MUNI
FACULTY
OF INFORMATICS

Building a Web-based Interactive Network Visualization in Vue.js

Bachelor Thesis

Marko Řeháček

Masaryk University
Faculty of Informatics

Brno, Autumn 2019

Declaration

I declare that this thesis is my original work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marko Řeháček

Supervisor: Mgr. Kristína Zákopčanová

Acknowledgements

The presented work is a part of the research project "**Complex Analysis and Visualization of Large-scale Heterogeneous Data**" supported by the Ministry of the Interior of the Czech Republic within the program "Security Research for the Needs of the State Program 2015–2020" under identification VI20172020096. The project is a cooperation between the Faculty of Informatics (Masaryk University), Institute of Computer Science – ÚVT (Masaryk University), and the Police of the Czech Republic.

On a more personal level,

I want to heartily thank my parents Daniela and Vlastimil, for without their support, I would never have been able to do the things I love.

An extra-special thank you belongs to my close friends for providing me with their insightful advice, commenting my writings, and taking me outside when needed.

Thanks to everyone who has provided me with food and listened to my excuses during the writing of this thesis.

And last, but not least, I want to express my immeasurable gratitude to Kiki. For all her kindness, mentoring, shepherding me through the project for the last year, and the opportunity to work on a meaningful thing.

Abstract

Visualization is an effective way of gaining valuable insight into data. For instance, in the criminal investigation domain, analysts facilitate their decision-making using network visualizations, which allow them to find connections in complex datasets modeling real-world phenomena. This thesis acknowledges the importance of building specialized visualizations and describes web-based approaches to their development with the focus on their interactivity. The common usage of the D3.js visualization library may be outdated for complex use cases and the work proposes an approach to building web-based visualizations using the recent features of frontend frameworks. The approach is evaluated on a case study of a network visualization prototype. The prototype was built using framework Vue.js and state management library Vuex. The benefits of selected technologies are presented on several interaction techniques in the network diagram. The primarily discussed technique is node aggregation, for which this work presents a new algorithm. The implemented prototype forms a part of the research project “Complex Analysis and Visualization of Heterogenous Big Data”.

Index terms

data visualization, custom visualization, interactive visualization, web-based visualization, network visualization, visual analysis, criminal investigation, JavaScript, D3.js, frontend frameworks, Vue.js, Vuex

Contents

0 Introduction

"The purpose of (scientific) computing is insight, not numbers." Richard Hamming

Since the rise of computing, we have experienced a massive information boom. We live in an era where we collect enormous amounts of complex data [1] and when we want to extract information out of it, we are challenged with its processing and analysis. The insight can often be very valuable, and we naturally seek new ways of gaining it. Pursuing advances in industry and scientific research requires us to come up with new ideas that will allow people to explore and reason about big or complex datasets. To address this issue, a multidisciplinary field of ***data science*** emerged. One of the methods data science uses is ***data visualization***, which exploits our visual perception to help us understand data more efficiently than we can just by reading a text or numbers.

The topic related to this work is data visualization in the criminal investigation domain. Analysts solving criminal cases require specific tools in order to efficiently work with ***big heterogeneous data*** collected by the police. ***Network visualizations*** are one of the tools which help analysts find connections in the data and facilitate their decision-making process, and these visualizations will be described later.

The primary goal of this work is to design a system architecture for web-based interactive visualizations, and based on that architecture, propose a prototype of a network visualization for the police. The prototype will form a component of a system developed in university's research project "Complex Analysis and Visualization of Large-scale Heterogeneous Data", which we refer to as ***project Analysis***.

The first chapter introduces the reader to the field of data visualization, the tools used in the field, and web-based visualization. The second chapter presents a common approach to building web-based data visualizations using the library ***D3.js***, discusses the possible limitations of the library, and introduces recent concepts related to web development that could improve the development of these visualizations. The third chapter presents an approach to building visualizations using ***frontend frameworks*** to solve development problems when the visualizations become complex. The fourth chapter presents a case study of the prototype developed for the project Analysis, which uses the presented approach.

1 Data visualization

One of the ways data science helps people reason about complex data is presenting them graphically, using a technique called **data visualization**. We can define data visualization as **the use of computer-supported, interactive, visual representations of data to amplify cognition**[1]. In the following sections, we will explain the meaning behind the terms used in the definition.

Visualization can be described as an efficient way of communicating a message by visually presenting it. Well-crafted visualizations can **amplify our cognition**: they "extend our ability to think ... by assisting memory and representing the data in ways, that our brains can easily comprehend" [2].

More precisely, visualization is the process of ***mapping*** data or information to visual output in a way that is meaningful and understandable to the viewer. Figure 1 illustrates this process on a real-world example of a bar chart, which visualizes how crowded a public place is during the day. The chart consists of rectangular bars of different height, corresponding to the number of visitors during each hour – the number of people is mapped to the height of the rectangles. This allows viewers to easily compare those numbers visually.

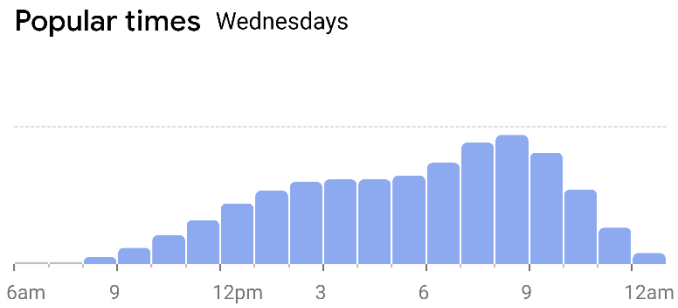


Figure 1: Bar chart example from Google Maps, visualizing the number of visitors of a public place.

Simple visualizations like this can be drawn manually in graphical editors; however, once we start dealing with datasets with multiple dimensions or containing millions of data entries, it becomes almost impossible to do this process by hand. It becomes necessary to have the visualization process aided by computation.

The use of computers changes the visualization process: the data visualization practitioners do not need to manually draw charts; instead, they design a set of rules for the visual mapping, and then write software for generating the visualization according to the rules[2] [3, p. 2]. Furthermore, having computers included in the design process allows prototyping, where the programmer can easily adjust the visualization process to improve its output. However, there is another great benefit of **computer-supported** visualizations. When visualizations

[1] Presented definition of data visualization is based on the definition for information visualization, popularized by Card et al. [38].

[2] The term *visualization* is used interchangeably for the visual mapping process and for its output.

run on a computer, there is a technical possibility to create **interactive visualizations**. Interactivity opens several new ways for data visualization to be more powerful.

1.1 Interactive data visualizations

> "The world has moved on from reading and observing to seeing and interacting." [4]

Before computers became a suitable platform for visualization, we were only able to make static, precomposed views of data, which did not necessarily cater for our needs. Now we can allow viewers to interact with the visualization to make it more useful and suited for their case. For example, it may allow them to dynamically change the view, i.e., approach the data from a different perspective and possibly reveal hidden information.

Another challenge of data visualization is finding a good representation for datasets with several dimensions. Our visual processing capabilities, together with the size of the visualization, limit the number of visual elements that can be displayed at once. One of the ways interactivity helps tackle this problem is by allowing the viewer to change the level of detail of the visualization and reduce or expand the data shown.

> "Dynamic, interactive visualizations can empower people to explore the data for themselves." [3]

By using interaction, we can also create a visualization that fulfills several roles at once and is therefore useful for different audiences and use cases [3, p. 2]. Firstly, the visualization may allow us to **explore** to get familiar with the dataset and determine what might be interesting, so we can later form a hypothesis. Secondly, if we already have a hypothesis, which is backed up by the visualization, we may want to present that information to our audience. The visualization can allow us to **explain** and **tell a story.** [5]

Even though this is not a comprehensive list of all the benefits, we can already conclude that interactivity can play a significant role in the data visualization process [6], [7].

The following section introduces the different types of software used for building data visualizations. Throughout the thesis, this software will be referred to as the **data visualization tools**.

