



# Safety Connection Manager UI

## A State-of-the-Art Approach

### BACHELORARBEIT

zur Erlangung des akademischen Grades

### Bachelor of Science

im Rahmen des Studiums

### Software & Information Engineering

eingereicht von

**Daniel Herczeg**

Matrikelnummer 01633037

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dieter Etz, Dipl.-Ing.(FH) MBA

Mitwirkung: Thomas Frühwirth, Dipl.-Ing.

Wien, 30. Jänner 2022

---

Daniel Herczeg

---

Dieter Etz





# **Safety Connection Manager UI**

## **A State-of-the-Art Approach**

### **BACHELOR'S THESIS**

submitted in partial fulfillment of the requirements for the degree of

### **Bachelor of Science**

in

### **Software & Information Engineering**

by

**Daniel Herczeg**

Registration Number 01633037

to the Faculty of Informatics

at the TU Wien

Advisor: Dieter Etz, Dipl.-Ing.(FH) MBA

Assistance: Thomas Frühwirth, Dipl.-Ing.

Vienna, 30<sup>th</sup> January, 2022

---

Daniel Herczeg

---

Dieter Etz



# Erklärung zur Verfassung der Arbeit

Daniel Herczeg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Jänner 2022

---

Daniel Herczeg



# Kurzfassung

Web-Applikationen haben bereits vor Jahren Einzug in unseren Alltag gehalten. Während wir sie zuerst vorwiegend im privaten Umfeld wahrgenommen haben, sind sie seit einigen Jahren immer stärker auch in professionellen und industriellen Settings anzutreffen. Was als Web 2.0 begann, führte schon sehr bald zu IoT, Industry 4.0, und Edge-Computing. Das sind bei Weitem keine leeren Schlagworte mehr, wie ein Blick in die aktuellen Kataloge führender Hersteller von industriellem Equipment zeigt. So haben führende Hersteller wie Siemens und B&R bereits Web-basierte Lösungen in ihr Portfolio aufgenommen. Die Vorteile von Web-Applikationen sind vielfältig. Sie beinhalten gute Skalierbarkeit, einfache Änderbarkeit, schnelle Auslieferung von Updates und sie funktionieren auf einer Vielzahl von Displays ohne große Anpassungen. Weiters können Clients sehr simple Anzeigeräte sein. Was es braucht, ist lediglich ein einfacher moderner Webbrowser. Daher behandelt diese Arbeit die Entwicklung einer modernen Web-Applikation, welche es Mitarbeitern in einem industriellen Umfeld erlaubt, sicherheitsrelevante Netzwerkgeräte zu visualisieren und diese bei Bedarf zu verwalten. Weiters wurde eine kurze Umfeldanalyse durchgeführt, um den Bedarf einer solchen Applikation zu ermitteln.





# Abstract

Web applications found their way into our everyday lives years ago. Since they first appeared in our private lives, web applications have become more and more relevant in many professional and industrial settings. What began as Web 2.0 soon led to IoT, Industry 4.0, and edge computing. However, these are by no means empty buzzwords, as a look at the current catalogs of leading manufacturers of industrial equipment shows. Leading manufacturers such as Siemens and B&R have already included web-based solutions in their portfolios. Web-based solutions offer a range of advantages, such as good scalability, adaptability, and fast rollout of updates. Further, they work on a large number of displays without significant adjustments. In addition, clients can be simple display devices, as all it takes to run a web application is a modern web browser. Therefore, this thesis discusses developing a state-of-the-art web application that allows employees in an industrial environment to visualize safety-relevant network devices and manage them if necessary. Furthermore, a short market analysis was carried out to determine whether there is a need for such an application.



# Contents

<b>Kurzfassung</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>Contents</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Problem Statement . . . . .	1
1.2 Aim of the Work . . . . .	2
1.3 Structure . . . . .	3
<b>2 State of the Art Analysis</b>	<b>5</b>
2.1 Industrial HMI Devices and Common Features . . . . .	5
2.2 Graph Drawing Algorithms . . . . .	8
2.3 Web Technologies . . . . .	10
2.4 UI/UX Design . . . . .	21
<b>3 Project Details and Implementation</b>	<b>25</b>
3.1 This Project's Technology Stack . . . . .	25
3.2 Backend Features . . . . .	26
3.3 Frontend Features . . . . .	29
3.4 Backend Application Implementation Details . . . . .	31
3.5 Frontend Implementation Details . . . . .	38
3.6 Testing . . . . .	41
<b>4 Summary, Improvements, and Outlook</b>	<b>43</b>
4.1 A High-Level Summary of the Implemented System . . . . .	43
4.2 Added Value, Lessons Learned, and Outlook . . . . .	44
4.3 Results . . . . .	48
<b>Acronyms</b>	<b>55</b>
<b>Bibliography</b>	<b>57</b>



# Introduction

## 1.1 Motivation and Problem Statement

A modern factory floor comprises numerous complex and expensive machines that may pose risks to an operator's safety, health, and life in case of a critical malfunction. Aside from endangering the operating personnel, a malfunctioning device or machine may cause costly damages to existing equipment or the environment.

Safety defines the absence of unacceptable risk of injury or damage to the health of people, either directly or indirectly, for example, as a result of environmental damage. Functional safety requires a system or equipment to operate correctly in response to its inputs. Safety and functional safety must always be determined considering the system as a whole as well as the environment with which it interacts [Bel06].

Computerized systems are not easy to predict, and testing alone may not be sufficient to ensure reliable operation during safety-critical conditions. Therefore, more complex programmable electronic solutions call for different assessment methodologies and safety considerations. Several standards were developed across the globe to address these problems in various safety-critical applications ranging from aviation to the chemical process sector. While the standards initially focused on software, it became clear that safety requires a holistic approach [Bel06].

The international IEC 61508 standard provides guidelines for implementing safety-related systems that use electrical, electronic, or programmable electronic technology so that the whole system achieves an acceptable level of functional safety. This standard applies to implementing, controlling, and maintaining any safety-related electric, electronic, or programmable control or protection system [Bro00].

In general, such systems might operate on-demand or continuously. On-demand systems act to protect against hazardous conditions. An example of such a system is a kill-switch

that cuts the power to a motor when it detects a dangerous operating temperature within the motor housing [Bel06, Bro00]. These dangerous operating conditions may, for example, be caused by malfunctioning equipment, the environment, or human error. A motor that drives a belt in a factory is an example of a continuous system. The system would need to reduce the belt speed as long as workers perform maintenance on a nearby machine [Bel06].

Modern production facilities often employ a wide array of safety sensors and switches. Physical buttons and controls allow operators to stop a machine in case of a problem. Various sensors in combination with an input processor such as a Programmable Logic Controller (PLC) or a computer can automatically stop machines when they detect dangerous operating conditions. In addition, safety-critical hardware is often duplicated to strengthen the operation's fault-resistance by creating physically redundant networks of sensors. Further, it is usually not desired that a detected anomaly in one area of a factory causes the entire manufacturing process to shut down. Instead, operators may want to configure sensors in such a way that they control devices nearby.

The sheer amount of safety-critical devices and machines results in complex networks that can quickly become too difficult to overview, maintain, and manage. In addition, attaching new safety-critical devices may be cumbersome, and a faulty configuration may compromise the safety or efficiency of the factory.

### 1.2 Aim of the Work

As mentioned in Section 1.1, factories include a large number of switches and sensors that continuously report information about their state to an electronic, electric, or programmable electric controller. The controller can then decide whether it needs to act to prevent hazardous situations. The safety logic can then send signals to actuators in the factory in order to avoid failure conditions, for example, by opening a pressure release valve. These sensors, controllers, and actuators are connected via links, and this work presents a web-based computer program that allows operators to view the current configuration of safety-critical devices in the network.

The presented solution displays a portion of the factory's physical network and overlays the relevant safety-critical connections. The devices are connected via an existing network infrastructure, and an external discovery algorithm allows operators to add and remove safety-critical hardware dynamically. However, the discovery mechanism and the physical construction of the network are not part of this work.

This thesis focuses on the software that displays the results of the underlying algorithms and the User Interface that allows operators to configure and change safety-relevant parameters. The final User Interface solution should function on various standard-sized displays and offer an overall good User Experience.

## 1.3 Structure

Chapter 1 introduces the reader to the problems that motivated this work. It also discusses how this thesis aims to solve the issues presented. Chapter 2 gives an overview of the current state of the art in industrial HMI applications. It briefly introduces common features shared among modern HMI devices. Then chapter 2 discusses current web technologies in great detail and outlines the reasons for selecting certain technologies for this project. Lastly, it also summarizes recent trends in UI and UX design. Chapter 3 discusses the project requirements, planning details, and the final implementation. Chapter 3 also discusses features that are not part of this project. Lastly, Chapter 4 of this thesis summarizes the final solution and findings of this project and possible future work that could improve the project discussed in this thesis.





# State of the Art Analysis

This chapter summarizes the state of the art of Human-Machine Interfaces at the time of writing this thesis. Several aspects that make up complete industrial HMI systems have been studied for this project, and the most important aspects have been summarized in this chapter. First, this chapter introduces readers to common features of modern industrial HMI solutions. Identifying typical features across such devices allows the author to design the UI with these capabilities and limitations in mind. Then, this section identifies a few common methods for drawing graphs. Doing so is important to understand how the frontend of this project will display the graphs. Finally, as the frontend will be implemented using modern web technologies, the last part of this section discusses the state of the art of web-based applications as well as design considerations for the UI.

## 2.1 Industrial HMI Devices and Common Features

A short market sampling phase was conducted to query what HMI devices large manufacturers offer and to learn about the features these devices commonly include. For that purpose, the online catalogs of two major industrial suppliers, namely Siemens and B&R, were studied. This section does not represent an exhaustive market study.

### 2.1.1 Siemens SIMATIC Range of Devices

The Siemens SIMATIC IPC range of industrial panel-PCs share a few features that are interesting for this project [Sie21b]. For one, all models of the SIMATIC IPC range come with single- or multi-touch capable full-color displays with varying sizes between 7" and 24", depending on the model. An Intel-based microprocessor forms the base of every model in this range. These are typical processors one would also find in home- and office applications. Finally, all devices across the SIMATIC IPC range of panel-PCs support standard high-level operating systems such as Microsoft Windows.

Contrasting the SIMATIC IPC range, Siemens also offers a thin-client range of industrial HMI panels called the SIMATIC Industrial Thin Client [Sie21a]. These devices come with a range of full-color capacitive multi-touch displays ranging from 15" to 22". The displays also offer a resolution up to full HD. These devices contain limited processing power, designed for displaying web-based applications. For that purpose, they can connect to an existing network using a wired Ethernet interface. The SIMATIC Industrial Thin Client has a simple operating system that offers limited features. Its primary purpose is to display web applications via an HTML-5 capable web browser.

### 2.1.2 B&R Automation Panel Range

Similar to Siemens, B&R Automation offers a range of modular panel display solutions. These devices come in sizes between 4.3" and 15.6", and they include a full-color capacitive HD multi-touch display. [Aut21].

For this project, the B&R Automation Power Panel T-Series range of HMI displays is the most relevant current product of the B&R portfolio. This range of devices functions similar to the Siemens Thin Client displays discussed above. Customers can choose from various screen sizes suitable for their application. The HMI terminal uses a standard Ethernet interface to communicate with a controlling device. The terminal itself only has limited control capabilities. It can either function as a VNC terminal or as a simple client with a full-screen web browser that runs web applications.

### 2.1.3 Commonly Found Features in Industrial HMIs

This thesis did not conduct an extensive market analysis. Instead, it investigated two specific industrial HMI product ranges made by two major suppliers. Therefore, it is not possible to make statements about future developments in industrial HMI applications. However, this short market snapshot analysis reveals a few aspects that modern HMI solutions have in common. Modern HMI solutions are often based on standard PC hardware in a more robust and rugged package. The investigated products offer large full-color single- or multi-touch displays. Besides executing proprietary industrial software such as WinCC, modern HMI panel-PCs can also run high-level operating systems such as Microsoft Windows 10. These aspects mean that modern industrial panel-PCs are well-equipped for running web-based applications such as the one proposed in this thesis.

There seems to be a trend away from physical buttons and simple HMI displays towards more elaborate integrated touch-screen solutions. Instead, many HMI solutions include some sort of processing power, often even in the form of a full PC running a high-level OS such as Windows. However, some panels are designed with web-based applications in mind. They offer enough processing power to run a modern web browser that can render and execute web-based applications such as the one presented in this thesis.

Other devices from the same manufacturers seem to confirm this hypothesis. There appears to be a trend away from solely built-in panel-PCs toward more flexible hybrid or mobile solutions such as rugged thin clients and industrial-grade tablet PCs. Such devices offer operators more flexibility, and the software discussed in this thesis should be flexible enough to work on displays of varying sizes and resolutions.

### 2.1.4 Siemens WinCC Unified – A Modern Web-based HMI Software Solution

With its TIA portal, Siemens offers its customers an extensive suite of software solutions specifically tailored for use in industrial settings [Sie21c]. This section of the thesis summarizes three important features of WinCC Unified, a visualization software inside of the Siemens TIA portal. WinCC is compatible with the aforementioned Siemens HMI solutions.

WinCC Unified is based on current web technologies, and the system allows operators to access it via any modern web browser [Sie21f]. The use of web technologies allows the system to scale to almost any display size and still retain its usability. Furthermore, clients do not need lots of processing power, complicated hardware, or an elaborate operating system to display the WinCC Unified interface. Instead, all that is required is an uncomplicated connected terminal with a built-in web browser.

So-called faceplates within WinCC Unified allow system engineers to define HMI elements that look, feel, and function the same across various devices. These configuration objects can be reused as needed and, thus, reduce the configuration effort and factory downtime [Sie21d]. Besides faceplates, user-defined styles allow operators to fine-tune the HMI experience within WinCC.

Lastly, View-Of-Things allows operators to use WinCC Unified on certain Siemens PLCs to inspect settings, previously made using the TIA portal [Sie21e].

The three presented features are the most relevant ones for this thesis, as they represent the main features of the discussed project. First, the project should be web-based, and it should only require a modern web browser to work. Next, the proposed software solution should function on various screen sizes. Further, the finished product should allow operators to effortlessly inspect and update vital system parameters.

The new system could work within WinCC Unified, as the Siemens software suite allows operators to define custom web-views in WinCC that display websites that are not part of WinCC. Such a website could be the software proposed in this thesis. In contrast, the new software solution could also work as a competing product with limited features compared to WinCC. However, the limited features also result in a less steep learning curve and a shorter time to market.

