's native JavaScript interpreter (37).

Can a general-purpose digital shadow platform work with no backend at all, and what are the limitations of doing this?

Without a centralized server to store data, a pure client-side web application has some of the same limitations as native applications, one of which is that the user's project data is stored locally, with no built-in mechanism for access from other devices.

Without a centralized backend to gather data from selected API endpoints, the client will need to be responsible for fetching the data directly from the individual endpoints. This removes the API data's detour via the digital shadow backend, which can reduce latency.

However, being able to fetch data from the frontend requires that the APIs have a relaxed a same-origin policy (45). Having the client fetching data directly from the APIs will make the fetch rates of the APIs less predictable, depending on how many clients are running simultaneously.

A more significant disadvantage with having the client fetch data directly from the selected API endpoints is that all the authentication credentials need to reside in the client. This makes it challenging to share digital shadows. This will be further covered when discussing the next research question.

The digital shadow would either need to be shared along with the required authentication credentials, which is not ideal with regard to security, or it could be made possible to share a digital twin which would require the recipient to input their own API credentials. However, there is no guarantee that the person one would like to share a digital shadow dashboard with has the necessary credentials to access the APIs it depends on. Even if the recipient has the necessary credentials, the digital shadow dashboard might depend on many APIs, making it cumbersome for the recipient to fill in all the required credentials.

If the APIs used do not provide historical data, being able to create analytics which depend on historical data may prove difficult. A pure client-side platform is not well suited for storing historical data, both because the ephemeral nature of browser sessions would result in periods

of missing data, and because of limitations to the amount of data which can be stored in localStorage.

A compromise for sharing dashboards could be to share static dashboards, using the data which was the most recent at the time it was shared. In this case the exported dashboard would not be a digital shadow, but merely a digital model, since there is no mechanism to ensure that it continues to mirror the entity it models.

8 Conclusion

A prototype for a general-purpose digital shadow application was developed. Experimenting with using the prototype to create various digital shadows suggests that for basic aggregations and data transformations, it is feasible to interpret user-submitted calculations in a domain-specific language not native to the browser, in the browser.

High complexity calculations do, however, come with noticeable delay, which even if they last no longer than the time of a typical HTTP request, causes bad user experience because of JavaScript's single-threaded, blocking nature.

Perhaps more importantly, moving digital shadow calculations to the browser means that the digital shadow cannot expose the result of those calculations as an API endpoint, thus limiting itself to being an end destination for data, unable to be utilized by other modules.

9 Further work

9.1 Overview

This chapter suggests further research based on ideas which were not prioritized during implementation due to them not being relevant for the thesis statement.

9.2 Develop a general-purpose digital shadow SaaS solution

More than showing that it is feasible to run user-submitted logic safely in the browser in general-purpose digital shadow applications, the prototype has shown how a general-purpose digital shadow application is able to provide value rapidly, with minimum effort. Being able to define 2D and 3D visualization, as well as derived values, as code through a frontend code interface not only removes the hassle of having to deal with setting up backend infrastructure, but also enables the application to be used as a use-case agnostic tool for communicating of live data.

Deciding to implement the general-purpose digital shadow application prototype as a pure frontend application imposes some limitations on its ability to perform analytics on historical data, as well as its ability to enable safe and easy sharing of projects with other users.

To avoid all of these limitations, and also make the application more sellable, the general-purpose digital shadow application could be implemented as a SaaS solution. All data fetched from external data sources could go via the backend and be piped through user-submitted functions before being passed on to the frontend. This would enable continuously storing and accessing past data, and make sure that the frontend only has access to the data it needs. If the application exposes an open API, it could further be used and built upon by other systems.

9.3 Develop a visual programming for user-submitted logic

While giving users access to a code interface gives them a lot of freedom to present data in any way they wish, it also comes with a steep learning curve, and may be off-putting to non-technical users. The Digital Shadow Language interpreter developed as part of this project consists of two parts: A parser which parses code written in Digital Shadow Language to an abstract syntax tree, and an evaluator which evaluates the syntax tree to a value expression object and then converts it to JavaScript.

To make the process of defining code more accomplishable for less technical users, a visual programming language, i.e. a graphical interface for defining logic, could be developed. If the state of the visual programming language interface is designed to resemble the structure of the abstract syntax trees produced by the Digital Shadow Language parser, the evaluator which is already implemented could be reused and the current parser could be replaced by the visual programming language interface.