1.2 Data visualization tools and their attributes

First, let us describe what are the possible characteristics of available data visualization tools in order to help us understand their categorization and allow us to select appropriate tools for given tasks.

Bostock and Heer [8], [9], renowned for their work in the data visualization field, identified three key attributes of visualization tools. This thesis will refer to them to assess the benefits and drawbacks of discussed technologies. We can define and use the attributes as follows:

i. ***expressiveness***[3] – "What kinds of visualizations can I build with this tool?" – the variety of possible visualization forms the tool allows to create
ii. ***efficiency*** – "How long will it take to create the visualization using this tool?" – the cost of development when using the tool
iii. ***accessibility*** – "How hard it is to start building visualizations without prior knowledge of the tool?" – the learning curve of the tool

Based on these attributes, Bostock and Heer [9] divided the visualization tools into the following categories:[4]

i. **Office suits**, such as *Microsoft Office* or *Google Docs/Sheets*, that can generate predefined charts, allowing only for minor visual customization (low expressiveness). We can refer to these tools as *closed*, as the user cannot create a new form of visual encoding or customize all visual aspects as desired. Creating the charts does not require programming: it is therefore fast (high efficiency) and does not require any prior knowledge of the tool (high accessibility).

ii. **Analytical tools**, such as *Tableau* or *Power BI*, still support only predefined charts, but they also provide tools for data manipulation and analysis. These tools are mostly used in the field of Business Intelligence to provide insight into business operations using data. When compared to the first category, they offer to build more complex and somewhat interactive visualizations, still without a need for programming. Users, however, need to already know how to use them.

iii. **Programming tools (libraries and frameworks)**[5], such as *D3.js, p5.js, Vis.gl, Plotly.js, Google Charts* or *ggplot2*, tend to be the most versatile tools that are

[3] The term *expressiveness* does not refer to the metric for information concentration, as it is typically used in traditional visualization literature.

[4] Bostock and Heer defined the tools using the term *visualization systems*. Throughout this thesis, I will instead refer to them as *visualization tools*, to simplify terminology. The categories are not strict.

[5] Software libraries are ready-made implementations for solving specific problems. Software frameworks can instead be viewed as skeletons for applications and embody a reusable system architecture.

meant for programming the data visualizations. Each of these tools has a specific tradeoff between expressiveness, efficiency, and accessibility.

Many of these tools are high-level, such as *Plotly.js* or *Google Charts*, which are libraries that provide ready-to-use charts for the web (examples in Figure 2). They are easy to use; however, creating new visualizations or modifying the existing ones is difficult, and we can consider these tools

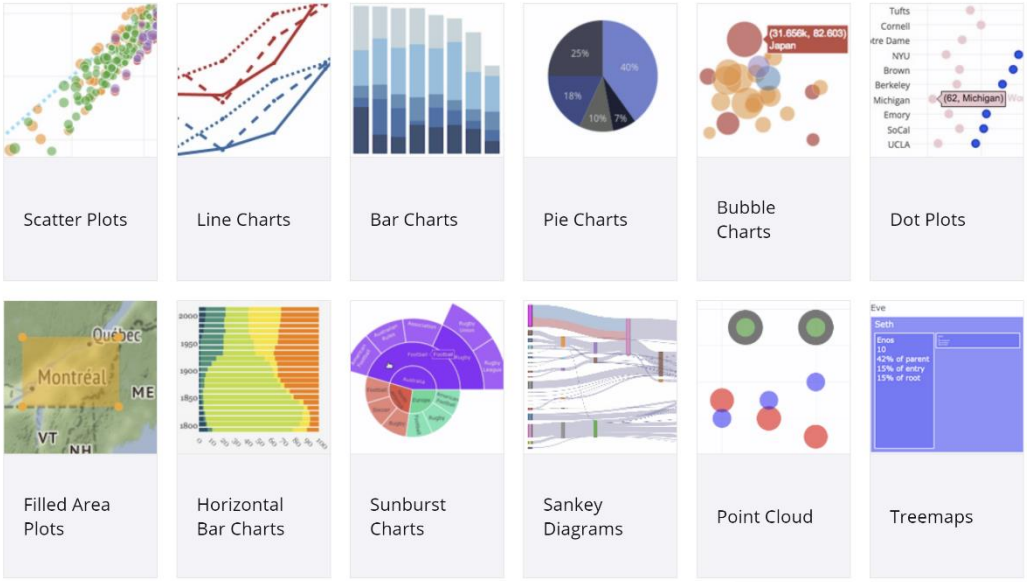


Figure 2: Example of categories of charts provided by Plotly.js [41].

closed. These libraries are mostly used by web developers to enhance their web applications.

Tools with fewer abstractions, such as *D3.js* or *p5.js*, can be harder to work with, but they do not tend to put limitation to what we can be built with them. We can describe these tools as *open*, meaning they are open for *customization*, open for building any visualization form. They are expressive at the expense of efficiency or accessibility and are mostly used by data visualization practitioners.

This thesis focuses on the last category, the category of programming tools, and especially on the subcategory of open tools, such as D3.js. They allow us to build *custom* and interactive visualizations, which are becoming increasingly relevant in solving complex problems of scientific domains, such as biomedicine, AI, or criminal investigations.

1.3 Custom visualizations

Many visualization problems can be solved simply by using predefined charts in software libraries. However, there are problems, which require data visualizations explicitly designed for solving them. So, in order to solve them, we often customize a standard visualization method by combining several methods, or even compose

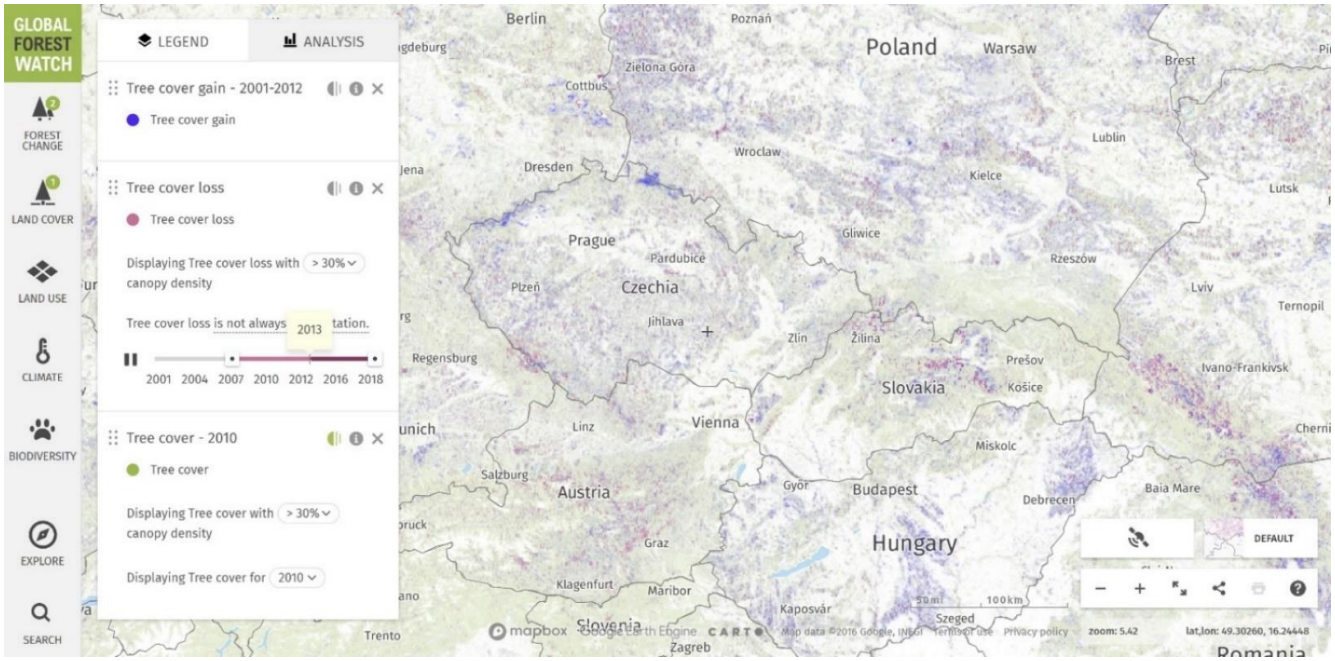


Figure 3: Example of a custom data visualization, explicitly created for analyzing deforestation. Available at https://www.vizzuality.com/project/global-forest-watch/.

entirely new types of visualizations, as depicted on Figure 3. We can then describe all these visualizations as **custom visualizations**.