## 2.2 Graph Drawing Algorithms

The project discussed in this thesis focuses on visualizing an existing hardware topology of relevant security devices and machines on a factory floor. For that, the finished product should display a graph of some sort that quickly lets operators visualize a relevant section of the complete network they are working with. Finding an adequate representation of the devices, safety switches, and sensors presents itself to be a visual optimization problem commonly found throughout many scientific disciplines. Structures that involve nodes and connections between the nodes can often be represented using graphs [Tun94].

It is important to note that a set of nodes and their connecting edges can be represented in infinitely many ways. However, the usefulness of any graph depends on its ability to convey the encoded information (i.e., its readability for the individual user). To be effective, a graph must convey the meaning of the encoded data quickly and clearly [DBETT94]. On the other hand, an inappropriate visual presentation of information can confuse users [Sug02], and in the case of this project, result in costly mistakes. Therefore, a few selected graph layout algorithms have been studied prior to implementing the final solution in order to find a suitable algorithm that can present a factory network structure in a visually pleasing way.

What is visually pleasing often differs from person to person, and it also depends on the cultural background. However, four often competitive principles seem to form a common base. The first two are that a graph should minimize the number of crossing edges and that the finished graph should achieve some form of visual balance (i.e., spread the nodes evenly in the available area) [KK<sup>+</sup>89]. Next, edge lengths should be uniform, and nodes that are non-adjacent in the model should be farther apart in the resulting graph compared to adjacent nodes [Tun94]. This thesis uses these four principles as its base for drawing graphs. Interested readers can find various other static and dynamic rules that help create visually pleasing graphs identified and described by Sugiyama [Sug02].

### 2.2.1 The layering Method of Sugiyama, Tagawa, and Toda

The layering method of Sugiyama et al. is still the most popular approach for drawing graphs, and its basic four steps are commonly used in many graph drawing libraries and tools [Tun94, ESK04]. The layering method splits the graph into layers of equally spaced rows. Then, the algorithm distributes the nodes on the previously created layers with the goal of minimizing edge crossings and edge lengths. When an edge connects two adjacent nodes that got placed on non-adjacent layers in the graph, the algorithm adds dummy nodes to the layers between. The two original nodes and the inserted dummy nodes form a path over multiple layers. In a later step, the algorithm removes these dummy nodes and replaces them with curved edges. A detailed description of the algorithm can be found in Eiglsperger et al. [ESK04]. It also discusses the layering method's runtime characteristics.

There are conflicting views as to whether the Sugiyama method yields good results in reasonable time constraints. Some authors conclude that the layering method produces aesthetic drawings fast enough [Tun94], while others discuss various options to improve the speed of certain steps in the algorithm [ESK04]. However, the consensus is that each of the four sub-steps of the Sugiyama method is an NP-hard problem. Therefore, implementations of the Sugiyama algorithm use heuristics and algorithms that approximate the ideal solution. An efficient implementation also makes use of efficient data structures that hold the graph information.

In the project presented in this thesis, the graph calculation and drawing routines are executed on the client-side (i.e., the device running the web browser that displays the frontend web application). As discussed above, these client devices can be of any sort, and they may sometimes offer only limited CPU power and memory. When a user requests a graph view of the network, the device should not take longer than a few seconds to update and render the graph view. Therefore, it will be important to pay close attention to the runtime and memory requirements of the graph rendering algorithms used in this project. The exact specifications and implementation details will be discussed in chapter three of this thesis.

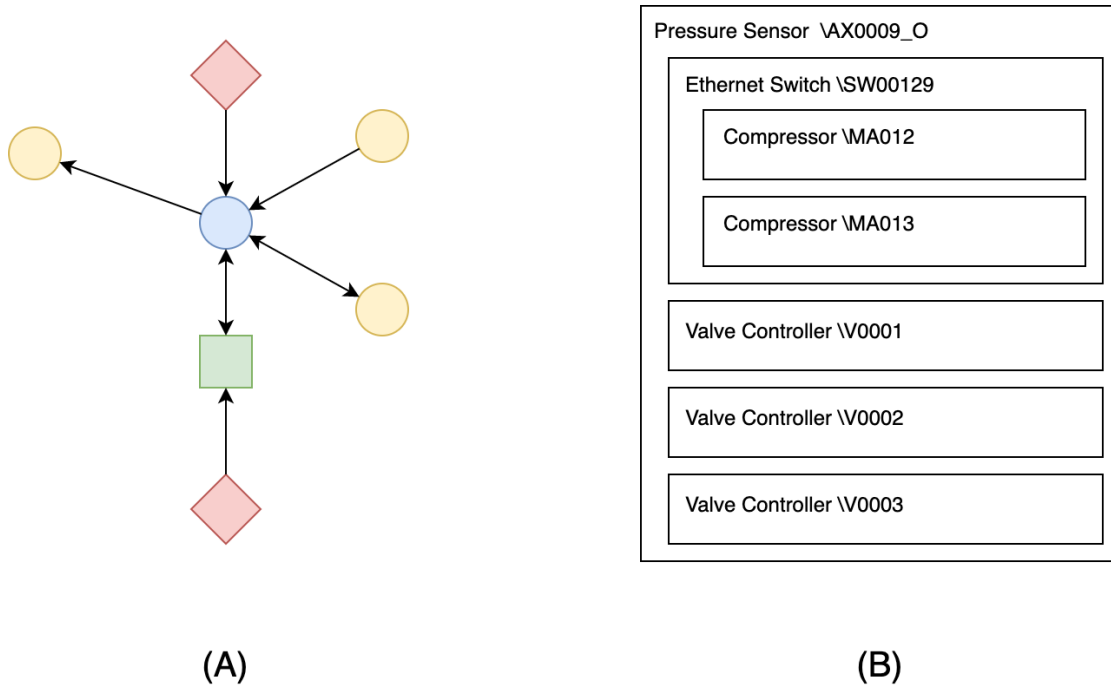


Figure 2.1: This figure illustrates the two types of graphs relevant for this project

### 2.2.2 The Spring Embedder Model

The spring embedder model is another class of algorithms for automatically generating graph representations of a network. Algorithms that implement this model are also referred to as force-directed algorithms [FK12]. This class of algorithms produces very elegant solutions that work well for simple undirected graphs. Spring embedder algorithms often produce symmetric graphs with uniform edge lengths, which are two properties this project wants to achieve when rendering graphs such as the one in Figure 2.1 A.

Force-directed algorithms utilize an energy value to characterize the quality of a graph drawing. A function maps each layout to such an energy value so that graphs that have adjacent nodes closer together and non-adjacent ones spaced farther apart receive a low energy value [Kob12]. As discussed above, this closeness of adjacent nodes and the spatial separation of non-adjacent ones are two more desirable factors for graphs. Another important factor is that visually pleasing graphs should have as few crossing edges as possible. However, The energy-function of the embedder model does not consider crossing edges. Fowler et al. [FK12] discuss several concrete implementations of force-directed algorithms as well as their specific runtimes.

Many graphs can not be made free of crossings in practice, and it is, thus, often necessary to make a compromise by relaxing one of the requirements. In most cases, that will either be to allow a few crossings or to allow curved edges. Which constraint to relax depends on the context of the application and the author's preference. Several algorithms and theses exist that aim to solve this problem, and Bläsius et al. give an overview of the problem and two possible solutions [BRR17].

Spring-embedder (force-directed) algorithms seem to be a reasonable option for drawing graphs in this project. This class of algorithms achieves most of the desired goals discussed earlier in this thesis, and the algorithms do so within reasonable time constraints.

## 2.3 Web Technologies

Throughout the last few years, websites have changed from static pages that deliver content such as news articles and image galleries to a mostly interactive experience. Many modern websites provide their users with services that look and feel like desktop applications. Users do not have to install any software except for a modern web browser. Therefore, this thesis focuses on a web-technology-based implementation for the presented project, and this section investigates current trends and the state of the art of web technologies.

### 2.3.1 RESTful Web Services

RESTful services are applications that correctly implement certain principles as described by Fielding [Fie00]. Fielding's conclusion was that if an application adheres to a few principles, it also shows desirable properties such as scalability, simplicity, and decoupled design. Often, RESTful services are published on the Internet, and they typically use the

HyperText Transfer Protocol (HTTP) for communication, although there is no constraint that enforces utilizing HTTP [Lan16].

In a RESTful web application, the server holds some resources, and clients can request a representation (e.g., a JSON-encoded document) of a resource's state (i.e., the actual data associated with a resource). For that, a client only needs to know about the server's interface, how to access it (i.e., the resource's identifier, or URI), and what the answer from the server looks like. Clients do not need to know anything about the internal structure of the server or the implementation details behind it [Lan16].

Aside from transmitting representations of a state when instructed, a server must adhere to six principles to be considered RESTful [Fie00]. Note that one of these principles is optional, even by the original definition [Lan16]. Some authors loosened this constraint, and various maturity models exist that classify applications according to which of the principles they implement. One such model is the Richardson Maturity Model [Pau14, Fow10]. According to the original definition by Fielding, the software presented in this thesis will adhere to all mandatory RESTful principles.

The following section contains a summary of the original six RESTful principles. Interested readers may refer to [Fie00] or [Lan16] for further details:

1. Server-Client

RESTful web applications use a clearly separated client-server architecture. Servers hold resources that they manage, and clients request representations of the resources that they can work with (e.g., update data values and send the representation back to the server for storage in a database).

2. Stateless

This constraint states that the client keeps all information about the session. The server does not know anything about it. From the server's perspective, each interaction with a client is unique, and the client must transmit all data required to fulfill a request each time a request is made.

3. Cache

Responses from the server must be marked as either cacheable or non-cacheable.

4. Uniform Interface

RESTful applications must have an interface that's decoupled from the implementation. First, a RESTful application must hold resources that have to be accessible via the server's interface. Anything that can be addressed with an identifier (i.e., a URI) may be viewed as a resource within a RESTful application [CJP<sup>+</sup>20].

Next, in a RESTful application, resources are manipulated via their representation. As noted earlier, clients do not directly access the resources (e.g., by modifying a database entry). Instead, the client requests a representation from the server, updates it, and sends the altered version back to the server. The server then handles the database update, for example.

Furthermore, each message must have all information that the other side needs to understand the message in isolation. So, for example, each message must contain a media type associated with it. When using HTTP, the HTTP methods (e.g., GET, POST, DELETE) may be used to inform the recipient of a message what to do with the data.

Lastly, RESTful applications should use links to navigate between application states (HATEOAS). For this purpose, representations can be enhanced with links to relevant resources.

### 5. Layered System

Clients should not be required to know about any layers behind the public interface. There may even be various proxies between the client and the actual server, and the client does not have to know that.

### 6. Code-On-Demand

The server may send code (e.g., a JavaScript snippet) to the client to extend the client's functionality. This is the only optional RESTful constraint, and this project does not implement it.

GraphQL is an alternative solution for building modern web-based APIs. However, the method is only mentioned here for the sake of completeness. Interested readers may learn more about GraphQL and its benefits in various external sources such as [BMV19].

### 2.3.2 The MVVM Pattern

One of the requirements for this project is that users get notified as soon as possible whenever something changes in the underlying data storage. So, for example, if somebody plugs in another safety switch, the exploration algorithm detects this change and submits it to the data storage. The frontend should get notified of this change as soon as possible so that it can change its representation and inform the operator, if necessary.

MVC is a commonly used design pattern when building layered applications. Model-View-Controller is a three-layer model, and it belongs to the family of multi-layer architectural models. MVC exercises separation of concerns, as it contains three modules, each responsible for a very specific task. The view component models the User Interface, the model contains the logic required for all types of interactions with the underlying data storage, and the controller modules contain the business logic of the application [AA18]. Note that a module may consist of multiple classes.

One significant problem of the MVC design pattern is that the controller may grow into a single large monolithic block that defeats the purpose of splitting the application into smaller, more manageable units. This problem often occurs because the controller is also responsible for handling user input [AA18], and it, thus, must provide many callback functions if an application contains many input fields. In addition, it might not be clear which module should handle input and data validation.



As mentioned, MVC is popular among full-stack developers, and as a consequence, each class is more developer-friendly. Designers and team members with less experience in software development might have a more difficult time trying to get an overview of existing code that follows the MVC pattern. In contrast, the MVVM pattern aims to present itself as a more designer-friendly alternative to MVC [L<sup>+</sup>16].

The Model-View-Viewmodel pattern is similar to MVC in a sense that both implement a layered software architecture. Model-View-Viewmodel contains three layers, namely a view, a model, and a view-model component. The view and the model components have the same responsibilities as in MVC. The view implements the User Interface, and the model handles the underlying data storage. The major difference between MVC and MVVM is the view-model component that sits between the view and the model components.

The view and the model components of a program that implements the MVVM pattern can be fully isolated and written in different programming languages. A frontend application, such as a website, implements the view, while a backend application, such as a Java server, implements the backend logic. The View-Model component sits between the frontend and the backend and translates commands and actions between the other two software components. MVVM was specifically developed to simplify the development of event-driven User Interfaces. With the use of data-binding connections, the view and view-model components stay synchronized. These connections may be uni- or bi-directional, depending on the application's requirements [L<sup>+</sup>16]. With this type of connection, the developers do not have to implement the view component in such a way that it needs to constantly poll the view-model for changes, and the model component does not need to inform the view of changes in the underlying data structure. These interactions are performed by an MVVM handler that comes with popular MVVM frameworks.

What makes the MVVM pattern so interesting for modern web-based applications is that this model allows two separate teams of developers to simultaneously work on the frontend and backend without interfering with each other. All the teams need is a description of the interface that connects the two components. In addition, the frontend team may consist of engineers who specialize in UI design, and the backend team can contain software developers who specialize in backend development [SM10]. Furthermore, the teams are no longer required to use the same programming language. Instead, markup languages and even automated GUI generation tools may be used, which enables designers who are inexperienced in programming to work on the frontend.

### **2.3.3 Overview of Commonly Used Programming Languages and Frameworks in Web-Applications**

As RESTful web services describe a set of design rules rather than a concrete language or implementation, any programming language could be utilized for building modern RESTful web applications. However, there are a few languages and frameworks that are commonly used for this purpose [Pau14].

As this project contains a web-based frontend, the use of HTML5, CSS, and JS/TS is inevitable for the web-based client. This project aims to employ a state-of-the-art framework to reduce the development time and increase the robustness of the final frontend client.

For the backend part of this project, various programming languages and frameworks may be used. As mentioned, the RESTful pattern does not enforce the use of a specific protocol or programming language. However, it makes sense to use a modern and commonly used programming language to achieve good maintainability for the final solution.

### 2.3.4 Java, Apache Maven, and Spring Boot

One way to implement the backend is to use Java as the programming language. Besides other alternative options, RESTful backends in Java are commonly built on top of Spring Boot in combination with Apache Maven.