10 Bibliography

1. Sande, Odd Harald Sjursen & Børhaug, Andreas. (2019). Developing a client for a digital twin cloud platform.

2. Cisco Survey Reveals Close to Three-Fourths of IoT Projects Are Failing [Internet]. Cisco Newsroom | The Network. 2017 [cited 22 June 2020]. https://newsroom.cisco.com/press-release-content?articleId=1847422

3. Walker, Jeff & Chapra, Steven. (2014). A client-side web application for interactive environmental simulation modeling. Environmental Modelling & Software. 55. 49–60. 10.1016/j.envsoft.2014.01.023.

4. Grieves, Michael. (2011). Virtually Perfect: Driving Innovative and Lean Products through Product Lifecycle Management.

5. Grieves, Michael. (2015). Digital Twin: Manufacturing Excellence through Virtual Factory Replication.

6. Jones, David & Snider, Chris & Nassehi, Aydin & Yon, Jason & Hicks, Ben. (2020). Characterising the Digital Twin: A systematic literature review. CIRP Journal of Manufacturing Science and Technology. 10.1016/j.cirpj.2020.02.002.

7. Deuter, Andreas & Pethig, Florian. (2019). The Digital Twin Theory. Industrie 4.0 Management. 2019. 27-30. 10.30844/I40M_19-1_S27-30.

8. Digital Twin [Internet]. Gartner Glossary [cited 23 June 2020] https://www.gartner.com/en/information-technology/glossary/digital-twin

9. Fuller, Aidan & Fan, Zhong & Day, Charles. (2020). Digital Twin: Enabling Technologies, Challenges and Open Research.

10. Glaessgen, Edward & Stargel, David. (2012). The digital twin paradigm for future NASA and U.S. air force vehicles. 10.2514/6.2012-1818.

11. Chen, Yubao. (2017). Integrated and Intelligent Manufacturing: Perspectives and Enablers. Engineering. 3. 588-595. 10.1016/J.ENG.2017.04.009.

12. Liu, Zheng & Meyendorf, Norbert & Mrad, Nezih. (2018). The role of data fusion in predictive maintenance using digital twin. AIP Conference Proceedings. 1949. 020023. 10.1063/1.5031520.

13. Zheng, Yu & Yang, Sen & Cheng, Huanchong. (2018). An application framework of digital twin and its case study. Journal of Ambient Intelligence and Humanized Computing. 10. 10.1007/s12652-018-0911-3.

14. Vrabič, Rok & Erkoyuncu, John & Butala, Peter & Roy, Rajkumar. (2018). Digital twins: Understanding the added value of integrated models for through-life engineering services. Procedia Manufacturing. 16. 139-146. 10.1016/j.promfg.2018.10.167.

15. Madni, Azad & Madni, Carla & Lucero, Scott. (2019). Leveraging Digital Twin Technology in Model-Based Systems Engineering. Systems. 7. 7. 10.3390/systems7010007.

16. Gartner Survey Reveals Digital Twins Are Entering Mainstream Use [Internet]. Gartner Newsroom. 2019 [cited 24 June 2020] https://www.gartner.com/en/newsroom/press-releases/2019-02-20-gartner-survey-reveals-digital-twins-are-entering-mai

17. Gartner. Market Guide for Digital Twin of an Organization Technologies. 2020

18. Altran. Digital Twins: Creating Digital Operations Today to Deliver Business Value Tomorrow. 2019

19. AWS IoT Device Shadow service [Internet]. AWS IoT. 2020 [cited 24 June 2020] https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html

20. Ditto documentation overview [Internet]. Eclipse Ditto. 2020 [cited 24 June 2020] https://www.eclipse.org/ditto/intro-overview.html

21. Understand and use device twins in IoT Hub [Internet]. Microsoft Docs. 2020 [cited 24 June 2020] https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins

22. Neumann, Andy & Laranjeiro, Nuno & Bernardino, Jorge. (2018). An Analysis of Public REST Web Service APIs. IEEE Transactions on Services Computing. PP. 1-1. 10.1109/TSC.2018.2847344.

23. The WebSocket Protocol [Internet]. Internet Engineering Task Force (IETF). 2011 [cited 24 June 2020] https://tools.ietf.org/html/rfc6455

24. Fink, Gil & Flatow, Ido. (2014). Introducing Single Page Applications. 10.1007/978-1-4302-6674-7_1.

25. Nygård, Klaus. (2015). Single page architecture as basis for web applications. P. 20

26. Mernik, Marjan & Heering, Jan & Sloane, Anthony. (2005). When and How to Develop Domain-Specific Languages. ACM Comput. Surv.. 37. 316-. 10.1145/1118890.1118892.