Programming custom visualizations may involve the use of lower-level tools, sometimes down to the level of manipulating with pixels. The reason is that the more directly we control the graphical rendering system, the more control we can have over its visual output, and every additional layer of abstraction between may prevent us from creating the exact visualization we want to achieve. However, developing visualizations with low-level tools and without abstractions is complicated and costly, therefore inefficient it is most likely inefficient for most data visualization problems.

To improve efficiency, we need tools that can be domain-specific for visualization, provide abstractions without limiting the possibilities of visualization, and have a reasonable tradeoff between expressiveness, efficiency, and accessibility. [9]

1.4 The web platform

One way we can create custom interactive data visualizations is by building them as applications running in a web browser, using a combination of technologies created for the web. We can call this combination the *web-standard technologies,* and they include:

- **HTML**, which defines the structure and content of a web page document,
- **CSS**, which defines the aesthetics of the document,
- **SVG**, which allows for 2D vector graphics,
- **JavaScript**, which facilitates the interaction within the page.

The combination of these web-standard technologies is commonly used as a tool for creating data visualizations. When we assess them as a tool using the attributes defined earlier, we can claim they are relatively:

- **expressive**, as they can now match the expressiveness of low-level graphical systems
- **efficient**, as web technologies use time-proven abstractions for rendering visual elements
- **accessible**, as there are web technologies with easy learning curves

1.4.1 Benefits of web-based visualizations

Data visualizations can benefit from being web applications. For example, visualizing directly in a browser has a significant accessibility advantage over desktop applications: users do not need to install additional software [8, p. 15]. "Publishing on the web is thus the quickest way to reach a global audience [3, p. 3]".

Moreover, the web-based approach allows the use of visualizations in online newspapers. Visualizations now often enhance **storytelling** in the field of data journalism: it is becoming more and more common for prominent journals like *The New York Times, Washington Post, or Bloomberg* to complement news with interactive data visualizations [10]. If the visualization is implemented for the web, it should be possible to embed it in the same form into an online news article or a complex web application.

From a different, development perspective, building the visualization using web-standard technologies also has several benefits:

i. Performance and rendering capabilities of web browsers are significantly better than in the previous years [8, p. 2]. Ongoing adoption of HTML5 canvas and WebGL technologies[6] means the support for 3D or otherwise computationally demanding visualizations.
ii. The gap between desktop and web applications is disappearing in terms of real-time interactivity and the user interface design [7]. Users expect the same level of interactivity and well-crafted user interfaces.
iii. Seamless cooperation between the web-standard technologies encourages inter-element communication between the UI and the visualization itself.
iv. It promotes cooperation between web developers and data visualization practitioners.

---

[6] WebGL is a browser's API allowing high-performance rendering of interactive 2D and 3D graphics. Canvas API focuses on 2D graphics. [36]

[7] Many popular companies are now developing fully capable browser applications instead of the desktop ones: examples from popular productivity software may include Microsoft's or Google's office suites, or even fully featured design tools, such as Figma.

2 Web technologies and tools for visualization

Web-standard technologies, in themselves, can be used to build web applications. To build web applications more quickly and effectively, we can use various tools, libraries, and frameworks. The same applies to interactive data visualizations – to build them more effectively, we can use additional tools, specific for creating visual output.

In 2009, Bostock and Heer from the Stanford Vis Group argued for the development of new visualization tools, as they found out that none of the available tools were good enough in bridging the gap between low-level graphical rendering systems and high-level visualization tools. The new tool was supposed to allow creating custom web-based visualizations with minimal effort of the developer and without a third-party browser plugin. [9]

They have released ***Protovis***, a JavaScript library for constructing visualizations by composing basic graphical primitives. The key element of the toolkit was a presentation layer, which was responsible for rendering the graphical elements. The layer, however, restricted the possibilities of the visualizations, as it could only render the elements that were supported. Soon after, as a response to the increased browser support of SVG and CSS, they have reevaluated the use of the presentation layer. The focus of their new work was reducing the expressivity limitations and improving rendering performance. [9] [11]

2.1 D3.js (D3): the visualization library

Their new goals were met in the design of the ***D3.js*** library, which was published in 2011 by the Stanford Vis Group as a novel approach for building custom interactive visualizations on the web in JavaScript. D3 allows the developers to bind data to elements of HTML documents and then transform the elements to generate the desired visualization. For example, it can easily create interactive SVG charts with animations. [11]

Instead of abstracting the graphical display with a presentation layer, D3 focuses on directly using web technologies for the manipulation with the structure and content of a web page [11]. This concept is crucial in understanding the concepts of D3 and web technologies in general, and it will be explained in the following section.

D3's approach proved to be successful, and the library has become the most popular tool for building visualizations on the web [8]. One of the key goals for D3 was improving accessibility to promote developer collaboration [11]. It essentially succeeded, and despite the reusability issues[9], has created a thriving ecosystem of examples, and higher-level tools build on top of it. Open-source implementations of various visualization methods in D3 can be easily found online[10].

Also, many higher-level tools were developed using D3. We can categorize them followingly:

i. **Charting libraries** like *Plotly.js*, *C3*, or *Britecharts*, which provide the programmers with a set of standard chart types, implemented in D3. Usually, they are hard to customize, as it would require modifications to the library's core code.
ii. **Component libraries for frontend frameworks** like *react-d3*, *Semiotic*, or *ngx-charts*. Visualizations are often embedded in web applications and these libraries provide the programmers with ready-to-use modules, that can be easily integrated into applications built with frontend frameworks. They may be even harder to customize as they involve an additional layer of abstraction.

D3 was released in 2011, and it certainly was a new and better approach to building visualizations for the web [11], [12]. As eight years have passed, it may be, however, necessary to reevaluate how we use it.

2.1.1 Document Object Model (DOM) manipulation

As mentioned earlier, the common usage of D3 consists of direct manipulation with the web page using a browser's API. When the browser parses HTML, it builds itself an abstract tree-like data structure, which it uses for rendering the page. This structure is then accessible by the web application through an interface, identically called the ***Document Object Model***.

---

[8] Many authors are stating this [39]. Furthermore, on GitHub, the popular open source development platform, searching for keyword "visualization" yields D3.js first, with 88 thousand people's likes, a substantial lead. We did not find a library of similar type and popularity.

[9] Bostock [40] proposed a code pattern for reusability after the release of D3, described on his blog available at https://bost.ocks.org/mike/chart/. This pattern, however, does not seem to be widely used.

[10] Plenty of official examples are available at Observable, https://observablehq.com/@d3. Older examples from Bostock can be seen at https://bl.ocks.org/mbostock.

2.1.1.1 Document Object Model (DOM)

To be more precise, DOM refers to an object-oriented representation of HTML/XML documents while simultaneously referred to as its programming interface [13]. For instance, JavaScript can use the DOM to access and manipulate the hierarchical structure and content of the web page. Having a standardized interface to access the document is precisely the concept that allows cooperation between the web-standard technologies.

The previously mentioned hierarchical structure of the page is encoded in pairs of HTML tags. Each pair represents an element of the page, which has a specific relationship to its surroundings. This structured is illustrated on Figure 4.