As described by Suryotrisongko, Spring Boot offers a plethora of features that help developers build robust RESTful backend applications in Java [SJT17]. In addition, the Spring Boot framework has become a de-facto standard for various types of applications developed with the Java programming language [Wal15]. In his book, Walls also describes how the original spring framework started as a light-weight alternative to JavaEE Beans. However, the original Spring framework has had the significant downside of being configuration-heavy, which occupies developer time early on in a project. In addition, new updates in sub-modules may break existing dependencies. Spring Boot, the newer version of the Spring framework for Java, solves these issues, as it automates most of the configuration generation [JJVR17].

Either way, besides web-based RESTful APIs, Spring Boot also offers developers tools for building various other types of programs ranging from light-weight microservices to monolithic enterprise applications. Regardless of the project, the main concepts of Spring Boot are Inversion of Control, Dependency Injection, Aspect-Oriented Programming, and the use of the Java persistence API.

As the newest version of Spring uses automatic configuration, it is typically used in combination with a software project and dependency management tool such as Apache Maven or Gradle. An official online-tool [Spr21] lets developers set up a new Spring Boot project based on so-called starters. These templates automatically configure the project and include standard dependencies according to the project type. More experienced developers may want to configure Spring manually, and the framework offers that option, too [WSL<sup>+</sup>13].

One argument that supports using the Spring Boot framework in this project is that the developers are already proficient in using Spring Boot due to a previously conducted university project and external work done in other small project teams. Therefore, it would not take too long to develop a functional prototype, as the training period would be very short. Furthermore, Spring Boot has become a de-facto standard for developing

RESTful Java backends, and the framework is widely supported by a large community of developers. Therefore, it would be easy to find assistance should problems arise during development.

### 2.3.5 NodeJS

NodeJS is an open-source runtime environment for executing JavaScript code outside of a browser. This program lets programmers shift JavaScript code execution from the client-side to a (web)server. Therefore, NodeJS allows developers to implement scalable RESTful APIs using JavaScript.

In particular, NodeJS is asynchronous and event-driven. In the case of a RESTful API executed using NodeJS, this means that the runtime environment will listen for incoming client requests. It is important to note that NodeJS runs in a single thread, and the runtime environment does not spawn new threads for clients that connect to the API. Instead, NodeJS relies on application code that is mostly asynchronous in nature (i.e., does not contain blocking calls such as file read/write operations). NodeJS utilizes standard JavaScript callback hook semantics should a program require to make a blocking call [Nod22b]. In general, libraries are also written this way. Therefore, blocking calls are a rare exception in NodeJS rather than the norm. Interested readers may refer to the official NodeJS documentation for further details on blocking calls in NodeJS [Nod22a, Nod22c].

Either way, the non-blocking event-driven characteristics combined with the powerful underlying V8 JavaScript execution engine allow NodeJS to handle a large number of practically simultaneous client requests without any major slowdowns or problems [PNR<sup>+</sup>20].

Similar to Spring Boot, NodeJS also includes a package manager, called NPM (Node Package Manager). This additional software allows developers to load and install additional modules and use them in their projects.

Another benefit of using NodeJS for writing the RESTful backend of this project is that the developers would only have to use a single programming language for both their frontend and the backend applications. This means that there would be a shorter assimilation period when switching between the two parts, and the same set of tools could be used for the entire development process. These properties make NodeJS a solid choice for developing the RESTful API of this project.

### 2.3.6 The PHP-based Laravel Framework

PHP is a programming language that's widely known for running on web servers. The main motivation behind developing PHP was to enhance static web pages (i.e., websites that consisted solely of HTML and CSS files) with dynamic functionality such as outputs from database queries. However, modern PHP-based frameworks such as Laravel allow

developers to utilize PHP in more modern ways, for example, to develop RESTful APIs [Lar22].

Laravel may be used to develop APIs that only run on the server-side and serve as the backend for a larger distributed system. However, Laravel also allows developers to build full-stack applications. In addition, Laravel can run in a docker container, which may be profitable for various situations. In contrast to maven, for example, Laravel does not require developers to spend much time with changing configuration files, as Laravel can be used out-of-the-box.

However, Laravel seems to be much more complicated to get started with, as there is an overwhelming number of products put together under the Laravel product name. Further, the framework claims to be highly scalable, and the website states that:

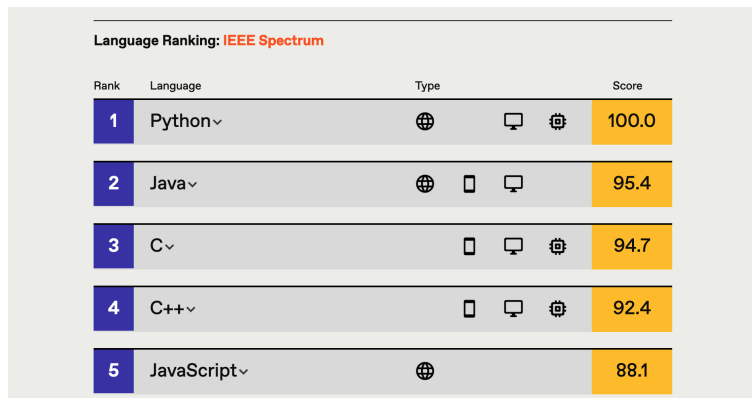
"Laravel is incredibly scalable. Thanks to the scaling-friendly nature of PHP and Laravel's built-in support for fast, distributed cache systems like Redis, horizontal scaling with Laravel is a breeze. In fact, Laravel applications have been easily scaled to handle hundreds of millions of requests per month." [Lar22]

This number may sound impressive at first, but upon further inspection, "hundreds of millions of requests per month" (aside from being a very vague statement) breaks down to about forty requests per second, which is a minuscule number compared to the thousands of requests that NodeJS claims to handle per second. Other authors also suggest that Laravel scales poorly compared to NodeJS [PNR<sup>+</sup>20].

$$100,000,000 \text{ requests} / 30d / 24h / 60m / 60s = 38.580 \text{ requests per second}$$

The Laravel framework uses the MVC pattern. However, as mentioned earlier, this project aims to implement the MVVM pattern. The differences between the two patterns and why MVVM is preferred for this project are discussed in Section 2.3.2 of this thesis.

In addition, Laravel is PHP-based, which some developers might see as a positive thing. However, many developers would also argue that PHP is a rather outdated language, despite receiving regular updates. In general, PHP seems to be a less popular language amongst developers compared to Java and JavaScript [KD17]. Further, the author of this thesis does not have any previous experience working with the Laravel framework. Therefore, the learning curve may be quite steep, and it could take a long time to get a functional prototype ready. Overall, Laravel and PHP do not seem to be a good fit for this particular project.



Rank	Language	Type	Score
1	Python	Web, Desktop, Server	100.0
2	Java	Web, Mobile, Server	95.4
3	C	Mobile, Server, Embedded	94.7
4	C++	Mobile, Server, Embedded	92.4
5	JavaScript	Web	88.1

Figure 2.2: A screenshot of the results of the IEEE spectrum ranking of 2021 [Spe21].

### 2.3.7 Python and Flask

Python is another programming language that has gained much popularity among developers throughout the last years [KD17]. As Figure 2.2 illustrates, the IEEE Spectrum website even concludes that Python was the most popular programming language of 2021, with Java coming in second place [Spe21].

Therefore, it makes sense to investigate Flask, a popular web framework for the Python programming language. Further, it has to be evaluated whether Python and Flask are viable options for developing the project discussed in this thesis.

First, it is important to note that Flask itself does not include all the features that are commonly needed for building modern RESTful APIs. Due to this property, Flask is often referred to as a micro-framework, which means that it offers a slim core component that implements the most basic features needed for a web application. Flask alone can perform basic tasks such as handling incoming user requests, sending responses, markup cleanup, and data serialization. In addition, the Jinja Template engine can be used to conveniently send HTML responses to clients. Template engines hold a user-defined template of, for example, an HTML page where certain parts of the page got replaced by placeholder strings. The template engine can then replace these placeholders with certain concrete values to generate the final HTML response. Aside from these basic features, programmers have to either supply their own functionalities or install small extensions that handle additional features such as accessing database tables.

The principle behind Flask is very reminiscent of microservices, which are a collection of extensions where each service fulfills exactly one particular task (e.g., fetching data from a MySQL database). Together, many small services form a larger application. So, in a sense, microservices are not like full-size modules (e.g., a module that can handle all MySQL operations) that can be installed using a package manager such as NPM. Instead, they are a collection of small extensions where each service offers one particular function to extend an application. Using microservices has a range of advantages. Mainly, the individual services are loosely coupled, which allows each service to be developed,

tested, and maintained in isolation from the other services. Furthermore, services can fairly easily be swapped out for others should developers find a version that better suits their project requirements. Often, applications that want to use these microservices communicate with the service via a (RESTful) API.

Due to the simplicity of the Flask core, it is said to be very easy for new developers to learn how to implement RESTful APIs using Flask. Furthermore, Flask is a popular framework for Python, which means that it should be easy to find learning resources. Due to its simplicity, the learning curve should also be fairly flat. In addition, Flask extensions are typically open-source, which means that the code could be maintained should the original developers abandon their code. In addition, custom extensions can be written to achieve the exact behavior that's required for a particular project. However, the downsides of Flask are that many required features are not part of the Flask core, and some extensions may be of poor quality. Therefore, it might happen that certain features that are available in other frameworks, such as Spring Boot, may have to be completely re-invented to work with Flask.

Similar to PHP and Laravel, the risk of using a framework that the developers of this project have never used before is too high considering the time and budgetary constraints of the presented project. Therefore, Flask is not a viable option for this project.

### 2.3.8 WCF and ASP.NET

Microsoft offers a similar solution to the Spring Boot framework for Java in regards to creating a RESTful backend for web applications. The Windows Communication Foundation (WCF) is a framework for developing distributed applications. As mentioned earlier, every distributed application can be made RESTful as long as it follows the set of rules discussed above. Therefore, the WCF can also be used to develop a REST API using the Microsoft .NET framework. However, this option is only mentioned for the sake of completeness. In contrast to Spring Boot, the community behind WCF seems to be smaller, and the developers of this project have not had the chance to work with WCF or ASP.NET before, yet. Therefore, learning the fundamentals of WCF would be time-consuming, and the preparation period prior to the implementation phase would be much longer. Therefore, this option is not investigated any further.

### 2.3.9 Angular

Angular is one of the most popular frameworks for building single-page frontend applications using modern web programming languages such as TypeScript, HTML5, and CSS3. A simple description of TypeScript would be that it is an extension of JavaScript that, besides other features, adds strong typing information to standard JavaScript. A compiler translates TypeScript code to standard JavaScript so that the translated scripts can run in any modern web browser without the need for extensions on the client-side.

As mentioned, developers utilize Angular for building single-page applications. A classic website typically consists of multiple HTML pages, one or multiple style sheets, JS files,

and additional resources such as images. When a user navigates to the website, they typically get greeted by a landing page (i.e., home-page), and from there, users click on links to navigate the website. With each link click, the browser requests a new HTML file and all associated resources from the webserver. However, single-page applications do not work this way. As the name implies, they consist of a single page, and when users click on links, the browser replaces the content of the current page with the requested content without fetching another HTML document. This approach reduces loading times and has a positive effect on the overall User Experience [Sak19].

Angular is component-based, and applications made with Angular typically contain multiple components. Components take over the role of the previously mentioned multiple HTML documents, and they get loaded by the single-page application when needed (e.g., when the user clicks a link in the app). Each component consists of a TypeScript class that models the component's behavior, an HTML template that describes the content, and a style sheet, that holds information about the component's layout and style. The HTML templates can use placeholders that the TypeScript file of a component can fill in at runtime. Angular automatically updates the HTML file should the underlying data change, for example, when a user updates the text in an input field. The encapsulated approach that Angular follows allows different people to work on the same component simultaneously, and it allows different teams to work on different components.

Angular supports the use of numerous helpful modules that programmers can add to their projects as needed via the dependency management feature. However, it is important to mention that adding modules, especially ones that are not part of Angular, is always associated with risks. Missing or incorrectly referenced dependencies can quickly break a project. The Angular CLI tools are helpful in that regard, but configuring Angular, especially after updating modules, is almost always a time-consuming task. Therefore, it will be best to avoid external modules in this project unless there is no alternative. Angular supports using a UI toolkit that allows developers to quickly and effortlessly craft a visually pleasing and uniform look and feel throughout their entire application. However, it is also possible to use external UI packages such as Bootstrap or define a custom User Interface style.

As Angular includes numerous features and configuration options, the learning curve can be very steep for new developers. However, Angular is fully open-source, well documented, and supported by a large community. Therefore, training material that discusses every single feature can easily be found. As with Spring Boot, the developers of this project have already conducted multiple complex projects that utilize Angular, and therefore, the training period would be rather short.

### 2.3.10 Vue.JS

Compared to Angular, Vue.JS is a more light-weight open-source component-based library that supports building modern single-page web applications. However, instead of using decorators and custom file extensions like Angular, Vue.JS uses custom HTML tags to define the three main blocks of each component. Those three blocks are the same as in Angular, namely the template, script, and style. Even though Vue.JS uses a different syntax, keywords, and file structure, the underlying principles mostly remain the same, at least from the developer's point of view. Most importantly, Vue.JS also supports using the MVVM pattern, which is an important aspect of this project. Depending on the type of binding, Vue.JS can also update the view if the underlying data changes, for example, due to user input.

Similar to Angular, Vue.JS is also backed by an active open-source community. It is easy to obtain training material, and the learning curve is much flatter compared to Angular. Overall, Vue.JS seems to offer developers more freedom of choice regarding how they want to write their code. First, Vue.JS allows developers to choose how they want to organize their file structure, and second, developers may use ECMAScript 6 or TypeScript during development [Woh18]. However, so far, the project team has only built one small single-page application using Vue.JS. Therefore, more training would be necessary before implementing this more complex project.

### 2.3.11 React

Depending on the source, React (sometimes also referred to as ReactJS) is either the most popular or the second-most popular modern frontend UI framework for web-applications [Woh18, Agg18, Sak19]. Regardless of its exact positioning in the ranking, ReactJS is undeniably one of the top three frameworks together with Angular and Vue.JS.

React drastically differs from the two previously discussed options in some ways. However, all three frameworks and toolkits share the property of being component-based. While Angular used TypeScript and Vue.JS used ECMAScript 6 or TypeScript to describe components, ReactJS utilizes an extended version of JavaScript called JSX. This description language combines elements from HTML and JavaScript and allows programmers to define small, simple, and reusable building-block-like components within the framework. These components are fully encapsulated, and they manage their own internal state. Programmers define an interface that allows components to receive input values from their parent components, and the end-product is a component that ReactJS can place in the DOM of a website. Note that, unlike Angular components, child components in React can not pass data back to their parents without third-party extensions [Sak19]. Besides being modular and reusable, React components manage their own state. This means that similar to Angular, when the underlying data changes, the UI reflects these updates and communicates them with the user. React also employs the Node Package Manager to manage dependencies and to allow developers to add external packages to their projects.