27. Kadam, M. & Pradhan, Madhavi & Nalamwar, Sonali. (2011). Security against cross site scripting (XSS) attacks: signature based model on server side.. 1368. 10.1145/1980022.1980368.

28. Kirsten, S. Cross-Site Scripting (XSS) [Internet]. Software Attack | OWASP Foundation. 2020 [cited 25 June 2020] https://owasp.org/www-community/attacks/xss/

29. Element.innerHTML [Internet]. Web APIs | MDN. 2020 [cited 25 June 2020] https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

30. Javed, Ashar & Schwenk, Jorg. (2014). Systematically Breaking Online WYSIWYG Editors. 8909. 122-133. 10.1007/978-3-319-15087-1_10.

31. DOM onevent handlers [Internet]. Developer guides | MDN. 2019 [cited 25 June 2020] https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Event_handlers

32. Yue, Chuan & Wang, Haining. (2013). A Measurement Study of Insecure JavaScript Practices on the Web. ACM Transactions on the Web (TWEB). 7. 10.1145/2460383.2460386.

33. Snehi, Jyoti & Dhir, Renu. (2005). Web Client and Web Server approaches to Prevent XSS Attacks. INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY. 4. 345-352. 10.24297/ijct.v4i2b1.3222.

34. <iframe>: The Inline Frame element [Internet]. HTML: Hypertext Markup Language | MDN. 2020 [cited 25 June 2020] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

35. Window.postMessage() [Internet]. Web APIs | MDN. 2020 [cited 25 June 2020] https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

36. About Caja [Internet]. Google Developers. 2012 [cited 25 June 2020] https://developers.google.com/caja/docs/about

37. Terrace, Jeff & Beard, Stephen & Katta, Naga. (2012). JavaScript in JavaScript (js.js): sandboxing third-party scripts. 9-9.

38. O'Regan, Gerard. (2013). John McCarthy. 10.1007/978-1-4471-5340-5_39.

39. Davis, Martin. (2014). McCarthy John. Recursive functions of symbolic expressions and their computation by machine, part I. Communications of the Association for Computing Machinery, vol. 3 (1960), pp. 184–195.. The Journal of Symbolic Logic. 33. 117. 10.2307/2270078.

40. Queinnec, Christian. (1970). Modules, Macros and Lisp. 10.1007/978-1-4615-3422-8_10.

41. Khomtchouk, Bohdan & Weitz, Edmund & Karp, Peter & Wahlestedt, Claes. (2016). How the strengths of Lisp-family languages facilitate building complex and flexible bioinformatics applications. Briefings in bioinformatics. 19. 10.1093/bib/bbw130.

42. Realtime data [Internet]. Oslo City Bike. [cited 26 June 2020] https://oslobysykkel.no/en/open-data/realtime

43. Haverbeke M. Eloquent JavaScript. Third edition. No Starch Press. 2018.

44. Node.js [Internet]. [cited 29. June 2020] https://nodejs.org/en/

45. CORS errors [Internet]. HTTP | MDN. 2019 [cited 29 June 2020] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors