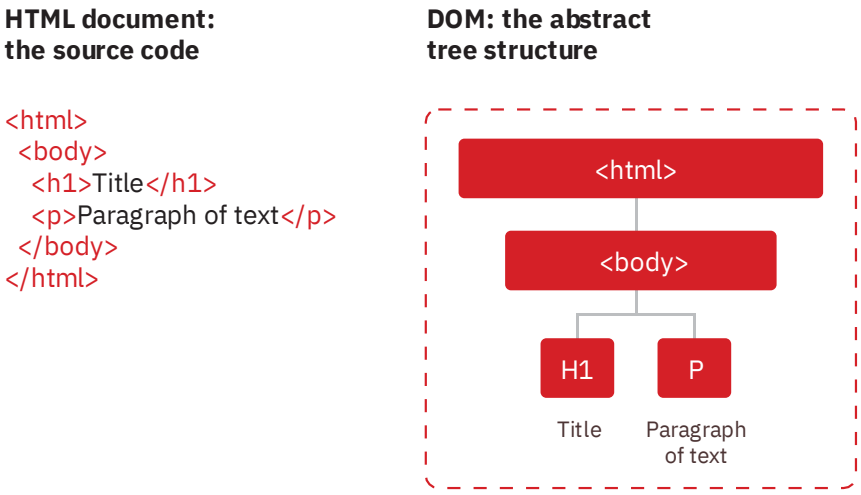


Figure 4: Hierarchical structure of HTML document.

As we build web-based visualizations, we need to understand how the browser operates with the page, as it navigates its hierarchy to apply styles and actions to the page elements, which we manipulate with. [3]

2.1.1.2 Is DOM manipulation necessary?

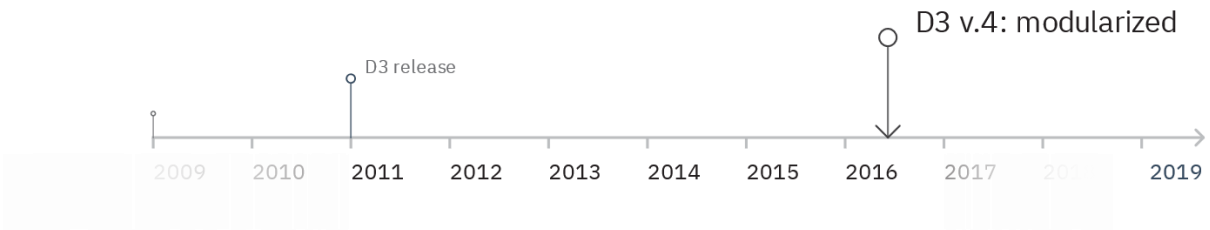
"**Data visualization, instead of DOM manipulation, should be the focus of anyone trying to learn a data visualization library**", mentions Elijah Meeks, founder of The Data Visualization Society, in his recent article (2018), where he explains parts of the D3 to new users, and asks the expert users to reexamine their approach to D3. [14]

As building visualizations with D3 inherently means to directly manipulate with the DOM of a web page, learning materials often focus on the parts of the library that allow it. A significant prerequisite for a new developer is therefore learning to add, transform, or remove elements of the page using D3's tools for DOM manipulation in a code pattern specific to the library, which, ideally, should not be necessary. Furthermore, DOM manipulation is very prominent in D3's API and commonly mentioned in learning materials and examples, which possibly

creates a misconception that D3 cannot be used without its DOM manipulation tools [14].

However, as Meeks states [14], at the time D3 was designed, there was no better approach. It is necessary to understand D3 in the time frame of web development. Only after its initial release, new solutions for DOM management appeared in the design of frontend frameworks, which could change how we work with the D3. To see if we have an option to alternatively approach DOM manipulation while using it, we first need to know how the library is structured.

2.1.2 Structure of D3



D3 was originally structured as a monolithic library. This has changed in June 2016, when the authors have decided to split it into approximately 30 separate micro-libraries. Most of the code was refactored to support the new modular structure, which has allowed the developers to use only the modules relevant to their needs.

In the fourth version, released in June 2016, D3 was divided into approximately 30 micro-libraries. Most of the code was refactored to support the new modular structure of the library, which has allowed the developers to use only the modules relevant to their needs. Meeks [14] visualized the new structure of D3 by grouping its functions and modules according to the functionality they provide, as shown in Figure 5.

We can also divide the modules of D3 to two categories by their dependency on the DOM. We should note that data visualization and data utility modules can be used without the provided DOM utilities [15], [16].

The most important modules are the data visualization modules, labeled as **Dataviz** in Figure 5, such as *geo*, *scale,* or *force.* They create drawing instructions from data: either generating SVG code, document elements, or annotating datasets with information for rendering layouts. They do not access nor require the DOM. [14]

The other category, **DOM utilities**, such as *select*, *drag*, or *zoom*, are then used to apply transformations to the page. The transformations are generated by data

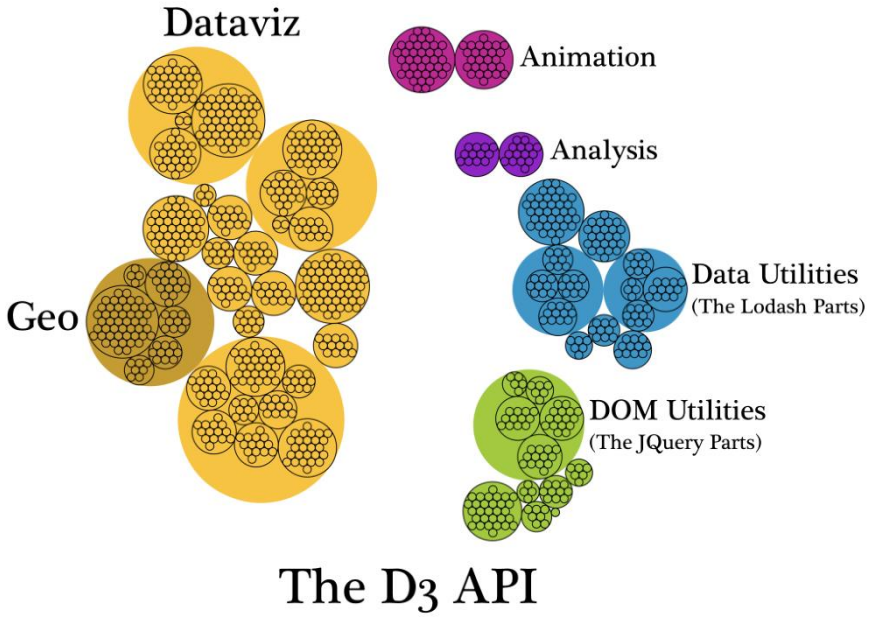


Figure 5: A hierarchical diagram of D3 functions, grouped by their respective modules and categories. Meeks [14].

visualization modules. The DOM utilities allow querying for the elements of the DOM using CSS selectors and inserting or removing them from the document.

2.1.3 Pitfalls of the D3 library

D3 works well for simple visualizations or in small one-off projects (e.g. charts in news articles). However, using D3 in large-scale projects can lead to **spaghetti code**: a code that lacks structure, and therefore is difficult to maintain and hard to read. Bostock [17] states that spaghetti code "is frequently caused by shared mutable state. When a piece of state is modified by multiple parts of a program, it is much harder to reason about its value."

The state in the D3 code is in fact difficult to track for the developer, as the DOM itself can be regarded as the state. The manipulation with the DOM happens in an unorganized manner throughout the program with a series of D3 commands (*chained declarations* similar to the builder pattern in software design). The library cannot enforce code structure as it is only a library. This problem can be referenced as the ***invisible document structure problem***. Another problematic part is the use of event bindings. These bindings are created on specific elements of the document.

If we can generalize these problems, we could conclude that the D3 code can easily be **hard to read**, even for developers experienced in D3. Figure 6 shows an implementation of a simple chart, taken from official D3 examples, proving just how complicated the code can look like. These problems are common in examples found online [18] and they can be alleviated by encapsulating the code, for the most parts.