The good encapsulation and modular nature of components within React and the component's ability to reflect changes in their underlying data storage are two very important aspects of this project. The final app could, for example, hide certain less relevant components on smaller displays or arrange them in a different order if necessary.

A positive remark has to be made about the React documentation. It is extremely comprehensive without being overwhelming, and it offers two styles depending on the reader's preferred learning style. It either provides users with detail-rich textual descriptions and occasional examples or a full learning-by-doing course that contains various small tutorials that users can follow. These measures help keep the learning curve flat, and learning the core principles of ReactJS should not take up a significant amount of time.

## 2.4 UI/UX Design

It is relatively easy for individuals to intuitively recognize a visually pleasing UI layout when they see one. However, aesthetics and why humans find certain designs beautiful is not as intuitive to describe, and it is a broad topic that spans various scientific disciplines such as psychology, philosophy, art, neuroscience, and also information technology [SW18]. A good User Interface can make working with a system seem intuitive and easy, and users who find a graphical UI visually pleasing often associate higher usability with the system [Tra97]. However, a system should focus on achieving a good balance between perceived beauty and usability, and designers should not sacrifice usability and ease-of-use in favor of improved aesthetics.

### 2.4.1 Psychological Aspects in UI/UX Design

The importance of a good UI design is undeniable, as Schmidt and Wolff summarize in [SW18]. They mention that a good interface does not only increase the perceived usefulness of a system but also reduces the user's error rate (e.g., when making inputs in a system). Further, users are more likely to enjoy interacting with a system if the User Interface is visually appealing and well designed. Lastly, a system's credibility is also often judged by its looks, and potential buyers are more likely to use an e-commerce system that they perceive as well-designed [SW18].

In this context, the use of certain colors may be a good choice for one setting, while it might be inappropriate in another. One example is the color blue. Yang et al. conclude that the color blue is not a good choice for food-related topics, as it is often perceived as unappetizing [YMX<sup>+</sup>16]. In addition, designers should not use too many different colors. Instead, a limited palette should be employed [SW18]. Lastly, it is important to note that cultural differences and the user's sex make a difference regarding how users perceive certain color combinations [SW18].

Further, Schmidt et. al. identify symmetry and visual complexity as two more leading factors for good UI design [SW18]. Axis symmetry shows a positive correlation with the

perceived attractiveness of websites. However, their literature analysis concludes that it is currently unclear whether users prefer complex or simple user interfaces.

### 2.4.2 Design Principles for Web-Interfaces

What is perceived as good-looking changes over time, and everyone has their personal preference. Therefore, this section does not focus on concrete design philosophies such as flat design. Instead, it summarizes a few key methods one can employ when building web interfaces that will most likely have to function across varying screen sizes and resolutions. As mentioned earlier, one of the key requirements of this project is that the proposed software solution must function flawlessly across a broad range of displays. The UI should be usable on small screens such as built-in HMI displays as well as large PC monitors, for example, in a centralized control station.

Therefore, the design of the final solution will have to adapt to changes in screen sizes and resolutions and still be usable. In their work, Margea et al. summarize a few possible User Interface design philosophies that yield web interfaces that scale according to the display size [MMVH17]. In their work, the authors specifically investigate mobile web trends. They mention that today many users visit web pages using a smartphone with a relatively small display compared to a full desktop computer. Therefore, website developers have to ensure that their web interfaces scale well across various classes of devices. Margea et al. identify a few leading trends that designers may follow to achieve this effect.

Mobile-First is such a design philosophy. In summary, this theory states that website limitations imposed by small screen sizes force designers to focus on content and functionalities that matter. In turn, this philosophy should also help users with larger screens to focus on the content and not get lost in a plethora of features. This philosophy emerged early in the history of mobile Internet browsing in 2005. At that time, more often than not, a mobile website was an afterthought to an already existing desktop representation. With mobile-first, developers shift their focus and build the mobile website first. Then, they ensure that the mobile version also looks and feels good on larger screens rather than the other way around. Margea et al. also mention that website developers should still follow the mobile-first approach, even when building responsive websites.

In responsive web design, the server supplies each client with the same website, stylesheet, and resources, regardless of the client platform, screen size, or resolution. Responsive web design is a client-side approach where the stylesheet contains various rules that only get executed by the client's browser if certain criteria are met, for example, on certain screen sizes.

In contrast, dynamic serving is a server-side approach. Here, the server dynamically creates different versions of the HTML document according to the client's needs. For that, the server often uses the user-agent to determine how to generate the final HTML document for each client.

In graceful degradation, devices with large screens receive all the features. Then, with shrinking screen sizes, more and more features are omitted to ensure that the most

important ones are still functional, even on smaller devices. In the context of this project, the full desktop and tablet representations of the final frontend UI could contain all the features, while the mobile version could serve as a simple viewer that lets operators see the network graph but not modify it.

Aside from these four design philosophies, Margea et al. discuss several other ones in more detail [MMVH17]. However, this thesis only mentions these four, as they seem the most relevant for this project.

### **2.4.3 This Project's Approach to UI/UX Design**

As mentioned in Section 3.1, this project uses Angular's built-in material UI elements for building the frontend. Therefore, the developers do not have to worry about hand-crafting stylesheets to style every single element on the page. Design-philosophy-wise, the project follows the responsive design principle and uses graceful degradation when it is no longer viable to make elements smaller or re-arrange them to make them fit on a smaller screen.



# Project Details and Implementation

This chapter examines the planning and implementation phase of the project discussed in this thesis. First, it introduces the project goals by describing the most important features and desirable non-functional requirements. Furthermore, the chapter also discusses features that are explicitly not part of this project, as well as possible extensions and enhancements that could be implemented in future projects.

## 3.1 This Project's Technology Stack

After analyzing state-of-the-art tools, patterns, languages, and frameworks, the final technology stack for this project looks as follows:

1. Backend

This project's backend application will be written in Java using the Spring Boot framework in conjunction with Apache Maven to manage the project's dependencies. This technology was chosen due to the previous projects conducted using Spring Boot and the possibility to very quickly obtain a functional prototype using a pre-configured project starter.

2. Backend testing and quality assurance measures

The backend will be tested using the JUnit test framework and black-box testing. For that purpose, a test plan will be created that can be found in Appendix A of this thesis. For this project, all of the test cases must not return any errors. Code-Coverage metrics will not be employed in this project. However, Sonar-Lint will be used to increase the overall readability of the code.

#### 3. Frontend

As discussed by [Woh18], the three UI frameworks presented in this thesis offer very similar features. Furthermore, [Sak19] compared the performance and build-sizes of a project that was built using all of the three discussed frameworks. In conclusion, ReactJS and Vue.JS exhibited the smallest final builds, and Vue.JS was the fastest framework to render the application used in the author's project. Regardless, the frontend portion of this project will be developed using Angular solely due to the experience gathered in previous projects and existing time constraints. Furthermore, Angular comes with built-in styles and UI elements, which this project will use to omit the need for external packages such as Bootstrap.

#### 4. API testing

All REST API tests will be conducted using postman.

#### 5. Development tools and version management

The IntelliJ IDE will be used to develop the backend, as this IDE offers integration options for the tools mentioned above. The frontend will be developed using Visual Studio Code. A custom Git repository will be used to track project changes and manage releases.

## 3.2 Backend Features

### 3.2.1 Functional Requirements

The main goal of this project was to implement a RESTful API that establishes a connection to an existing database and serves as a proxy for clients to interface and interact with the data stored in the database. Therefore, implementing support for the main CRUD (Create, Read, Update, Delete) operations was an absolute necessity for the backend application of this project. Furthermore, there exist four entities that the backend has to manage, namely devices, ports, safety variables, and links. Each entity is represented by a table in the database. The CRUD operations had to be implemented for each of the entities.

Aside from requesting all data from the backend server, the backend should also enable clients to define filters that allow them to narrow down the list of results returned by the backend application. This filter option helps reduce the amount of unnecessary data that has to be transferred over the network, and it also helps speed up algorithms that have to iterate over all elements of a result set.

Next, the backend application should create graph representations of the entities in the database. In this network graph, the nodes represent the device in the database, and the edges that connect the nodes correspond to logical or physical connections between the devices. Such a connection can be a simple network connection such as an Ethernet cable that links two devices, but it could also be a safety connection that links a safety

sensor to safety logic or an actuator. Regardless of the type, the backend server should correctly identify all connections between the devices and include them in the graph.

Another key feature that was part of the original project description is that the backend server should react to changes in the database whether they be made using the RESTful API or externally (i.e., by sending SQL queries directly to the database). The backend server should then update its caches and graph representations if necessary. Ideally, the backend application should also notify all active clients of the changes made in the database so that the clients can update their local representation of the data.

Some operations in the backend application may take several seconds to finish. One example is the generation of a large network graph. Therefore, the backend application should employ caches that store the results of such long-running tasks. In addition, the application must ensure that the caches either get updated or purged should the underlying data storage change.

Even though user authentication was not part of the original project description, a rudimentary authentication mechanism was implemented using JSON Web Tokens (JWTs). This was done due to the state-of-the-art nature of this project, as nearly all APIs provide some sort of user authentication to prevent unauthorized or malicious parties from making potentially harmful requests to the backend of a distributed application. Therefore, the server also creates and verifies JSON Web Tokens that clients have to submit with each request they make to a secured API endpoint in the backend.

### **3.2.2 Non-Functional Requirements for the Backend**

The backend server should not take longer than five seconds to respond to any user request. If a request takes longer to process, the server should keep the connection with the client alive so that the delay does not cause the connection to close.

The implementation of the backend application should be maintainable and open for future adaptations, for example, a more sophisticated security implementation.

### **3.2.3 Features that are not Part of this Project**

The entities managed by the backend application are stored in a database. The creation and maintenance of that database are not part of this project. Instead, an external device exploration program implemented in another project handles those tasks. This exploration application detects changes in the network structure and updates the database by making direct queries to the database server.

Further, ensuring the consistency of the stored information in the database is also not part of this project, which means that the database has to maintain a consistent data store through triggers and foreign key relations. However, the backend application presented in this thesis implements rudimentary integrity checks and operations that cascade delete commands to linked objects, for example, ports of stored devices. However, the backend server does not verify whether foreign keys reference valid entities in the database. The

decision to omit these additional consistency checks was mainly made for performance reasons, as modern database management systems are well optimized for such tasks.

Further, user authentication, data encryption, and other security features were not part of the original project description. However, the project team decided to add a rudimentary authentication mechanism to protect the API from malicious clients. Apart from that, no other security features were implemented in this project.

#### 3.2.4 Possible Future Backend Adaptions

The exploration mechanism mentioned above could use the REST API that the backend server provides for storing, updating, and deleting entities in the database instead of directly accessing the database entries. Doing so would make the change-detection mechanism easier to implement, and it would also ensure consistent entity validation across all clients and services that access the data. The current implementation of the backend application already contains endpoints for creating, updating, and deleting all entities. However, more sophisticated consistency checking and entity validation would be required should the exploration program use the API in the future.

Next, a future project could strengthen the aforementioned rudimentary security and authentication features of the current backend implementation. It is important to mention that the project team agreed that users would not have to actively input a username and password before they could use the frontend application of this project. A possible approach that completely omits the need for users to input their credentials could use asymmetric key authentication. A future project team could extend the database to store the public key of each client device that wants to access the backend API. During startup, the clients would have to request a challenge from the backend server that each client encrypts using their private key. If the backend server can then decrypt the challenge using the client's public key, the client may access the secured API endpoints.

Another possible future addition could be a feature that automatically notifies clients of changes in the underlying data storage. This feature was omitted in this project because the API had to be RESTful, and one major component of the REST design philosophy prohibits the server from storing information about the clients. As mentioned in Section 2.3.1, the communication between the clients and the server must be stateless in a RESTful API. However, a notification feature like this would require the server to keep some information about each client in order to notify all of them of changes, thus resulting in a backend application that is not truly RESTful.



## 3.3 Frontend Features

### 3.3.1 Functional Requirements

The frontend application should be able to establish a connection to the RESTful API implemented on the backend server. Most importantly, the frontend part of this project must not access the database directly. Instead, the frontend web application should request representations of the resources whenever necessary. The backend server may or may not cache the resources, and the frontend client should not have to know about the internal structure of the API or whether or not the backend uses caches. The frontend application should be able to transmit modified representations of the entities it previously received from the backend should the frontend client want to update a resource.

The frontend client should display a list of devices stored in the database. Next, the frontend web application should also display a detailed description of the devices and links stored in the database.

Further, the frontend web application should offer clients the possibility to review and update certain values of devices. The website should also perform basic validity checks that ensure that certain fields, for example, a device's name are not empty. However, similar to the backend, the main validity and consistency checks will be performed by the Database Management System.

The frontend application should display a graph that it requests from the backend server. As described above, the nodes in the graph correspond to the devices in the network, and the edges between the nodes describe connections between the devices. The graph representation should allow users to effortlessly distinguish physical (network) connections from logical (safety) connections. Users should have the option to choose a device from the aforementioned list of devices (e.g., by drawing safety-relevant edges in a different color). The frontend application should then update the graph so that the graph shows the incoming network connections of the selected device.

The frontend server should be able to ask the backend application to check whether the underlying data storage has changed. As mentioned above, it was originally planned that the backend server automatically notifies active clients of changes. However, this feature had to be changed in order to ensure that the entire application follows the original set of REST design principles. These update requests should happen automatically and periodically. However, the frontend UI should also contain a button that lets users manually update the data.

Since the backend part implements a rudimentary JSON Web Token authentication mechanism, the frontend client will have to authenticate itself in order to access the secured API endpoints. For that purpose, the frontend application contains hard-coded login credentials that it uses to request a JWT from the backend. The frontend web application then supplies that token with each request and automatically takes care of token-renewal when necessary.

Lastly, the frontend website should implement a responsive design that works across a range of display sizes and resolutions. Ideally, all features should be available on all displays commonly found in modern devices. However, the frontend application should employ graceful degradation should a display be too small to fit all the content.

#### 3.3.2 Non-Functional Requirements

The frontend User Interface should respond to user input without any noticeable delays. Further, it should be easy to operate and not contain many different pages like classic websites. Instead, the most important features must be accessible on a single page without having to navigate any menus. Lastly, the design should focus on simplicity rather than visual splendor.

#### 3.3.3 Graph Requirements for this Project

One goal of this project is to make the final solution as modular as possible. Therefore, all graph drawings will be generated by one or multiple concrete graph drawing modules that each implement exactly one algorithm.