11 Appendices

Appendix A - Dashboard code example from chapter 6

In Chapter 6 - Results, the following widgets code was used:

```
(do
  (define
    surfaces
    [{ :color 'rgba(0, 0, 255, 0.5)'
        :points [[-25 0 -75] [25 0 -75] [25 0 75] [-25 0 75]] }
     { :color 'rgba(0, 255, 0, 0.5)'
        :points [[-25 0 75] [-25 0 -75] [-75 5 -75] [-75 5 75]] }
     { :color 'rgba(0, 255, 0, 0.5)'
        :points [[25 0 75] [25 0 -75] [75 5 -75] [75 5 75]] }])
  (define
    get-lines
    (fun
      rotation
      (do
        [{ :color 'brown' :points [[-25 0 -5] [-35 0 -5]] }
         { :color 'brown' :points [[-25 0 5] [-35 0 5]] }
         { :color 'brown' :points [[-25 0 -5] [-25 0 5]] }
         { :color 'brown' :points [[-35 0 -5] [-35 0 5]] }
         { :color 'brown' :points [[-25 5 -5] [-35 5 -5]] }
         { :color 'brown' :points [[-25 5 5] [-35 5 5]] }
         { :color 'brown' :points [[-25 5 -5] [-25 5 5]] }
         { :color 'brown' :points [[-35 5 -5] [-35 5 5]] }
         { :color 'brown' :points [[-35 0 -5] [-35 5 -5]] }
         { :color 'brown' :points [[-35 0 5] [-35 5 5]] }
         { :color 'brown' :points [[-25 0 -5] [-25 5 -5]] }
         { :color 'brown' :points [[-25 0 5] [-25 5 5]] }
         { :color 'brown' :points [[ 25 0 -5] [ 35 0 -5]] }
         { :color 'brown' :points [[25 0 5] [35 0 5]] }
         { :color 'brown' :points [[25 0 -5] [25 0 5]] }
         { :color 'brown' :points [[35 0 -5] [35 0 5]] }
         { :color 'brown' :points [[25 5 -5] [35 5 -5]] }
         { :color 'brown' :points [[25 5 5] [35 5 5]] }
         { :color 'brown' :points [[25 5 -5] [25 5 5]] }
         { :color 'brown' :points [[35 5 -5] [35 5 5]] }
         { :color 'brown' :points [[35 0 -5] [35 5 -5]] }
         { :color 'brown' :points [[35 0 5] [35 5 5]] }
         { :color 'brown' :points [[25 0 -5] [25 5 -5]] }
         { :color 'brown' :points [[25 0 5] [25 5 5]] }
         { :color 'brown' :points [[-25 5 -2.5] [-75 5 -2.5]] }
         { :color 'brown' :points [[-25 5 2.5] [-75 5 2.5]] }
         { :color 'brown' :points [[-75 5 -2.5] [-75 5 2.5]] }
         { :color 'brown'
```

```
            :points [[-25 5 -2.5]
                     [(- (* 25 (- 1 (cos rotation))))
                      (+ 5 (* 25 (sin rotation)))
                      -2.5]] }
          { :color 'brown'
            :points [[-25 5 2.5]
                     [(- (* 25 (- 1 (cos rotation))))
                      (+ 5 (* 25 (sin rotation)))
                      2.5]] }
          { :color 'brown'
            :points [[(- (* 25 (- 1 (cos rotation))))
                      (+ 5 (* 25 (sin rotation)))
                      -2.5 ]
                     [(- (* 25 (- 1 (cos rotation))))
                      (+ 5 (* 25 (sin rotation)))
                      2.5]] }
          { :color 'brown' :points [[25 5 -2.5] [75 5 -2.5]] }
          { :color 'brown' :points [[25 5 2.5] [75 5 2.5]] }
          { :color 'brown' :points [[75 5 -2.5] [75 5 2.5]] }
          { :color 'brown'
            :points [[25 5 -2.5]
                     [(* 25 (- 1 (cos rotation)))
                      (+ 5 (* 25 (sin rotation)))
                      -2.5]] }
          { :color 'brown'
            :points [[25 5  2.5]
                     [(* 25 (- 1 (cos rotation)))
                      (+ 5 (* 25 (sin rotation)))
                      2.5]] }
          { :color 'brown'
            :points [[(* 25 (- 1 (cos rotation)))
                      (+ 5 (* 25 (sin rotation)))
                      -2.5]
                     [(* 25 (- 1 (cos rotation)))
                      (+ 5 (* 25 (sin rotation)))
                      2.5]] }
          { :color 'brown'
            :points [[-27.5 5 -2.5] [-27.5 35 -2.5]] }