Structuring the data visualization application using a frontend framework can enhance the quality of the code in several ways, which is described in the following sections.

Figure 6: An official D3 example of a chart illustrating the D3 code's low readability. The examples are available at https://bl.ocks.org/mbostock.

```
<style>
  …
</style>
<svg width="960" height="500"></svg>
<script>
  var svg = d3.select('svg'),
    margin = { top: 20, right: 20, bottom: 30, left: 40 },
    width = +svg.attr('width') - margin.left - margin.right,
    height = +svg.attr('height') - margin.top - margin.bottom

  var x = d3
      .scaleBand()
      .rangeRound([0, width])
      .padding(0.1),
    y = d3.scaleLinear().rangeRound([height, 0])

  var g = svg.append('g').attr('transform', 'translate(' + margin.left + ',' + margin.top + ')')

  d3.tsv(
    'data.tsv',
    function(d) {
      d.frequency = +d.frequency
      return d
    },
    function(error, data) {
      if (error) throw error

      x.domain(
        data.map(function(d) {
          return d.letter
        })
      )
      y.domain([
        0,
        d3.max(data, function(d) {
          return d.frequency
        })
      ])

      g.append('g')
        .attr('class', 'axis axis--x')
        .attr('transform', 'translate(0,' + height + ')')
        .call(d3.axisBottom(x))

      g.append('g')
        .attr('class', 'axis axis--y')
        .call(d3.axisLeft(y).ticks(10, '%'))
        .append('text')
        .attr('transform', 'rotate(-90)')
        .attr('y', 6)
        .attr('dy', '0.71em')
        .attr('text-anchor', 'end')
        .text('Frequency')

      g.selectAll('.bar')
        .data(data)
        .enter()
        .append('rect')
        .attr('class', 'bar')
        .attr('x', function(d) {
          return x(d.letter)
        })
        .attr('y', function(d) {
          return y(d.frequency)
        })
        .attr('width', x.bandwidth())
        .attr('height', function(d) {
          return height - y(d.frequency)
        })
    }
  )
</script>
```

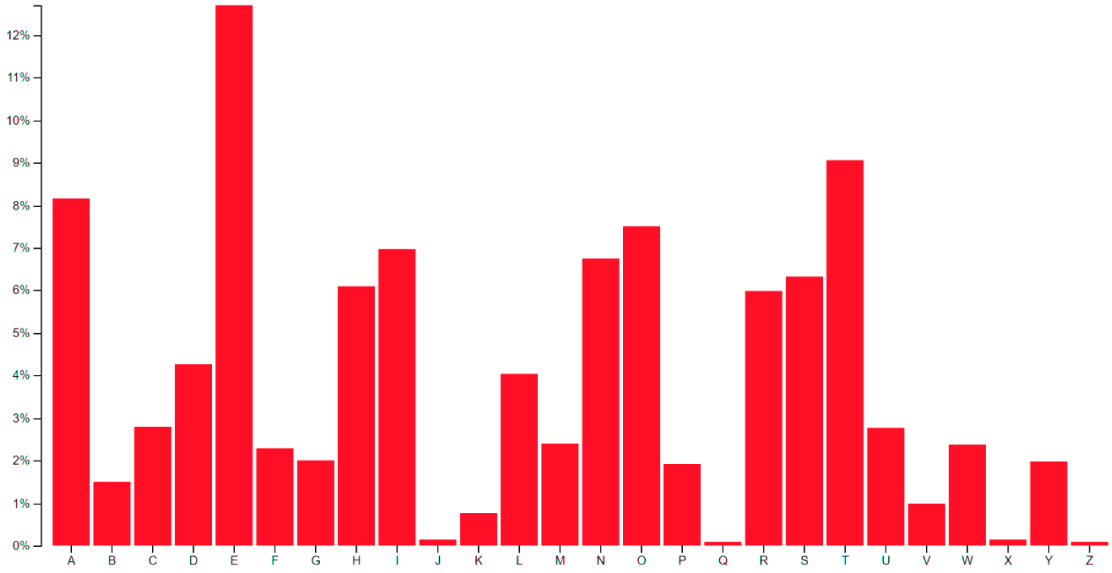


Figure 7: The bar chart generated from the code example in Figure 6.

2.1.3.1 State and declarativity

When we refer to a ***state***, we refer to data that web application depends on to render the page, such as a dataset with some numbers, a server response, or whether the user has selected some option in a web form. Managing this state is becoming hard when we consider how complex are user interfaces nowadays. And it is the same situation with complex data visualizations. In their context, the state can mean whether the user has clicked on some part of the chart what can for example evoke display of a tooltip, how zoomed-in the visualization is, or what data is currently displayed.

Managing the lifecycle of visualization then becomes problematic, when it further combines with D3's ***enter/update/exit code pattern***[11] for binding data to document elements. Adding elements in *enter* callback, manipulating them in the *update,* and removing them in the *exit* means we need to reason about the state transitions of the visualization. We must program how the DOM should change. Moreover, some of the D3 modules save their state in their objects or directly on the DOM.

2.1.4 Future of D3

During the same time when D3 was released, web developers started to create ***frontend frameworks*** to solve multiple issues in the field of web development, such as reusability and organization of code in large-scale projects. New programming concepts were born to aid web development.

---

[11] *Enter/update/exit* is a code pattern that constitutes the core interactions of the D3 library. More about this pattern can be found in any D3 literature, or online: https://medium.com/@c_behrens/enter-update-exit-6cafc6014c36.

One of the most useful concepts that the frontend frameworks have brought was enforcing a declarative approach to rendering the user interface, which meant that the programmer did not have to manipulate the DOM. In frontend frameworks, the DOM is invisible to the programmer and the framework automates its management.

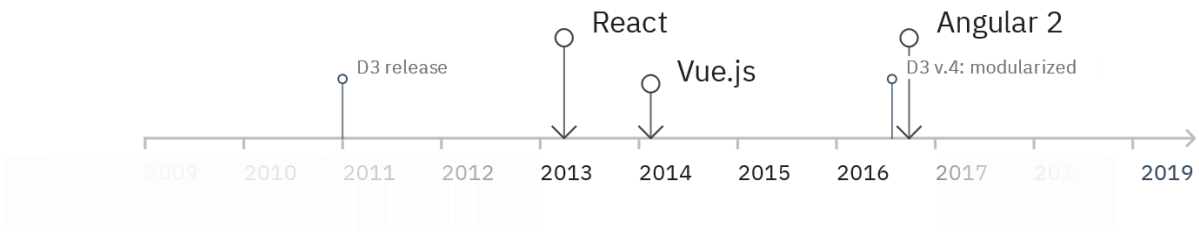
Now if we decide to use D3 with a frontend framework, we will quickly discover a problem: if we try to manipulate the DOM manually using D3, we will interfere with the framework's mechanisms that manage the DOM. We can cause conflicts resulting in application errors. [19, p. 282]

However, thanks to D3's new modular architecture, we can now omit the library's approach to DOM manipulation. A possible solution will be presented in Section 3.1.3.

2.2 Frontend frameworks

Around 2011 when D3 was released, the community of web developers was hoping to solve a significant maintainability issue of the web platform: they needed reusable widgets of HTML; a custom, programmer-defined HTML tags. This would later allow them to compose complex *client-side dynamic*[12] applications from these widgets.

The old, ***page-based*** development approach, where we compose the page from multiple files using a server-side scripting language such as PHP, was hindering the development [20]. Ongoing implementation of a standard that would modularize HTML was slow, and developers turned to create their own solution using JavaScript. [21]



As a response, they have created the frontend framework *React*, which featured a ***component-based*** approach to building web applications. The approach consisted of splitting the code of the application into multiple independent units. Then from these units, a web page was composed on the client side; as opposed to the old page-based approach, where the page was composed on the server side. The designers of the framework have argued that the separation of concerns was

[12] A **client-side dynamic web page** is using scripts running in the browser to provide interactivity, determine the document's content and aesthetics, i.e., dynamically update the DOM. [37]

not about the boundaries between web-standard technologies, but more about separating the units of functionality by creating components. [21]

The component-based approach has proved to be successful, as the web development community adopted it in a large scale.