The project should contain two graph representations of the physical network structure found in a fictional test factory. The first type of graph displays the network using a directed graph where the currently selected device sits in the center (see Figure 2.1 A). The graph drawing algorithm then places adjacent nodes, which may represent safety sensors, TSN-capable switches, or other IT equipment around the center node.

Next, the finished project should include a graph that follows the inclusion convention as described by Di Battista et al. [DBETT94]. Such drawings represent hierarchies using nested boxes, and the goal in this project is to utilize them to outline which safety sensor or switch controls which devices in a factory. Here, the outermost box represents a safety device, and the inner boxes represent the hierarchy of children controlled by that safety device (see Figure 2.1 B).

The motivation for adding multiple graph modules comes from the difficulty for machines to decide which representation makes the most sense in a certain scenario. Human operators have undergone special training, and they may want to visualize data in a certain way. Including multiple graphs allows operators to select the representation that works best for them in a given situation. The high modularization allows other engineers to effortlessly add new graph algorithms to the existing system.

#### 3.3.4 Possible Future Frontend Adaptions

Possible modifications to the frontend web application could include adding a feature for updating links and ports. In its current state, the frontend application only allows users to update devices. In addition, a future project could implement a more interactive graph that lets users dynamically drag edges from one node to another to reconfigure the safety connections in the network. However, adding a feature like that would require more

sophisticated validation mechanisms and integrity checks to be added to the frontend application. Nevertheless, the backend API already offers most of the features required for implementing such a function.

Another possible adaption to the backend could include a more appropriate user-authentication mechanism as described above. Alternatively, users could authenticate themselves using a unique username and a custom password.

## 3.4 Backend Application Implementation Details

### 3.4.1 Backend Structure

The backend project implements a three-tier architecture that incorporates a presentation, a business logic, and a data layer. Clients utilize the RESTful API implemented in the backend server's presentation layer to send requests to the backend application. Clients and the server exchange data by using Data Transfer Objects (DTOs). These simple objects only carry data and do not contain any business logic. Clients use JSON entities to send complex objects to the server. When the backend receives such an object, it internally de-serializes it and generates a new Java object based on the information in the DTO. Analogously, the server sends out DTOs with the requested entities. Apart from DTOs, clients can also use URLs and HTTP headers to pass data to the backend along with their requests.

Behind the RESTful presentation layer sits the business-logic layer, also referred to as the service layer of the backend application. The classes in this service layer implement the actual computational features that the backend application provides for its clients. Figure 3.1 illustrates how the layers communicate when a client requests a resource, for example, a network graph.

As mentioned, the backend server accepts client requests using its presentation layer. There, it parses the request and performs some basic validity checks. Those validity checks are not complicated, and they ensure that the supplied data meets the next layer's input contract to avoid producing errors. The presentation layer should be as straight-forward as possible, and there should be no logic implemented in it apart from simple input parameter inspection mechanisms, such as null-checks. Therefore, the presentation layer only analyzes whether some of the parameters are null and then passes them on to the presentation layer for further inspection. In the example use-case presented in Figure 3.1, the presentation layer instructs the service layer to determine whether the client-supplied device exists in the database. The business-logic implementation does not directly access the underlying data storage. Instead, it passes the request to the next layer in the chain, which is the data layer. In this backend application, the data layer is the last link in the chain. However, it would also be possible to implement additional data integrity and validation layers between the service and the data tiers. However, as the validation in this project is mostly done by the external database, that data-integrity layer was omitted. Using a separate layer for accessing the data storage allows the backend application to

### 3. PROJECT DETAILS AND IMPLEMENTATION

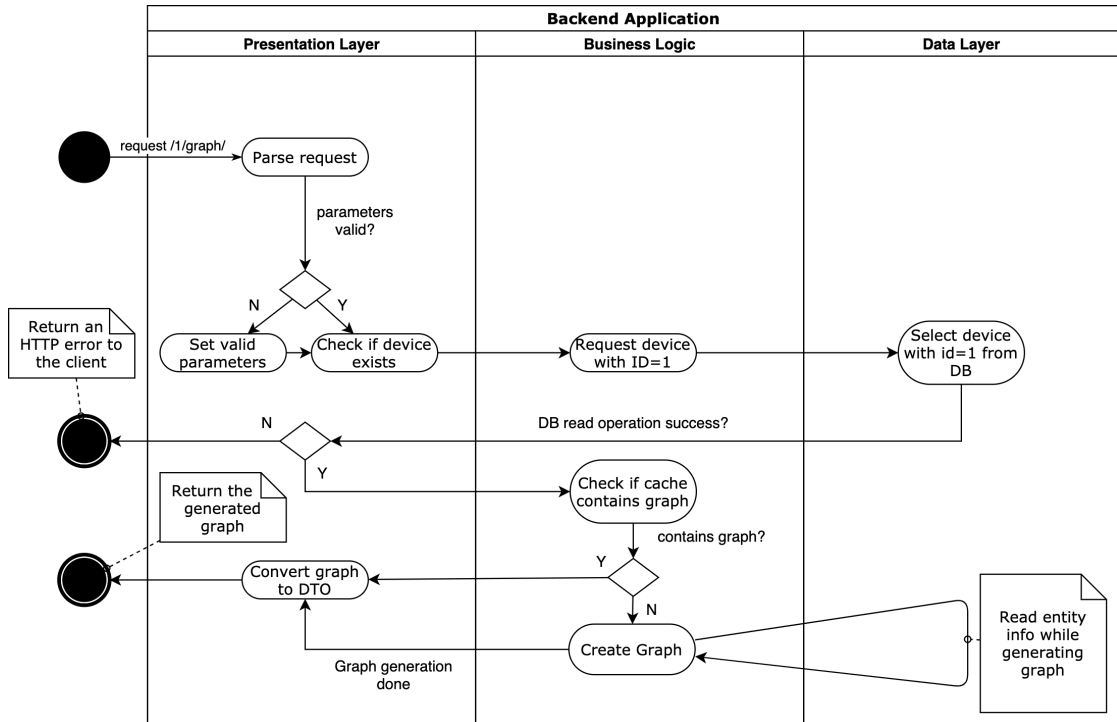


Figure 3.1: This activity diagram illustrates how the backend server handles user requests. Further, it outlines each layer's responsibilities and the communication between them.

use any other persistent data storage method, for example, simple CSV files, by swapping out the data layer without having to modify any of the other tiers.

The data layer then determines whether the requested device is present in the database. If the device does not exist in the database, the data layer raises an exception. The business logic tier passes that exception on to the presentation layer, which, in turn, hands it over to the requesting client. If the data layer finds the device in the database, it returns an object that contains the requested information to the service layer, which passes it on to the presentation layer.

Next, the presentation layer requests a graph representation from the business logic tier in a similar fashion. The service layer requests the graph representation from an internal cache. If there is a cache miss, the service layer generates the graph and communicates with the data layer in the process of doing so. The result is then returned to the presentation layer and successively to the client.

This simple use-case illustrates how the tiers of the backend application work together when handling requests. Using a multi-tiered approach usually introduces additional communication overhead and a larger number of classes than would be necessary. However, this technique also allows for easier future adaptations and replacement of layers should that become necessary. Either way, the client does not have to know about the internal

structure of the backend server to make requests, and clients communicate with a single well-defined interface implemented in the server's representation layer. Due to the multi-tiered nature of this application, future versions could add other communication endpoints besides the REST interface, for example, a GraphQL endpoint.

### 3.4.2 Caching

As illustrated in Figure 3.1, the backend uses a custom cache implementation that allows the server to respond to common user requests quickly. An assumption in this project is that the underlying hardware and the network structure do not change very often. However, a large number of clients might request many graphs per hour. Therefore, it would be inefficient and unnecessary to generate the same graph network for each client as long as the data in the database remains unchanged. Instead, it is much more efficient to generate a graph once when the first client requests it and then store it in a local cache before returning the result to the client. Subsequent client requests will then be served using the cached instance as long as the cache does not become invalidated by a database change.

As mentioned in Section 3.2.1, clients should be able to update their representation automatically should the underlying data in the database change. Instead of keeping a local cache, periodically requesting all entities, and then checking whether any of the entries changed, clients can request a hashcode from the server that lets them quickly determine whether their local data is still up to date. Requesting and comparing a single integer is much faster and also easier on the network than the aforementioned alternative. Therefore, it was a goal to write the backend caches in a way that allows for quick updates and checks for updates.

As these checks should work for various complex data types, the decision was made to implement a generic hash-based cache data structure. This custom cache implementation contains two standard Java maps. One map stores uniquely identifiable elements, for example, devices, where the element's id acts as the key in the map. The second map stores the hash of an object alongside its id.

This technique allows the backend server to quickly determine whether a complex object has changed without having to inspect any of its internal variables. Furthermore, the check only requires a single integer-compare operation.

This approach might be slower when building the cache, as each element's hash value has to be calculated before storing it in the cache. However, inspecting the entire cache for changes becomes much faster, as well as replacing updated elements in the cache. As discussed, write operations do not occur as often as read operations in this project.

```
1 public class Cache<T, E extends IDBearer<T>>
2 {
3     private Map<T, E> cachedElements = new HashMap<>();
4     private Map<T, Integer> hashes = new HashMap<>();
5
6     public boolean cached(T id) {
7         return cachedElements.containsKey(id);
8     }
9
10    public synchronized E updateOrAdd(E element)
11    {
12        int hash = element.hashCode();
13
14        if(cached(element.getID()))
15        {
16            if (hashes.get(element.getID()) != hash)
17            {
18                cachedElements.replace(element.getID(), element);
19                hashes.replace(element.getID(), hash);
20            }
21        }
22        else
23        {
24            cachedElements.put(element.getID(), element);
25            hashes.put(element.getID(), hash);
26        }
27
28        return get(element.getID());
29    }
30
31    public synchronized void updateOrAddMultiple(Collection<E> elements)
32    {
33        for (E e : elements)
34            updateOrAdd(e);
35    }
36
37    /* Other functions and comments omitted */
38 }
```

Listing 3.1: This listing outlines the most important implementation details of the custom cache structure used in the backend project.

#### 3.4.3 Graph Generation Algorithm

As shown in Figure 3.1, the backend server first determines whether a requested graph is already present in its internal cache. If that is not the case, the service layer calculates the requested graph and returns a representation to the client. This representation does not contain any concrete details about the devices (i.e., the nodes in the graph) and the links (i.e., the edges between the nodes). Instead, the returned representation object contains a list of nodes and a list of edges. These two lists hold the IDs of the objects in the graph without additional details. This approach ensures that small updates in

the database, such as a changed device name, do not affect the structure of previously generated and cached graphs. However, more substantial changes such as the deletion of a node or a link cause the cached graphs to become invalid. The backend server's periodic update detection, described in more detail below, invalidates the cached elements should such an operation occur.

The `GraphService` class employs two temporary lists and three stack structures when generating a network graph for a given center node and a given depth. The two temporary lists hold the edges and nodes that will be added to the resulting graph at the end of the generation procedure. One stack contains the next nodes the algorithm has to consider, the second stack contains the already visited nodes, and the last stack contains the already processed edges. In a first pass, the algorithm performs the following steps as long as the stack that holds the remaining nodes is not empty and the algorithm has not reached the given depth:

1. Retrieve the next node from the stack.
2. Load all safety variables associated with the current node.
3. For each safety variable do:
  - a) Skip the variable if it either does not represent an input or if it is not connected to any other variable.
  - b) Otherwise, load the other variable and add a link to a special `SafetyEdgeList`.
  - c) Add the device linked to the other variable to the stack of next nodes if it is not in the list of already added nodes.
4. Remove the current node from the stack that holds the next nodes to visit.
5. Add the current node to the list of added nodes

After this first pass, all safety edges have been identified and added to a special `SafetyEdgeList`. This custom class utilizes a map structure to store information about the safety edges in a network graph. In this project, the resulting graph should not contain multiple safety edges between two nodes. This is due to the fact that some safety devices may communicate using many safety variables that would lead to a large number of edges running between two nodes. To prevent this problem from happening, the graph generation algorithm summarizes the involved variables and replaces multiple edges running between two nodes with a single edge between the same two nodes. In addition, the newly generated edge must also contain information about the involved variables and their directions. When all edges have the same direction, the graph generation algorithm replaces the relevant edges with a new uni-directional edge. Otherwise, it introduces a new bi-directional edge between the two involved nodes. These considerations were pulled out of the graph generation part of the program and then placed in a custom

### 3. PROJECT DETAILS AND IMPLEMENTATION

---

SafetyEdgeList class so that the graph generation algorithm only has to call a single function that automatically summarizes edges if necessary:

```
1 public class SafetyEdgeList
2 {
3     private Map<String, SafetyEdge<Long>> safetyEdges = new HashMap<>();
4
5     public void addOrModifyExisting(SafetyVariable v1, SafetyVariable v2)
6     {
7         SafetyVariable i = v1.getInput() ? v1 : v2;
8         SafetyVariable o = v2.getInput() ? v1 : v2;
9
10        String key = o.getDeviceId().toString() + "," + i.getDeviceId().
toString();
11        String altKey = i.getDeviceId().toString() + "," + o.getDeviceId().
toString();
12
13        if(!safetyEdges.containsKey(key) && !safetyEdges.containsKey(altKey))
14            safetyEdges.put(key, new SafetyEdge<>(o.getDeviceId(), i.
getDeviceId(), o));
15        else if(safetyEdges.containsKey(key))
16            safetyEdges.get(key).addInvolvedVariable(o);
17        else if(safetyEdges.containsKey(altKey))
18            safetyEdges.get(altKey).addInvolvedVariable(o, EdgeDirection.Both
);
19    }
20
21    public List<SafetyEdge<Long>> toList() { /* ... */ }
22    public void clear() { /* ... */ }
23 }
```

Listing 3.2: This listing contains the custom SafetyEdgeList class. The addOrModifyExisting method summarizes edges and safety variables if necessary.

In a second pass, the algorithm now adds all necessary physical links and the remaining devices to the graph. Before starting the second pass, the graph generation algorithm places all previously added edges back on the stack that holds the remaining nodes, as these represent the minimum set of devices that must be inspected in this step. The program then also resets the depth before performing the following steps as long as the depth does not exceed the limit and the remaining nodes stack is not empty:

1. Retrieve the next node from the stack.
2. Add the current node to the temporary list of nodes to add to the graph.
3. Find all ports on the current device.
4. Find all links connected to any of the ports on the current device.
5. For each link do:
  - a) Skip the link if it already is in the added links list.



- b) Otherwise, find the port on the other side of the link and then subsequently the device that port belongs to.
  - c) If the device has not been visited before, add it to the stack of next nodes.
  - d) Add a new edge between the current node and the device to the added links list.
6. Remove the current device from the remaining nodes stack.
  7. Add it to the list of visited nodes.