          { :color 'brown'
            :points [[-27.5 5 2.5] [-27.5 35 2.5]] }
          { :color 'brown'
            :points [[-32.5 5 -2.5] [-32.5 35 -2.5]] }
          { :color 'brown'
            :points [[-32.5 5 2.5] [-32.5 35 2.5]] }
          { :color 'brown'
            :points [[-32.5 30 -2.5] [-32.5 30 2.5]] }
          { :color 'brown'
            :points [[-27.5 30 -2.5] [-27.5 30 2.5]] }
          { :color 'brown'
            :points [[-32.5 35 -2.5] [-32.5 35 2.5]] }
```

```
          { :color 'brown'
            :points [[-27.5 35 -2.5] [-27.5 35 2.5]] }
          { :color 'brown' :points [[-27.5 35 -2.5] [-30 45 0]] }
          { :color 'brown' :points [[-27.5 35 2.5] [-30 45 0]] }
          { :color 'brown' :points [[-32.5 35 -2.5] [-30 45 0]] }
          { :color 'brown' :points [[-32.5 35 2.5] [-30 45 0]] }
          { :color 'brown' :points [[27.5 5 -2.5] [27.5 35 -2.5]]
}
          { :color 'brown' :points [[27.5 5 2.5] [27.5 35 2.5]] }
          { :color 'brown' :points [[32.5 5 -2.5] [32.5 35 -2.5]]
}
          { :color 'brown' :points [[32.5 5 2.5] [32.5 35 2.5]] }
          { :color 'brown' :points [[32.5 30 -2.5] [32.5 30 2.5]]
}
          { :color 'brown' :points [[27.5 30 -2.5] [27.5 30 2.5]]
}
          { :color 'brown' :points [[32.5 35 -2.5] [32.5 35 2.5]]
}
          { :color 'brown' :points [[27.5 35 -2.5] [27.5 35 2.5]]
}
          { :color 'brown' :points [[ 27.5 35 -2.5] [30.0 45 0]] }
          { :color 'brown' :points [[ 27.5 35 2.5] [30.0 45 0]] }
          { :color 'brown' :points [[ 32.5 35 -2.5] [30.0 45 0]] }
          { :color 'brown' :points [[ 32.5 35 2.5] [30.0 45 0]] }
          { :color 'brown' :points [[-32.5 30 -2.5] [-75 5 -2.5]]
}
          { :color 'brown' :points [[-32.5 30 2.5] [-75 5 2.5]] }
          { :color 'brown'
            :points [[-53.75 17.5 -2.5] [-53.75 5 -2.5]] }
          { :color 'brown'
            :points [[-53.75 17.5 2.5] [-53.75 5 2.5]] }
          { :color 'brown'
            :points [[-64.375 11.25 -2.5] [-64.375 5 -2.5]] }
          { :color 'brown'
            :points [[-64.375 11.25 2.5] [-64.375 5 2.5]] }
          { :color 'brown'
            :points [[-43.125 23.75 -2.5] [-43.125 5 -2.5]] }
          { :color 'brown'
            :points [[-43.125 23.75 2.5] [-43.125 5 2.5]] }
          { :color 'brown' :points [[32.5 30 -2.5] [75 5 -2.5]] }
          { :color 'brown' :points [[32.5 30 2.5] [75 5 2.5]] }
          { :color 'brown'
            :points [[53.75 17.5 -2.5] [53.75 5 -2.5]] }
          { :color 'brown'
            :points [[53.75 17.5 2.5] [53.75 5 2.5]] }
          { :color 'brown'
            :points [[64.375 11.25 -2.5] [64.375 5 -2.5]] }
          { :color 'brown'
            :points [[64.375 11.25 2.5] [64.375 5 2.5]] }
          { :color 'brown'
```

```
          :points [[43.125 23.75 -2.5] [43.125 5 -2.5]] }
        { :color 'brown'
          :points [[43.125 23.75 2.5] [43.125 5 2.5]] }
        { :color 'brown'
          :points [[-32.5 35 -2.5] [32.5 35 -2.5]] }
        { :color 'brown' :points [[-32.5 35 2.5] [32.5 35 2.5]]
}
        { :color 'brown'
          :points [[-32.5 30 -2.5] [32.5 30 -2.5]] }
        { :color 'brown'
          :points [[-32.5 30 2.5] [32.5 30 2.5]] }])))
    (define center [0 25 0])
    (fun
      rotation-deg
      (do
        (define rotation-rad (* rotation-deg (/ 3.14 180)))
        [{ :label 'Bascule bridge (3D)'
           :is3d true
           :surfaces surfaces
           :lines (get-lines rotation-rad)
           :center center }
         { :label 'Span angle (degrees)' :value rotation-deg }
         { :label 'Span distance (m)'
           :value (* 2
                     (* 25 (- 1 (cos rotation-rad)))) }])))
```