2.2.1 Concepts introduced in frontend frameworks

The following sections explain concepts, whose knowledge will be essential in the next chapter, where we will be put them in the context of data visualizations. The reason is that they have the potential to improve the development of complex and interactive data visualizations.

2.2.1.1 Component-based architecture

Improves or allows: maintainability, reusability, extensibility, state management, separation of concerns

As previously mentioned, components were the driving force in the history of web development. They are the abstractions that allow us to build large-scale web applications by composing them from small, self-contained, and usually reusable pieces of code – **components** [22].

A component is a self-contained widget with isolated functionality, such as a page header, form, or button, and can contain other, child components. Figure 8 illustrates the component-based approach, which consists of splitting the user interface into parts, which will be represented in the code by their respective components. Then a hierarchy of components can be built to compose the whole application. Parts of the application, which share the same functionality, can be reused throughout the application when made into components.

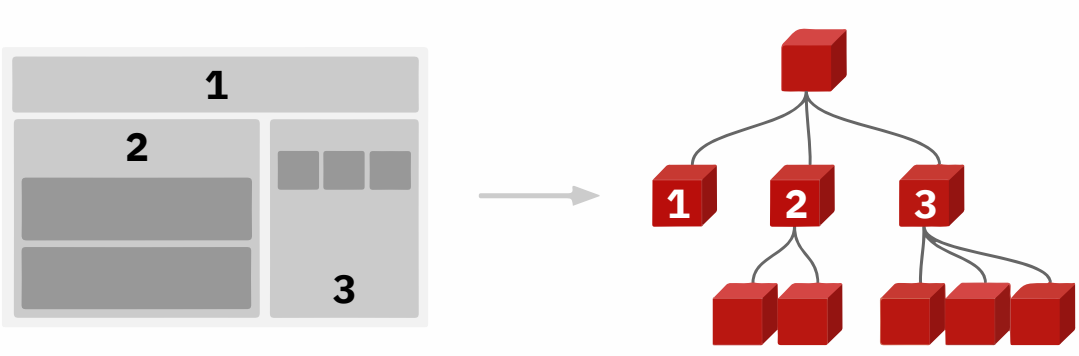


Figure 8: User interface visualized as a tree of components. Vue.js [22]

The component relies on its external input and local data, i.e. state, to render itself. This makes rendering efficient: when the component state changes, we can simply trigger a re-rendering of the specific component. This closely relates to the next concept, reactivity, which allows us to observe and respond to state changes.

2.2.1.2 Reactivity

Improves: code quality, reusability, development speed

Reactivity is a popular concept behind the formulas of spreadsheet software. In spreadsheets, we can define the value of a table cell using a formula that references other cells (such as A1=B1+C1). When the referenced values change (either B1 or C1), the program propagates these changes and recalculates the dependent cells (A1).

The same concept exists in programming: we can express variables declaratively, dependent on other variables. When the program evaluates these variables automatically and keeps their values up to date, we can call them *reactive*.

Major frontend frameworks[13] embrace the reactive programming paradigm, as it dramatically helps when rendering user interfaces based on the application's data. We can illustrate how reactivity works and why it is used on the frontend on an example from Evan You [23].

Imagine we have a variable value calculated using another variable's value, the one dependent on the other. In this example, the variable *b* should be equal 10 times the variable *a*:

26

```
let a = 2
let b = a * 10
console.log(b) // 20
```

In imperative programming, when we define *b* and later change the value of *a*, the variable *b* stays the same.

```
a = 4
console.log(b) // 20
```

Ideally, we would want to recalculate the *b*. Suppose we have a function, which updates the value every time it sees a change in dependencies:

```
onAChange(() => {
    b = a * 10
})
```

The value of b will now be always up-to-date. We can apply this principle to all the data in the application.

Suppose we have an HTML element and we want to set it's content dynamically:

```
<span id="cell-b"></span>
```

[13] React, Angular, and Vue.js.

Let us directly manipulate with the DOM to synchronize the element with the application data. We create a listener, which calculates and sets the element content every time it detects a change in the state:

```
<span id="cell-b"></span>

onStateChange(() => {
    document
        .querySelector('#cell-b')
        .textContent = state.a * 10
})
```

To take this concept even further, we can abstract from handling of the individual elements. In order to do this, we will use another concept, *templating*. A *template* refers to a structure of a document separated from its content using variables (or expressions) and may look as follows:

```
<span id="cell-b">
    {{ state.a * 10 }}
</span>
```

When the application state changes, the program fills the template with new data and then simply re-renders the document:

```
<span id="cell-b">
    {{ state.a * 10 }}
</span>

onStateChange(() => {
    view = render(state)
})
```

We have just achieved the synchronization of the DOM with the application state in a declarative approach. The approach is yet to be discussed.

2.2.1.3 Virtual DOM (VDOM)

Improves or allows: declarativity, the performance of the application

Virtual DOM is a concept where the frontend framework holds a *virtual* copy of the DOM, which is synchronized with the *real* DOM maintained by a browser. This concept is a principal part of building declarative user interfaces, as it allows the frameworks to abstract from direct DOM manipulation. In doing so, it also optimizes the performance.

More accurately, VDOM is a regular JavaScript object maintained by the frontend framework, which represents the DOM. When the framework needs to update the DOM, it does so by mutating the virtual object. This reduces the overhead of subsequent single calls to the browser's DOM API. When the framework decides it is the right moment to synchronize the virtual DOM with the real DOM, it does so using an efficient patching algorithm. These batch updates improve the performance of the application. [24]

2.2.2 Declarativity

The previously discussed concepts of components, reactivity, and virtual DOM management are closely related to the declarative style of programming in frontend frameworks.

Frameworks, in general, are based on the principle of ***inversion of control***, when they control the program flow instead of the programmer. In exchange for the control, they allow the developer to write in a more concise, declarative way, by abstracting the imperative actions in a layer invisible to the programmer. The programmer then, instead of writing *how* to do an action, describes *what* he or she needs.

To quickly illustrate how declarative programming paradigm differs from the imperative one, let us imagine a situation where we are taking a taxi to the nearest train station.

In an **imperative paradigm**, we tell the taxi driver:

> "Start the car, go straight for a few hundred meters, after a red building turn right, follow the street until, …, and stop near the entrance of the train station."

In a **declarative paradigm**, we tell the taxi driver:

> "Take us to the nearest train station."

The reason why frontend frameworks extensively utilize the declarative style is that it naturally fits the use case of programming user interfaces, where we want to describe *what* is the interface, not *how* to render it.

In the **imperative paradigm**, we can display a button by writing:

```
setPosition(pos.x, pos.y);
draw(buttonSize);
rotate(90);
…
```

In the **declarative paradigm**, we can display a button by writing:

```
<button>
```

This results in a cleaner, more concise code.

2.3 Choosing a frontend framework suitable for data visualization

As frontend frameworks were created for building general web applications, they may not always suit the use case of creating data visualizations. This section will introduce two requirements on frameworks that are essential for the FF approach. Then it will present an overview of the most popular frameworks, assessing whether they satisfy the requirements, and based on this, we will choose a suitable framework for building data visualizations.

2.3.1.1 Requirements

1. It should be focused on the presentation layer/rendering, which is crucial for building visualizations, as they consist of visual elements.
2. The framework needs to be lightweight and accessible: easy to learn for new developers, with minimal prerequisites coming from plain web-standard technologies. The approach needs to be as open to data visualization practitioners as possible.

2.3.1.2 Most popular frameworks compared

The frontend space is currently dominated by frameworks *Angular*, *React*, and *Vue.js*, as stated in the most recent JavaScript developer survey [25]. All these frameworks are component-based and use a similar architecture for separation of concerns.