After this second pass, the algorithm is mostly done computing the requested graph. All the data that has to be added to the graph is now stored in two temporary lists and the custom `SafetyEdgeList`. Next, the algorithm creates a new graph object and uses its `assemble` function to add all the calculated nodes and edges to the graph. Lastly, it returns the finished graph representation.

Aside from generating a graph for each device, the backend server application can also generate a graph that includes all edges and devices stored in the database. When generating the full network graph, the graph service leverages a simpler algorithm that retrieves all devices, links, and variables from the data storage layer. Then, the service layer adds all devices and network links to two temporary lists. Then, it iterates over all variables and adds them to the special `SafetyEdgeList` discussed above. Lastly, the graph generation algorithm assembles the graph using the two temporary lists and the `SafetyEdgeList` that contains all variable relations in the network.

Once done, the service layer caches the newly generated graph and returns it to the presentation layer of the application, which passes it on to the client who requested it.

#### 3.4.4 Database Update Detection

As mentioned, the backend application provides a mechanism that periodically detects changes in the database. The backend program uses a static global cache class that provides a separate cache for devices, links, safety variables, ports, and graphs. All other parts of the application that utilize caching reference the cache objects found in this class. As discussed in Section 3.4.2, a single integer compare operation suffices to determine whether any of the cached objects changed.

Whilst it would have been possible to leverage the singleton design pattern for the caches, the more complicated singleton implementation was dropped in favor of a simpler static class. However, static classes are less flexible and must be initialized at startup. It is possible to change this approach later, should the application, at some point, require lazy initialization.

The backend application periodically executes a new thread that requests all devices, ports, safety variables, and links from the database. Next, the thread determines whether the cache contains a stale entry that got deleted in the database. The thread then

empties each cache that contains at least one such stale entry. Then, it adds or updates all elements in each of the caches. This is where the cache internally employs the hash comparison technique described above. Then, the thread uses the new cache hash and compares it to the previous version. If these two hash values do not match for any of the four caches, the thread invalidates the graph cache and deletes all elements in it.

Clients can also request these cache hash values from the backend application and perform the same checks in the frontend. For that purpose, the update endpoint was added to the backend application. Clients can request all four hashes and a timestamp that describes when the last change happened. If clients keep a local copy of the hash values, they can quickly determine whether their local data, for example, a previously generated graph representation, is still valid.

## 3.5 Frontend Implementation Details

### 3.5.1 Frontend Application Structure

The frontend project is structured in a similar way to the backend application. However, the frontend web application only comprises two layers. The service tier handles all the communication with the backend API. It manages the URLs and URIs for all entities, creates HTTP requests, and performs the necessary communication steps. The Angular components form the second layer of the frontend application. As described in Section 2.3.9, each Angular component consists of three files. An HTML template defines the component's content, a TypeScript file implements a component's behavior, and a stylesheet defines how the component looks. Additionally, there can be a fourth file that defines component test cases. However, these will not be utilized in this project due to time constraints.

This project contains four main components, some of which comprise additional sub-components. The top navigation bar is the first component in this project. It is visible on every page of the application. In its current state, the frontend web application contains a single page, namely the dashboard component. However, additional components may be added in the future. This dashboard component comprises two sub-components. One of those displays a list of devices stored in the database, and the second sub-component renders the network graph generated by the backend application. The last two components are entirely functional and are not intended to be accessed by users. The login component contains hard-coded credentials that the frontend application uses to obtain a JWT from the backend server's authentication endpoint. The last component is a simple not-found page that the frontend application navigates to should a user input an incorrect URL.

The two sub-components on the application dashboard exchange events and data with their shared parent component. This procedure is important so that the dashboard can notify its sub-components should the underlying data change. If a user selects one of the devices in the device list on the dashboard, the device list sub-component sends an event to the dashboard component. The event contains the ID of the selected device. The

dashboard component then informs the graph sub-component of the changed ID. The graph sub-component then orders the graph service class, which is part of the frontend application's service tier, to request the graph for that device from the backend. Once it receives the data, the graph sub-component draws the network graph.

In addition, users can update the selected device using the graph sub-component. If they decide to do so, then the graph component informs the service layer and its parent component (i.e., the dashboard) of the changes. The dashboard then hides the graph and informs the device list sub-component of the changes. The list component then updates itself to ensure that the data displayed across all the sub-components in the dashboard of the frontend application remains consistent.

Lastly, the frontend application contains three simple components only used in pop-up dialogues. Two of these components provide users with additional information about a network or safety edge in the graph. When a user double-clicks an edge in the graph, the graph component detects the type of the double-clicked edge and then opens a pop-up window with the respective component. In that process, the graph component supplies the pop-up dialog with the affected edge ID. As the graph does not contain any concrete information about the edges other than the IDs, the component then requests additional data from the backend server before displaying it to the user. The third simple dialog component allows users to add new safety edges to the graph.

### 3.5.2 Graph Drawing Solution

A custom implementation yields the highest cost, as implementing, testing, and refining a robust and versatile automatic graph drawing algorithm may use a significant portion of the time available for implementing this project. However, a custom implementation offers the greatest freedom of choice, and it might be the best solution in specialized applications where no ready-made libraries that produce reasonable results exist. When considering the time and cost constraints of this project, a custom implementation may need to be simpler in nature compared to existing solutions.

External modules may go obsolete and become abandoned, which poses a risk and additional cost further down the line. In that event, another library would have to be found and integrated into the existing solution. Therefore, implementing a robust and maintainable custom graphing library is the most desirable option. However, due to the high cost associated with implementing, testing, and fine-tuning a custom graph drawing algorithm, the project team decided to utilize a ready-made network graph drawing library.

This project utilizes Vis.js [Vis22], a dynamic visualization library for JavaScript. The library implements a physics-based spring-embedder model for drawing network graphs, as described in Section 2.2.2. Besides, it offers CSS-based customization options that allow changing the appearance of the nodes and edges. Network graphs drawn with Vis.js are, furthermore, interactive. Users can add elements to a graph, drag them around, and delete nodes and edges interactively. In addition, the library also allows users to

pan the graph and zoom the view if necessary. However, these interactive features are disabled in the current version of the frontend application, as they added unnecessary complexity to the user interface. Apart from these interactive features, Vis.js includes support for user-generated events. One example is the double-click event that the graph raises whenever a user double-clicks an element in the graph.

The frontend application utilizes the double-click event to switch the currently active node and display edge details. If a user double-clicks on a node, the frontend application selects the double-clicked node as the active node and updates the graph accordingly. This behavior allows users to quickly switch between nodes in a graph. Once a user selects a node by either double-clicking it in the graph or by selecting it from the device list on the dashboard, the user can edit the selected node's details, such as its name and management address. Double-clicking an edge causes the UI to open a dialogue window that contains details about the edge. If a user single-clicks on or hovers the mouse cursor over one of the edges or nodes, the User Interface displays a tooltip that contains fewer details about the respective entity.

These interactivity features and the easy integration of Vis.js in the existing Angular project were the main reasons why the project team decided to employ Vis.js over other alternatives.

#### 3.5.3 Automatic Data Update Mechanism

As described above, the backend server periodically detects changes and generates a single integer value per updated cache. Clients can query this number to detect whether the data has changed since their last request. For this purpose, the frontend web application periodically sends a GET request to the backend server endpoint that supplies clients with the aforementioned hash values.

The decision was made to implement this automatic update mechanism in the dashboard component, as it contains the two sub-components that rely on automatic updates (i.e., the graph and device-list components) and benefit from them. The dashboard component contains an observable variable that emits periodic events. Users can adjust the frequency of the automatic updates from the web application's UI. By default, the frontend sends a request to the update endpoint every fifteen seconds. The server updates its caches every thirty seconds.

Users can change the auto-update frequency using the web application's top navigation bar. Here, users can select from a range of hard-coded values ranging from ten to sixty seconds. Alternatively, users are free to completely disable the auto-update feature and re-enable it at any later point in time. Once a user chooses an auto-update frequency from the top navigation bar, the navigation bar component sends an event to the update service of the frontend application. Each component that relies on automatic updates can subscribe to these events via the update service class. In this project, the dashboard component is the only one that receives these updates. Either way, the update service relays incoming events to all subscribed components as soon as it receives them. The

subscribed components can then adjust their auto-update frequency or disable the feature if so desired by the user.

Regardless of the chosen period, the frontend eventually requests an update from the backend (unless the feature got disabled) and then compares each of the hash values returned by the backend server to the locally stored ones. If any of these values change (e.g., due to an updated name in the devices table), the frontend redraws the currently active graph, and it also updates the device list it previously fetched from the backend server.

This simple implementation ensures that both the backend and the frontend applications act according to the RESTful architecture principle. Simultaneously, it also ensures that clients receive updates in a timely manner. The update frequency is adjustable in both the backend server and the frontend client so that operators can fine-tune the applications according to their requirements. The update period can be shortened should the app run in an environment with many changes. However, choosing more frequent updates generates overhead on the server, the network, and all clients. Therefore, careful consideration is required. A relatively short update period was chosen for this project, even though there are not many changes to the database during runtime. This was done to demonstrate the update feature without long delays.

## 3.6 Testing

### 3.6.1 Prototyping Environment

As this thesis discusses a web-based application, testing can be done with any device that offers an internet connection and a modern web browser. Therefore, functionality tests will be conducted on standard consumer PCs, laptop computers, and simple mobile tablet computers with varying screen sizes. The database server and the device exploration mechanism are not part of this thesis. However, the web application's backend server will also be tested on standard consumer hardware.

### 3.6.2 Backend Component Test Strategy

Component tests were conducted in the backend using the JUnit 4 test framework using the black-box approach. However, due to time constraints, tests were only created for the custom cache implementation, as it represents a fundamental feature of the backend application. Additionally, the graph service was tested in the backend using mock objects. None of the data-access layers were tested in the backend, as these features are implemented by JPA and Hibernate and assumed to be well-documented and tested. Lastly, the project does not automatically test any of the RESTful endpoints. Here, the project team decided to rely solely on manual testing.

#### **3.6.3 Backend Component Integration Strategy**

A vertical integration approach was chosen for this project, and the modules were integrated using a per-requirement approach. That is, a concrete requirement was implemented across all layers of the backend application and then integrated and tested as a whole. An example of a concrete requirement is that clients should be able to update existing devices in the database using a RESTful endpoint. Instead of implementing and testing each layer of the application separately, each requirement was implemented across all layers before testing the newly added feature. This was done due to the small project team and because each added feature yielded an executable and testable iteration of the application.

#### **3.6.4 Frontend Testing**

No automated frontend tests were conducted in this project. However, manual testing was done using different consumer devices such as smartphones, tablets, laptop computers, and desktop computers. Here, tests were conducted using all common operating systems and various common modern web browsers across all devices.

# Summary, Improvements, and Outlook

## 4.1 A High-Level Summary of the Implemented System

The finished project comprises two main systems that were implemented as independent applications, namely a frontend application and a backend server program. The frontend was implemented as a web-based single-page application using Angular. The backend application was implemented using Java and Spring Boot, a framework often used for creating RESTful APIs.

The frontend application comprises two logical layers. The module-layer contains all the necessary code for displaying the User Interface and handling user input. The service layer contains all code that provides services that implement features shared among the modules in the frontend application. These services include functionality such as managing automatic updates and communicating with the backend application.

Some modules in the frontend exchange information using data bindings and events. The device list and graph components utilize both approaches. Doing so is possible because both components share the same parent component (i.e., the dashboard). Data gets passed from one child component to the parent, and the parent then passes the data and events on to its other children. In addition, components that do not have a parent-child relationship (e.g., the top navigation bar and the dashboard) exchange data via services using a custom implementation of the observer design pattern.

The frontend services handle all the communication with the backend application. Before the frontend sends data to the backend, the data passes through a security implementation that adds the authentication header to outgoing HTTP requests. Then, the data gets transmitted to the backend application, where it passes through another security layer that checks whether a client is authorized to make certain requests.

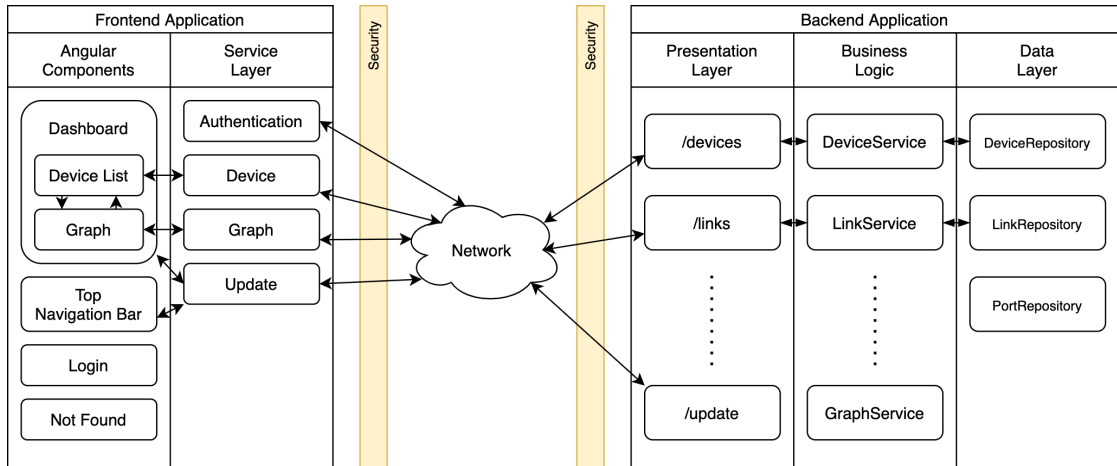


Figure 4.1: This block diagram gives an overview of the entire system. Note that the blocks and the connections between them are not exhaustive. They were only included for illustrative purposes.

Internally, the backend application is organized in three logical layers, as illustrated by Figure 4.1 and described in Section 3.4.1. Valid incoming requests reach the presentation layer, where the backend application performs only minor data-validation tasks before passing the request on to the business logic layer. The business logic layer implements the main features of the backend application. Examples of such features include data validation and the graph generation algorithm described in Section 3.4.3. For each entity in the system, there exists an interface in the data layer of the backend application. Database access is handled by JPA and Hibernate, and all database operations must flow through one of the data layer’s interfaces to ensure data integrity and consistency.

## 4.2 Added Value, Lessons Learned, and Outlook

### 4.2.1 Implemented Requirements and Omitted Features

Most features that were planned for this project got implemented in the final product. However, some features had to be modified or omitted entirely to meet certain constraints and limitations.