Appendix B - Running the prototype application

The prototype for a general-purpose digital shadow application which can safely run user-submitted code in the browser, is delivered alongside this thesis in a .zip file. The prototype has also been published on GitHub (see *Using the application*)

Running the application

The application is already compiled to a single HTML file and ready to use. It can be found in general-purpose-digital-shadow/dist/index.html. it can be run locally by simply opening downloading the index.html file and then opening it in Firefox or Chrome.

Using the application

To see how the application can be used, see Chapter 6 - Results or the README.md file on GitHub: https://github.com/OyvindSabo/general-purpose-digital-shadow

Building the project

If you make changes to the project, and want these changes to be reflected in the compiled general-purpose-digital-shadow/dist/index.html file, you can run the following command from the general-purpose-digital-shadow folder:

```
$ node build
```

This requires that you have node.js installed.

Continuously building the project while developing

While developing, it can be cumbersome to have to build the application for each change you make. To continuously build the application, you can run the following command from the general-purpose-digital-shadow folder:

```
$ node watch
```

Location of the Digital Shadow Language interpreter

The interpreter developed as part of this project for Digital Shadow Language can be found in general-purpose-digital-shadow/src/libraries/languageParser/LanguageParser.js

Appendix C - Digital Shadow Language examples

This appendix contains examples of code written in Digital Shadow Language, the Lisp-like domain-specific programming language designed as part of this project. Each function or special form will not be explained in detail, but the examples aim to be simple enough to be intuitively understandable and make the reader familiar with the syntax of the language.

```
(+ 4 (- 2 3))
```

The code above evaluates to 3.

```
(str 'Hello, ' 'world!')
```

The code above evaluates to 'Hello, world!'.

```
(if true 1 2)
```

The code above evaluates to 1.

```
(define a 2)
```

The code above evaluates to 2.

```
(do (define a 2)
    (define b (+ a 3))
     b)
```

The code above evaluates to 5.

```
((fun a (* 2 a)) 8)
```

The code above evaluates to 16.

```
(do (define myFunction
           (fun a b (+ a b)))
    (myFunction 1 2))
```

The code above evaluates to 3.

```
(< 1 2 3 4 5)
```

The code above evaluates to true.

```
(= 5 5 5 5 5)
```

The code above evaluates to true.

```
[1 (+ 1 1) (- 6 3)]
```

The code above evaluates to [1 2 3].

```
{ :a 1 :b 2 :c (+ 1 2) }
```

The code above evaluates to { :a 1 :b 2 :c 3 }.

```
(get { :a 1 :b 2 :c 3 } :b)
```

The code above evaluates to 2.

```
(get { :a 1 :b 2 :c 3 } ((fun (do :b))))
```

The code above evaluates to 2.

```
(get [1 2 3] (+ 1 1))
```

The code above evaluates to 3.

```
(map (fun x (+ 1 x))
     [1 2 3])
```

The code above evaluates to [2 3 4].

```
(filter even? [0 1 2 3 4 5])
```

The code above evaluates to [0 2 4].

```
(reduce + [1 1 1 1])
```

The code above evaluates to 4.

```
(do (define count-to-ten
           (fun x
```

```
                (if (< x 10)
                    (count-to-ten (+ x 1))
                     x)))
    (count-to-ten 0))
```
The code above evaluates to 10.

```
(entries { :a 1 :b 2 :c 3 })
```
The code above evaluates to [[:a 1] [:b 2] [:c 3]].

```
(entries [1 2 3])
```
The code above evaluates to [[0 1] [1 2] [2 3]].

```
(from-entries [[:a 1] [:b 2] [:c 3]])
```
The code above evaluates to { :a 1 :b 2 :c 3 }.

```
(slice 0 3 [0 1 2 3 4 5 6])
```
The code above evaluates to [0 1 2].

```
(slice 2 [0 1 2 3 4 5 6])
```
The code above evaluates to [2 3 4 5 6].

```
(slice -2 [0 1 2 3 4 5 6])
```
The code above evaluates to [5 6].

```
(keys { :a 1 :b 2 :c 3 })
```
The code above evaluates to [:a :b :c].

```
(values { :a 1 :b 2 :c 3 })
```
The code above evaluates to [1 2 3].

```
{ ((fun (do :a))) 1
  ((fun (do :b))) 2
  ((fun (do :c))) 3 }
```

The code above evaluates to { :a 1 :b 2 :c 3 }.

```
(|> 5
   (fun a (* a 2))
   (fun a (* a 2))
   (fun a (+ a 1))
   (fun a (* a 2)))
```

The code above evaluates to 42.

Øyvind Kanestrøm Sæbø

Client-Side Computing in Digital Shadow Applications

NTNU

Norwegian University of
Science and Technology