Angular is focused on providing an extensive set of functionalities needed for web development of large-scale applications, which subsequently makes the framework rather complicated, and not focused on the presentation layer. A prerequisite for developers is the knowledge of TypeScript[14] , which, together with the complexity of the framework's API, adds up to a hard learning curve. The framework is also stringent in the context of the developer's workflow. Overall, the developer's survey states[15] that it is probably best to avoid Angular in new projects.

React and Vue.js are similar in their focus on the presentation layer, i.e., building user interfaces[16], which makes them more lightweight. One difference between the two frameworks lies in the syntax used to express the UI. In React, we do not write HTML: we express documents in a declarative XML-based syntax using a JavaScript extension called *JSX*. Meanwhile, Vue provides an HTML-based templating system, which uses a domain-specific language that adds additional

[14] TypeScript is a superset of the JavaScript programming language with the addition of static type system. The language transcompiles to JavaScript.

[15] The conclusions from the State of JavaScript survey can be found at https://2018.stateofjs.com/front-end-frameworks/conclusion/.

[16] React and Vue.js focus on the *view-model* layer of *MVVM architecture* used in frontend frameworks.

syntax to HTML. When compared to JSX, Vue's templating system has a lower learning cost, as it stays close to HTML. It can make Vue more accessible for data visualization practitioners that only know plain HTML and JavaScript. [26]

Amongst the three frameworks, developers are stating [25] that Vue.js has the easiest learning curve. The results of the survey also show this as its most significant advantage. Official documentation states that "it is possible to start building non-trivial applications within less than a day of reading the guide" [26]. One of the reasons why Vue is easy to learn is that many of its features are incrementally adoptable. For example, the developer can optionally choose to write in TypeScript, use his preferred CSS preprocessor such as SASS, or integrate other libraries. This freedom, together with its HTML-based templating system, makes it easier for designers and less experienced developers to contribute to the codebase.

2.3.1.3 Conclusion

From the research made in the area, this thesis concludes that Vue.js framework meets the requirements set earlier and could be a good candidate for supporting data visualization practices. To sum up the reasons:

i. It has an easy learning curve.
ii. Data visualization practitioners accustomed to D3 and plain web technologies can transition well into Vue's ecosystem.
iii. It focuses on the presentation layer of the application (rendering) and is very free in terms of how a developer can use it, which makes it ideal to test new use cases, such as interactive visualizations.

Building a complex data visualization in Vue.js will be demonstrated on a case study presented in Chapter 4.

3 Frontend frameworks as visualization tools

It is not uncommon to see the usage of the frontend frameworks in data visualization practice. They are typically used for combining multiple visualizations into ***dashboard*** applications. Dashboards present multiple views on data in individual charts that are sharing the same screen space. The charts, usually created using D3, are combined and linked together using the frontend framework. An example of such dashboard is shown in Figure 9. [19, p. 276], [27]

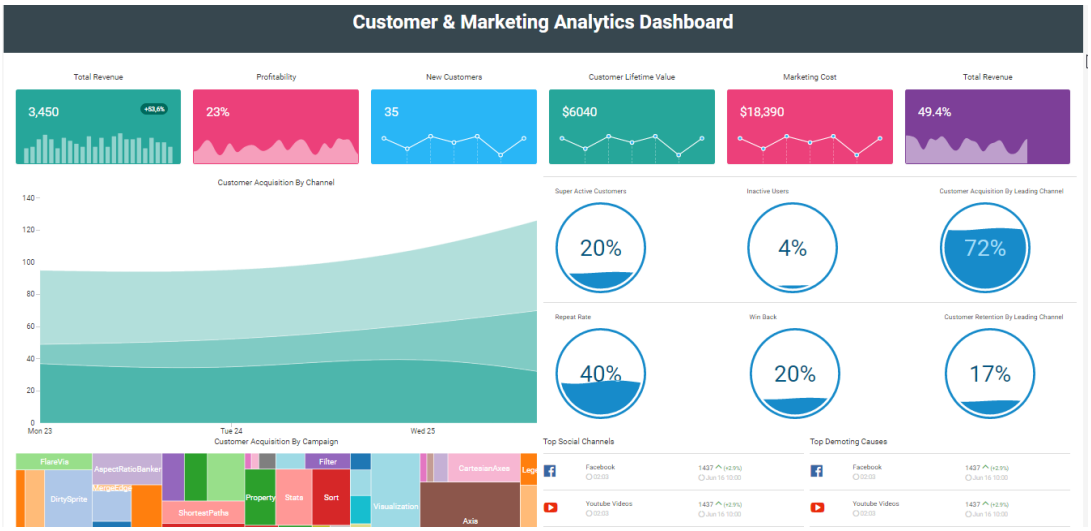


Figure 9: A generic example of a dashboard application. Dashboards are typically used in decision supporting systems – information systems that help businesses or organizations make problematic decisions based on the data. Wikimedia Commons [43].

However, when it comes to more complex data visualizations, frontend frameworks still have a lot of untapped potential. Specific frontend frameworks can be useful in the development of complex data visualizations, where the framework acts not only as a supporting structure for the (D3) visualization but as the core of the visualization itself.

This chapter presents an approach to building visualizations primarily using frontend frameworks and minimalizing the use of D3: only the library's data visualization modules. We will first discuss a possible pitfall of simultaneously using a frontend framework and D3 in a single application.

3.1.1 Compatibility issues of D3

The problem that arises when combining frontend frameworks and D3 is that both need to access the Document Object Model, and work with it in different ways, as we have seen in Section 2.1.1. As previously mentioned, in D3, we directly manipulate the DOM by adding, updating, or removing elements. The frameworks, on the other hand, do not even provide tools for such manipulation, as they abstract from it. They, instead, want to *own* the DOM, handling the updates to the user interface, so we do not have to. So when the framework synchronizes its virtual DOM with the real DOM, it assumes it has its current version. However, if we

simultaneously modify the real DOM using D3, then in most cases[17], we interfere with the framework's DOM management system. Therefore, our changes to DOM using D3 tools result in errors, as the framework supposed that it had the sole control over the DOM. We can refer to this as the ***DOM ownership problem***.

3.1.2 The blackbox approach

A common approach that avoids this problem consists in using the framework only to build a supporting structure for the application, not the visualization. We use D3's DOM manipulation to create the D3 visualization as a blackbox on a chosen container element, such as *<svg>*, inside which will D3 operate [27]. The element then remains empty from the framework's perspective, and the framework does not manage that part of the DOM, so we avoid any conflicts. We, however, lose most of the benefits of using the frontend framework.

3.1.3 The frontend framework (FF) approach

This thesis presents a more practical, yet still largely unexplored approach which consists of fully embracing the frontend framework features explained in Section 2.2.1. We will call this approach the frontend framework, or the ***FF approach***.

In this approach, a frontend framework is used wherever it is possible and effective, and D3 is used minimally and where justified: the library's DOM manipulation tools are completely avoided, and only its data visualization modules are used to calculate drawing instructions, such as axes on charts or positions in topological layouts. We will look at what this means in detail.

Direct DOM manipulation using D3's tools[18] is avoided for two reasons. The first reason is to avoid the **invisible document structure** stated in Section 2.1.3. The second reason is that most of the DOM manipulations will not be possible as they would be causing application errors. Every developer working on the application would then need to deeply understand the quirks of the D3's DOM manipulation while simultaneously using a frontend framework, and we want to avoid that too.

Nevertheless, the most important benefit of using frontend frameworks is that they can solve issues related to the rising complexity of visualization. The complexity demands more sophisticated solutions for code structure and application architecture, a clearer separation of concerns, more reasonable state management, and others. These issues cannot be solved just by using

---

[17] In some cases, simultaneous DOM manipulation with D3 is possible without issues, depending on the implementation of the specific framework's DOM management system. For example, changing attributes of an element that already exists in DOM may not cause any conflicts.