All database-related features got implemented, including establishing a database connection, reading entity values, and updating and deleting entries. The database contains four tables, and all required features were implemented for each of the four entities in the system. In addition, the backend application supports creating entities through their respective RESTful API endpoint. However, the frontend application does not fully utilize this feature. The web application only allows users to create safety connections but not other entities, such as devices or physical links. Therefore, this option is mainly preparation for upcoming work and future projects.



Further, all graph-generation and auto-update features were implemented in both applications. However, the final implementation is different from the originally planned approach. Originally, it was planned that the backend server keeps track of active clients and supplies each client with updates if required. However, this intended behavior clashes with the requirement of having a RESTful application, as the REST architecture requires the entire server-client communication to be stateless. Therefore, the server should not keep track of active clients. A different approach was implemented where the server provides an API endpoint that clients can use to fetch various hash values. These hashes describe the current cache state of the backend server, and the clients can use these hash values to determine whether something has changed in the underlying data storage since their last request. The frontend utilizes this feature to update its graph and device list when another client changes any values in the database.

The frontend application implements almost all features defined in the original project definition. Users can view a list of devices, update and delete each device, and the frontend UI renders a graph that shows how the devices are connected. That graph does not only display the (physical) network connections, but it also contains (logical) safety connections, which it draws using a different color. Users can double-click edges to open a detail view that displays the values associated with an edge. If the user works with a touch display, double-tapping a graph edge reveals further details.

Users can utilize a text input field in the device list component on the dashboard to search for devices. The search removes devices from the list that do not match the user-supplied search criteria. In the current version of the frontend application, users can search devices by their name. However, later versions of the project could include more sophisticated filter options. The filters are implemented locally in the frontend. No new data is requested from the backend server should a user change the filter criteria. Instead, the frontend application removes the entries from the device list that do not match the criteria.

One requirement that was identified during the analysis phase of this project but not implemented in the final product is an additional graph representation. In its current state, the frontend application uses a single graph module that could be replaced by other representations if necessary. This switch can happen dynamically, which means that the system could swap out graph representations during runtime on a user's request. However, due to time constraints, no additional graph modules were implemented.

#### 4.2.2 Possible Future Adaptions

The project team sees the greatest potential for future improvements in the backend server's graph generation algorithm. The entire process could be optimized by utilizing database joins. Doing so would also make the entire system more scalable by balancing the load of executing computation-intensive calculations to multiple physically independent systems. In addition, the backend server would have to send fewer requests to the database server, and the Database Management System (DBMS) could optimize incoming requests

and joins more efficiently. These measures would lead to less processing overhead and fewer data packets on the network, which would especially be important in large networks and in scenarios in which many clients operate simultaneously.

Future projects could work on optimizing the auto-update mechanism between clients and the server. Currently, the server and each of the clients perform polling to check whether their local caches are still up to date. However, all these operations produce potentially unnecessary load on the CPU of each device as well as the network infrastructure. Another approach would be to allow clients to subscribe to updates using an endpoint of the server. The server could then send updates to the clients if necessary. However, doing so would mean that the resulting backend server does not act strictly according to the core REST principles. This limitation is not necessarily a problem, but as this project's main goal was to build a RESTful API, such a feature had to be omitted. Therefore, future work could focus on analyzing the possible benefit and drawbacks of making such a change in regards to saved network packets per hour, reduced CPU load, response time, and other metrics. In addition, it could be possible to add some sort of real-time criteria to the system to ensure that updates reach clients within strict time limits.

As discussed earlier, the current system implements a basic JWT-based authentication mechanism, and the frontend uses hard-coded credentials to perform an automated login. It is critical that future work addresses these flaws in the system and provides a more secure means of user authentication. For example, it would suffice to implement a login screen where operators authenticate themselves using a unique username and password combination. However, the login process could possibly also be automated using asymmetric encryption between each client and the backend server. Future work could analyze possible solutions and implement one concrete method or a combination of multiple approaches. Finally, future work could also investigate other security flaws and vulnerabilities of the whole software system described in this thesis.

Students who specialize in UI/UX design and HCI could analyze the current frontend User Interface and study how users work with the interface. They could further detect possible shortcomings of the UI and implement a different solution. A user study could reveal what features professional users want to see implemented in the future, and different color combinations and styles could be analyzed to make updates and changes in the system more apparent. In addition, it could be analyzed how quickly novice users find all features in the UI, and how fast they learn to work with the most important functionalities of the system.

Quality assurance and testing were not of utmost importance in this project. While a few selected modules were tested, no extensive quality assurance or testing measures were taken. Therefore, students interested in software controlling and software quality assurance could implement more profound testing and integration strategies. Future work could implement more detailed black-box tests using dependable methods such as equivalence class partitioning. Additional white-box tests could analyze various coverage metrics and how they help improve the quality of the software solution presented in this thesis. Finally, automated frontend tests could help increase the quality of the UI.

Like extensively testing the software, ensuring data consistency in the database and across the entire application was primarily out of scope in this project. Currently, data consistency must be actively managed by users of the system and the DBMS. However, the backend could enforce more sophisticated data validation rules that rigorously ensure data consistency and compliance with database constraints in future versions. In addition, the current application does not correctly handle concurrent data changes. For example, if two clients update the same device in the front end, changes could be lost due to a write-write conflict, sometimes referred to as a lost-update. Therefore, adding a locking mechanism or some form of transaction management would be necessary in order to make the project function correctly in a multi-user environment. Further data-consistency and integrity considerations should also be done in future work.

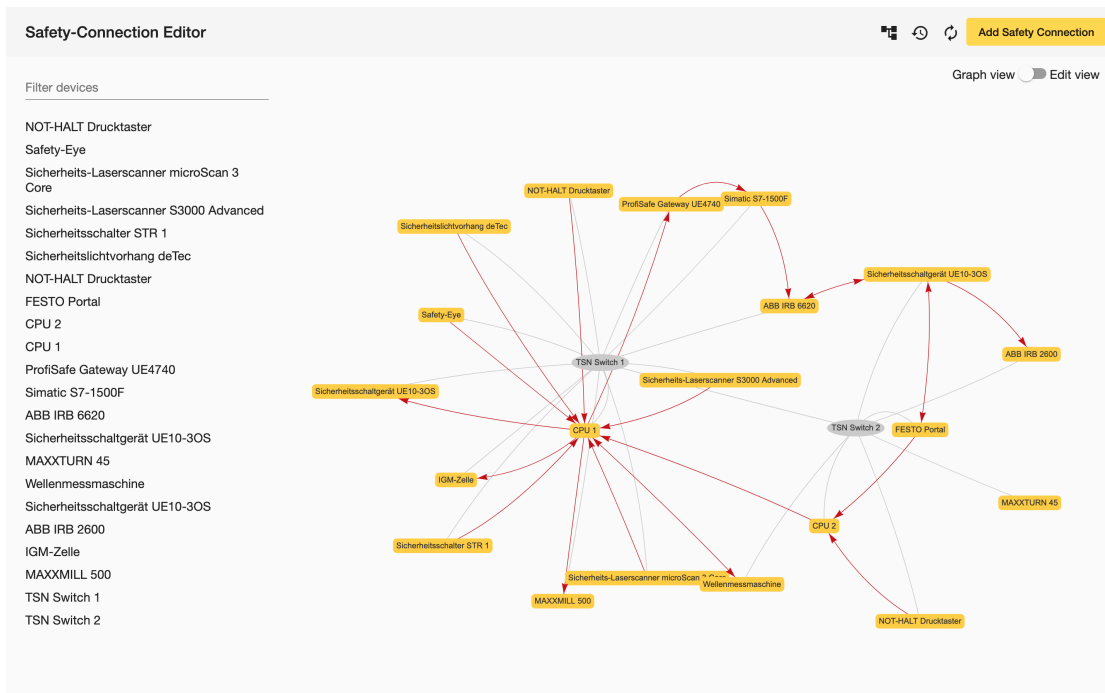


Figure 4.2: This screenshot shows the finished frontend UI. The red edges in the graph represent safety connections, the yellow nodes are safety-relevant devices, and the gray nodes represent other non-safety networking equipment, for example, switches.

### 4.3 Results

This work aimed to develop a functional prototype of a software solution that lets users view and manage safety connections between devices in an industrial setting. Automatically notifying users of changes in the underlying data storage was one of two critical requirements for this project. The second important feature mentioned in the original project description required the UI of the finished solution to be responsive and work on different devices and screen sizes. From the author's point of view, all critical goals were achieved during this project, and the developed software solutions offer all required features.

Figure 4.2 shows the finished frontend User Interface as displayed on a medium-sized screen of a common consumer-grade laptop computer. The graph representation shown in the screenshot comprises all devices present in the test database. The yellow rectangular nodes represent safety-relevant devices. The gray oval nodes represent other not safety-relevant networking gear, computers, and machines. In this example, the network only consists of the safety-relevant devices and two TSN switches. The gray edges in the graph represent physical network links between devices, and the red edges correspond to the safety links found in the database. The graph represents physical networking links using undirected edges, and it contains directed edges that correspond to the safety links. Uni-directional edges always originate at the node that outputs a safety function, and they point towards the safety input on another device. Bi-directional edges represent that multiple inputs and outputs of two devices are connected to one another. Double-clicking any edge opens a pop-up dialogue that contains further information. However, double-clicking any safety edge further lists all involved variables and their directions, as shown in Figure 4.3.

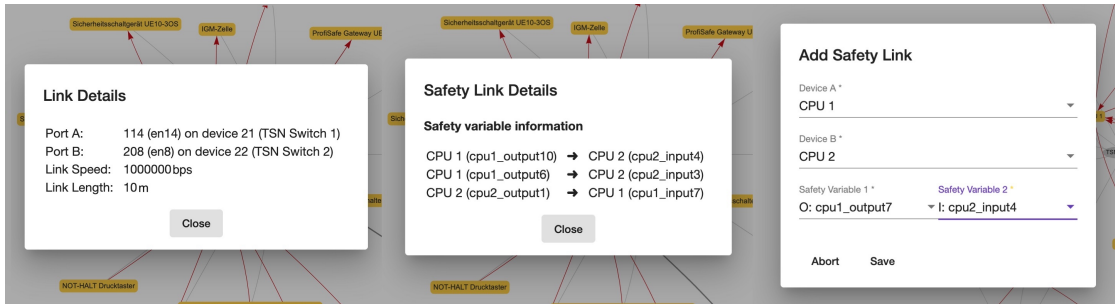


Figure 4.3: This figure shows all pop-up dialogues used in the frontend GUI. The leftmost pop-up window displays a detailed description of a physical link. The middle image shows the dialogue window that displays the variables summarized in a bi-directional edge in the graph. Finally, the rightmost dialogue lets users add new safety connections.

The rightmost screenshot in Figure 4.3 shows the pop-up window that lets users add safety links in the frontend application. Once a user selects two valid devices from the drop-down menus, the dialogue requests their associated safety variable information from the backend and displays the retrieved data using the two bottom drop-down menus. Here, the UI lets users select two variables they want to link. The drop-down menus contain already linked variables. However, users will not be able to choose such variables. Once the user selects two variables and clicks the save button, the UI dialogue performs some simple validity checks, for example, whether the user selected precisely one output and exactly one input variable. If all checks succeed, the frontend application sends a few update requests to the backend, which then handles updating the affected safety-variable entities in the database.

The finished implementation conforms to the results identified in the state-of-the-art analysis of this thesis. The backend builds upon the Spring Boot framework, a de-facto standard for developing RESTful web applications in Java. In addition, the frontend utilizes Angular, which is also considered a state-of-the-art solution for developing web-based single-page applications. Finally, the graph module used in this project implements a physics-based spring-embedder model.

As discussed in Section 4.2.2, the finished product should be considered a proof of concept or a functional prototype. Future work should be done to ensure that the application can grow into a reliable and practical solution that works in large-scale multi-user environments. The project team also concludes that further user analysis needs to be conducted to decide how valuable the system is under natural operating conditions.



# List of Figures

2.1	This figure illustrates the two types of graphs relevant for this project . . .	9
2.2	A screenshot of the results of the IEEE spectrum ranking of 2021 [Spe21].	17
3.1	This activity diagram illustrates how the backend server handles user requests. Further, it outlines each layer's responsibilities and the communication between them. . . . .	32
4.1	This block diagram gives an overview of the entire system. Note that the blocks and the connections between them are not exhaustive. They were only included for illustrative purposes. . . . .	44
4.2	This screenshot shows the finished frontend UI. The red edges in the graph represent safety connections, the yellow nodes are safety-relevant devices, and the gray nodes represent other non-safety networking equipment, for example, switches. . . . .	47
4.3	This figure shows all pop-up dialogues used in the frontend GUI. The leftmost pop-up window displays a detailed description of a physical link. The middle image shows the dialogue window that displays the variables summarized in a bi-directional edge in the graph. Finally, the rightmost dialogue lets users add new safety connections. . . . .	48









# Acronyms

- AOP** Aspect-Oriented Programming. 14
- API** Application Programming Interface. 12, 14–18, 26–29, 31, 38, 43–46
- CLI** Command-Line Interface. 19
- DBMS** Database Management System. 29, 45, 47
- DI** Dependency Injection. 14
- DOM** Document Object Model. 20
- DTO** Data Transfer Object. 31
- GUI** Graphical User Interface. 48, 51
- HATEOAS** Hypermedia as the Engine of Application State. 12
- HCI** Human-Computer Interaction. 46
- HMI** Human-Machine Interface. 3, 5–7, 22
- HTTP** HyperText Transfer Protocol. 11, 12, 31, 38
- IDE** Integrated Development Environment. 26
- IOC** Inversion of Control. 14
- JPA** Java persistence API. 14
- JS** JavaScript. 14, 18, 39
- JSON** JavaScript Object Notation. 11, 31
- JSX** JavaScript XML. 20
- JWT** JSON Web Token. 27, 29, 38, 46

**MVC** Model-View-Controller. 12, 13, 16

**MVVM** Model-View-Viewmodel. 13, 16, 20

**NPM** Node Package Manager. 15, 17, 20

**PLC** Programmable Logic Controller. 2, 7

**REST** REpresentational State Transfer. 18, 26, 28, 29, 33, 45, 46

**SQL** Structured Query Language. 27

**TIA** Totally Integrated Automation. 7

**TS** TypeScript. 14, 18–20, 38

**TSN** Time Sensitive Networking. 30, 48

**UI** User Interface. 2, 3, 5, 12, 13, 19–23, 26, 29, 30, 40, 43, 45–49, 51

**URI** Uniform Resource Identifier. 11, 38

**URL** Uniform Resource Locator. 31, 38

**UX** User Experience. 2, 3, 19, 46

**VNC** Virtual Network Computing. 6

**WCF** Windows Communication Foundation. 18

# Bibliography

- [AA18] Mariam Aljamea and Mohammad Alkandari. Mmvmi: A validation model for mvc and mvvm design patterns in ios applications. *IAENG International Journal of Computer Science*, 45(3):377–389, 2018.
- [Agg18] Sanchit Aggarwal. Modern web-development using reactjs. *International Journal of Recent Research Aspects*, 5(1):133–137, 2018.
- [Aut21] BR Automation. BR Automation Power Panel T-Series. <https://www.br-automation.com/en/products/hmi/power-panel-t-series-and-c-series/>, 2021. [Online; accessed Dec. 28, 2021].
- [Bel06] Ron Bell. Introduction to iec 61508. In *ACM International Conference Proceeding Series*, volume 162, pages 3–12. Citeseer, 2006.
- [BMV19] Gleison Brito, Thais Mombach, and Marco Tulio Valente. Migrating to graphql: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150. IEEE, 2019.
- [Bro00] Simon Brown. Overview of iec 61508. design of electrical/electronic/programmable electronic safety-related systems. *Computing & Control Engineering Journal*, 11(1):6–12, 2000.
- [BRR17] Thomas Bläsius, Marcel Radermacher, and Ignaz Rutter. How to draw a planarization. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 295–308. Springer, 2017.
- [CJP<sup>+</sup>20] Sean B Cleveland, Anagha Jamthe, Smruti Padhy, Joe Stubbs, Michale Packard, Julia Looney, Steve Terry, Richard Cardone, Maytal Dahan, and

- Gwen A Jacobs. Tapis api development with python: best practices in scientific rest api implementation: experience implementing a distributed stream api. In *Practice and Experience in Advanced Research Computing*, pages 181–187. 2020.
- [DBETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.
- [ESK04] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama’s algorithm for layered graph drawing. In *International Symposium on Graph Drawing*, pages 155–166. Springer, 2004.
- [Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
- [FK12] J Joseph Fowler and Stephen G Kobourov. Planar preprocessing for spring embedders. In *International Symposium on Graph Drawing*, pages 388–399. Springer, 2012.
- [Fow10] Martin Fowler. Richardson Maturity Model. <https://martinfowler.com/articles/richardsonMaturityModel.html>, 2010. [Online; accessed Dec. 30, 2021].
- [JJVR17] Željko Jovanović, Dijana Jagodić, Dejan Vujičić, and Siniša Randić. Java spring boot rest web service integration with java artificial intelligence weka framework. In *International Scientific Conference “UNITECH 2017*, pages 323–327, 2017.
- [KD17] Krishan Kumar and Sonal Dahiya. Programming languages: A survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5(5):307–313, 2017.
- [KK<sup>+</sup>89] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.
- [Kob12] Stephen G Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

- [L<sup>+</sup>16] Tian Lou et al. A comparison of android native app architecture mvc, mvp and mvvm. *Eindhoven University of Technology*, 2016.
- [Lan16] Kenneth Lange. The little book on rest services. *Dostupno na: <https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf> [4. svibnja. 2020]*, 2016.
- [Lar22] Laravel. Laravel Documentation for version 8.x. <https://laravel.com/docs/8.x>, 2022. [Online; accessed Jan. 01, 2022].
- [MMVH17] Romeo Margea, Camelia Margea, Bogdan Veche, and Călin Hurbean. Mobile first. current trends and practices in website design. *Annals of the University Dunarea de Jos of Galati: Fascicle: I, Economics & Applied Informatics*, 23(3), 2017.
- [Nod22a] NodeJS. Blocking vs. Non-Blocking. <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/>, 2022. [Online; accessed Jan. 01, 2022].
- [Nod22b] NodeJS. Introduction to NodeJS. <https://nodejs.dev/learn>, 2022. [Online; accessed Jan. 01, 2022].
- [Nod22c] NodeJS. What are callbacks? <https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/>, 2022. [Online; accessed Jan. 01, 2022].
- [Pau14] Cesare Pautasso. Restful web services: principles, patterns, emerging technologies. In *Web Services Foundations*, pages 31–51. Springer, 2014.
- [PNR<sup>+</sup>20] AA Prayogi, M Niswar, M Rijal, et al. Design and implementation of rest api for academic information system. In *IOP Conference Series: Materials Science and Engineering*, volume 875, page 012047. IOP Publishing, 2020.
- [Sak19] Elar Saks. Javascript frameworks: Angular vs react vs vue. 2019.
- [Sie21a] Siemens. Siemens SIMATIC Panel IPC. <https://new.siemens.com/global/en/products/automation/pc-based/simatic-ifp-itc.html#IndustrialThinClients>, 2021. [Online; accessed Dec. 27, 2021].

- [Sie21b] Siemens. Siemens SIMATIC Thin Client. <https://new.siemens.com/global/en/products/automation/pc-based/simatic-panel-pc.html>, 2021. [Online; accessed Dec. 27, 2021].
- [Sie21c] Siemens. Siemens TIA Portal overview. <https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/software.html>, 2021. [Online; accessed Dec. 28, 2021].
- [Sie21d] Siemens. Siemens WinCC Unified Manual. [https://support.industry.siemens.com/dl/dl-media/204/109794204/att\\_1069892/v1/144323721483\\_en-US/en-US/index.html#treeId=56a2b589b712698a75488565be436bc0](https://support.industry.siemens.com/dl/dl-media/204/109794204/att_1069892/v1/144323721483_en-US/en-US/index.html#treeId=56a2b589b712698a75488565be436bc0), 2021. [Online; accessed Dec. 28, 2021].
- [Sie21e] Siemens. Siemens WinCC Unified Manual. [https://support.industry.siemens.com/dl/dl-media/204/109794204/att\\_1069892/v1/144323721483\\_en-US/en-US/index.html#treeId=3db925c4eac1cb8850086c5a2798ea41](https://support.industry.siemens.com/dl/dl-media/204/109794204/att_1069892/v1/144323721483_en-US/en-US/index.html#treeId=3db925c4eac1cb8850086c5a2798ea41), 2021. [Online; accessed Dec. 28, 2021].
- [Sie21f] Siemens. Siemens WinCC Unified overview. <https://new.siemens.com/global/en/products/automation/simatic-hmi/wincc-unified.html>, 2021. [Online; accessed Dec. 28, 2021].
- [SJT17] Hatma Suryotrisongko, Dedy Puji Jayanto, and Aris Tjahyanto. Design and development of backend application for public complaint systems using microservice spring boot. *Procedia Computer Science*, 124:736–743, 2017.
- [SM10] Erik Sorensen and M Mikailasc. Model-view-viewmodel (mvvm) design pattern using windows presentation foundation (wpf) technology. *MegaByte Journal*, 9(4):1–19, 2010.
- [Spe21] IEEE Spectrum. IEEE Spectrum ranking of the most popular programming languages. <https://spectrum.ieee.org/top-programming-languages/>, 2021. [Online; accessed Jan. 01, 2022].
- [Spr21] Spring.io. Spring Initializr. <https://start.spring.io/>, 2021. [Online; accessed Dec. 31, 2021].



- [Sug02] Kozo Sugiyama. *Graph drawing and applications for software and knowledge engineers*, volume 11. World Scientific, 2002.
- [SW18] Thomas Schmidt and Christian Wolff. The influence of user interface attributes on aesthetics. *i-com*, 17(1):41–55, 2018.
- [Tra97] Noam Tractinsky. Aesthetics and apparent usability: empirically assessing cultural and methodological issues. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 115–122, 1997.
- [Tun94] Daniel Tunkelang. A practical approach to drawing undirected graphs. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1994.
- [Vis22] Vis.js. Vis.js homepage. <https://visjs.org/>, 2022. [Online; accessed Jan. 22, 2022].
- [Wal15] Craig Walls. *Spring Boot in action*. Simon and Schuster, 2015.
- [Woh18] Eric Wohlgethan. *Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js*. PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.
- [WSL<sup>+</sup>13] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, and Sébastien Deleuze. Spring boot reference guide. *Part IV. Spring Boot features*, 24, 2013.
- [YMX<sup>+</sup>16] Xuyong Yang, Tao Mei, Ying-Qing Xu, Yong Rui, and Shipeng Li. Automatic generation of visual-textual presentation layout. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 12(2):1–22, 2016.

# **Appendix A**

## FSS GUI Test Strategy

**Daniel Herczeg**

# 1 Introduction

## 1.1 Purpose

This document describes the test strategy used for testing the functionality of the Flexible Safety Systems GUI project using the version of the program handed in as part of the bachelor thesis this test plan is attached to.

## 1.2 Project Brief

The goal of the Flexible Safety Systems GUI project was to develop a graphical user interface that trained personnel can use to view and modify an existing network structure of safety-related devices in industrial environments. The finished product consists of two parts, a frontend application that can run in any modern web browser, and a Java-based backend application that allows the frontend to access the underlying database.

## 1.3 Audience

This document addresses the developers of the project, the supervisors (project assistants and professors), as well as students and engineers expanding the project in the future.

# 2 Test Strategy

## 2.1 Test Objectives

The objective of the automated and manual tests conducted throughout the development of this project is to verify that the system responds as expected to predetermined valid inputs and conditions. However, neither automated nor manual tests can guarantee the absence of errors in the finished project. Formal software verification is not part of the test strategy of this project.

## 2.2 Scope

Testing each component of the entire software project was not a viable option at this stage of the project due to time constraints. However, a few critical components were tested during and after development, namely the following two classes:

- a) Cache.java
- b) GraphService.java

These two classes were selected as they provide key functionality in the final software system. No automated tests were conducted for the frontend application.

Testing the database, external data-gathering mechanisms, standard functions provided by the frameworks in use, and the integration of components is not part of this test strategy.

## 2.3 Methodology

The project uses automated component tests for verifying that the two aforementioned backend components function as intended for a series of valid inputs. The project utilizes the JUnit4 framework for running the component tests. The Mockito library provides the test objects with mock functions and data in order to isolate the unit under test from external influences.

The project contains a separate test class for each of the two units under test (the Cache and GraphService classes), which were created after the classes were fully implemented and integrated. The test classes got implemented according to the test cases identified by the project team when in the original requirements of the project.

The project team executed the all test classes once the implementation was done as well as after every iteration of the project that either fixed runtime or compile errors or contained new features (regression tests). The test cases were not adapted throughout the project.

The frontend application of the project does not contain any automated tests. However, the project team and the stakeholders manually tested the frontend functionality after every major iteration of the project, typically in bi-weekly meetings throughout the entire development cycle of the project.

## 2.4 Further testing

As mentioned, the project does not include any formal acceptance or integration tests. However, bi-weekly meetings gave the stakeholders and the project developer the chance to demonstrate and inspect how the project functions. These informal meetings served as a form of user acceptance test, and the project developer respected and incorporated the feedback given by the stakeholders during these meetings. However, as the meetings were informal, no checklists, protocols, change requests, or other documentation exists.

## 2.5 Assumptions

- a) All data required by the project will be provided by an external exploration program or by hard-coded test data in the form of database insert queries.
- b) The underlying data-generation and exploration mechanism functions as intended.
- c) The software libraries and frameworks in use are well-tested and them causing an error in this project is unlikely.
- d) All features are fully implemented at the time of running the automated unit-test cases.
- e) The project developer performs regression testing whenever adding new features or fixing previous errors in the software.
- f) Problems, errors, and fixes will be tracked using Gitlab Issues using a project-internal non-public repository.

- g) The backend server will always run without connectivity problems when testing the frontend application
- h) The backend server application will never run when executing the automated backend unit-test cases.
- i) The IDE will always execute the test cases using the same procedure
- j) The underlying Java version does not change during development
- k) External libraries and frameworks will not be updated during development

## 2.6 Test Principles

- a) Testing focuses on the most crucial modules of the project due to timing constraints
- b) Backend testing happens completely automatically
- c) Frontend testing happens completely manually
- d) Mock Objects will be used to isolate the unit-under-test and to ensure deterministic, repeatable results
- e) Testing will be a repeatable and measurable activity with clearly defined input values, expected outputs, and goals.

## 3 Test cases

This section contains the test cases that were implemented in the two test classes. Each table describes a unit-test's supplied inputs, the expected behavior and output (if applicable), as well as the function that implements the test case in one of the two test classes.

### 3.1 GraphServiceTest.java

ID	Name / Description	Input	Output / Expected Behavior
1	Generating a graph without data returns an empty graph	Empty database	Empty graph
2	Generating a graph for a node without connections returns a graph that contains only that one node	A mock-database that only contains one device	A graph that contains only the one node from the database
3	Generating a graph with two nodes connected via a single edge returns a graph that contains precisely those nodes connected by a single, undirected edge	A mock-database that contains two independent devices and a single link	A graph that contains two nodes and one undirected edge
4	Generating a graph with two devices (ids 1 and 3) connected by two variables creates a graph that contains a directed edge between 1 and 3	A mock-database that contains three devices, six ports, two links, and six variables	The resulting graph contains the two devices and a directed edge between them
5	Generating a graph with a safety link between devices 1 and 3 generates a directed edge going from 1 to 3 but not from 3 to 1	A mock-database that contains three devices, eight ports, four links, and six variables	The resulting graph contains a directed link 1 -> 3 but not 3 -> 1
6	Generate a network graph for three physically connected devices with one logical link	A mock-database that contains three devices, six ports, two links, and six variables	The resulting graph contains all three devices, two physical links, and one logical link
7	Generate a complete network for the node with id 2	A mock-database that contains five devices, eight ports, four links, and seventeen variables	The resulting network contains five nodes, four physical links, and four logical links. The edge directions are as follows: 1 -> 3 3 -> 4 4 -> 2 2 -> 1

### 3.2 CacheTest.java

ID	Name / Description	Input	Output / Expected Behavior
1	Getting an element from an empty cache throws an exception	Empty cache	Exception
2	Checking whether an empty cache contains an element always returns false	Empty cache	false
3	Checking whether a non-existent element is in the cache returns false	Cache with one element	false
4	After adding an element to the cache, checking the cache for that element returns true	A cache that contains one entry	true
5	Getting a valid element from the cache returns that element	A cache with one element	The cache returns the requested element
6	Clearing the cache deletes previously added elements	Non-empty cache	The elements are no longer in the cache
7	Updating an existing element in the cache replaces that element	A cache containing one entry	The old value is no longer cached, the new value is in the cache
8	Updating one element in the cache leaves the others unchanged	A cache containing two distinct entries	One entry remains the other one gets replaced by an updated value

## 4 Conclusion

As concluded in the thesis this document is attached to, the presented test strategy is not exhaustive or complete, and further testing is necessary to ensure that the presented software solution functions as intended for valid inputs, invalid inputs, and edge cases. However, developing an extensive and detailed test strategy was not part of this project. Therefore, the two presented automated component tests focused on verifying that two of the most important modules in the project's backend application function as indicated by the project specification and original list of requirements.