[18] D3's tools for manipulating DOM include d3-selection, d3-selection-multi, and some of the interaction modules, for example d3-zoom or d3-drag.

a visualization library, and frontend frameworks offer several concepts that can help. A summary of the advantages will be given towards the end of this chapter.

3.1.3.1 Declarativity

When creating a component in frontend framework, we do not specify how it should change when underlying data changes. We instead declare how a component should be rendered, given the current state. When state changes, the component simply re-renders. We can use this to turn the whole visualization into a function of data, so we do not have to program state transitions.

Transitioning to this declarative approach for programming visualizations may be difficult when coming from D3, however, the benefits may become worthwhile, as the approach can reduce the size of the code, make it more readable, and prone to errors. Ian Mundy [28] provides practical advice on the required mindset of a programmer: “When writing in a frontend framework, it is often good not to think about how to accomplish the result, but instead think about how the component should look like in its new state.”

3.1.4 Related work

So far, only few case studies have been conducted on the FF approach. Also, there is a lack of examples of visualizations built using this approach. As the field of web development is advancing quickly, most recent information is usually not yet in the literature.

3.1.4.1 Shirley Wu: Illumination case study (2015)

The case study conducted by Shirley Wu [29] states how declarativity and state management of the React framework have aided in rebuilding a commercial visual tool Illumination. She describes how the tool heavily used D3, and its codebase became unmaintainable with the rising complexity of the application.

3.1.4.2 D4.js (2016)

D4.js[19] [30] is an experimental project that replaces the DOM interactions in D3 with a frontend framework to show that this approach is possible.

The author argues [30] that by using React, we can improve the performance of the application, readability of the code, and benefit from the ecosystem of tools built around it.

3.1.4.3 Semiotic (2017)

Semiotic[20] [31] is a data visualization library created by Elijah Meeks. It predominantly uses React and provides a well-crafted set of more complex

[19] Available at https://github.com/joelburget/d4

[20] Available at https://semiotic.nteract.io/.

visualization methods, as reusable React components. According to Meeks, minimalizing the use of D3 in the codebase still allows the same flexibility when creating custom visualizations, and in addition, we can benefit from better maintainability, reusability, and accessibility for regular web developers not experienced with D3.

3.1.5 Advantages of using a frontend framework with D3

i. **The frameworks are particularly effective at building user interfaces.**
Specific frontend frameworks (such as React or Vue.js) were created to be efficient at building user interfaces. A well-crafted UI can be vital for user interaction in complex data visualizations.

ii. **The frameworks offer reactive data flow.**
When the visualized data change, reactivity, instead of the programmer, can take care of updating the visualizations. The reactive updates are incredibly useful in the context of interactive data visualizations[21], as it is common for visualizations to allow users to manipulate the underlying data. Moreover, the reactivity can facilitate the linking of multiple visualizations, or their parts: interacting with some parts may update the others automatically. Reactivity also speeds up development time, results in a cleaner, error-prone code, and improves reusability, as stateless definitions are easier to share [17].

iii. **The frameworks offer tools for state management.**
Systems for state management may be necessary when developing complex data visualizations. An example may be the *Flux pattern*, and the next chapter will describe a library, which implements this pattern for the framework Vue.js.

iv. **The frameworks allow componentization of visualizations.**
Developing a complex visualization can be easier, if we split its parts into components. These components can be reused, improve separation of concerns, and overall maintainability of a codebase [32].

v. **The frameworks improve readability[22] of the code.**
D3 code is generally considered as hard to read, as it builds the document through a series of commands which generate elements. With the framework, we can instead use a templating system to write clean document structure.
Even though the official web of D3 [33] describes the library as “employing a declarative approach,” the description can be deemed deceptive, as DOM

---

[21] *Observable*, a new web-based visualization platform for writing and sharing of JavaScript snippets, embraces the concept of reactivity. Its co-founder (and the author of D3) Bostock [17] acknowledges the importance of the concept in data visualizations.

[22] Readability of a code refers to a programmer’s judgement on how easy it is to understand it. It relates closely to maintainability and is considered as fundamental factor in overall software quality [50].

manipulation is a manipulation with the state (which is represented by the DOM itself, instead of variables). The declarative approach of frontend frameworks is superior to D3's, as we express the UI/visualization in a more concise and effective way.

vi. **The frameworks improve reusability of the code.**
D3 is a library and thus cannot enforce application structure, whereas frontend frameworks force the developer to abide by a standardized code structure. The framework's key feature are reusable components.
The standardized component-based architecture improves maintainability, makes it easier for other developers to get familiar with the code, and it allows the possibility to share components using *packages*[23].

vii. **The frameworks can improve the performance of the application.**
Frontend frameworks optimize DOM updates by using Virtual DOM (Section 2.2.1.3).

viii. **Data visualization practitioners programming in D3 can collaborate with frontend engineers, and vice versa.**
Frontend engineers that are not familiar with D3 will be more inclined to collaborate on data visualizations. The codebase can be maintainable by the whole team.

3.1.6 Disadvantages

i. **Lack of published examples.**
The abundance of frontend frameworks divides the community, which means a lesser chance of finding the right example in the chosen framework.

ii. **Animating the user interface becomes harder.**
At the moment, the layer of abstraction provided by the frontend frameworks for dealing with user interface means it is harder to animate document elements, as we are not directly dealing with them. D3's transitions and animations are more natural to work with, as they can be expressed when programming state transitions.

[23] Packages are a way of sharing code modules between developers using a registry, such as *NPM*, the public registry of open-source JavaScript code.

4 Case study of network visualization module for the research project Analysis

This chapter presents a case study of a visual analysis tool for criminal investigation, to demonstrate how the **FF approach**, explained in the Section 3.1.3, together with the frontend framework Vue.js aided in its development.

The presented tool is a part of the system developed in a research project "Complex Analysis and Visualization of Large-scale Heterogeneous Data" (**project Analysis**). This work is a follow-up on research conducted by Kristína Zákopčanová [7] during her master's studies, which focused on the design if the visual analysis tool of project Analysis.

The following sections will introduce the goal of the project, describe the visual analysis tool, and summarize the research and development of the tool.
The technical sections assume that the reader has a knowledge in the extent of the previous chapters and is at least partially familiar with the frontend framework Vue.js.

4.1 Project Analysis

The primary goal of the project [34] is to create a system for analysis of ***big heterogeneous data***[24] that are collected by the national police agencies.
The system will provide a unified tool to aid criminal investigations, allowing the police forces to access, analyze, visualize and collectively share relevant data or information.

The reason why analysts need a unified tool is that currently, they must use several separate tools, which makes their workflow inefficient. Therefore, the goal is to propose a new system that will speed up the investigative process by integrating all the required tools in one place.

The system proposed by the project consists of three components:

i. ***Central Data Storage*** – a centralized storage platform to access and manage big heterogeneous data gathered by the police
ii. ***Analysis modules*** – a set of data analysis and transformation tools, which automate the processing of raw data often carried out manually by the analysts, such as advanced image or text recognition techniques
iii. ***Visual analysis tool*** – the frontend of the system with multiple data visualization modules to allow exploration, effective decision-making, and reasoning about the data in a way that is intuitive and comprehensible to the analyst

This thesis relates to the last of the listed components, the visual analysis tool.

---

[24] The term *big heterogenous data* refers to large (big data) datasets containing entries of multiple types. In our case, it can refer to metadata about phone calls, scanned documents, or financial transactions.

4.2 The visual analysis tool

"The visual representation of data allows for much faster information processing and pattern recognition. By extension this helps us gain insight into the data that leads to better explanation and decision making. With the wide range of built-in functionality, the network visualization forms a powerful tool for effective and efficient visual analysis." [7]

The visual analysis tool of the project Analysis intends to **help the decision-making process** of analysts by providing them a way to visually represent, explore, and examine data related to the criminal investigation process. In order to present the data, we first need the system's analysis modules to preprocess them. These modules then take raw data gathered by the police and prepare datasets that we can visualize using the tool. These datasets can then be opened and explored using our visualization.

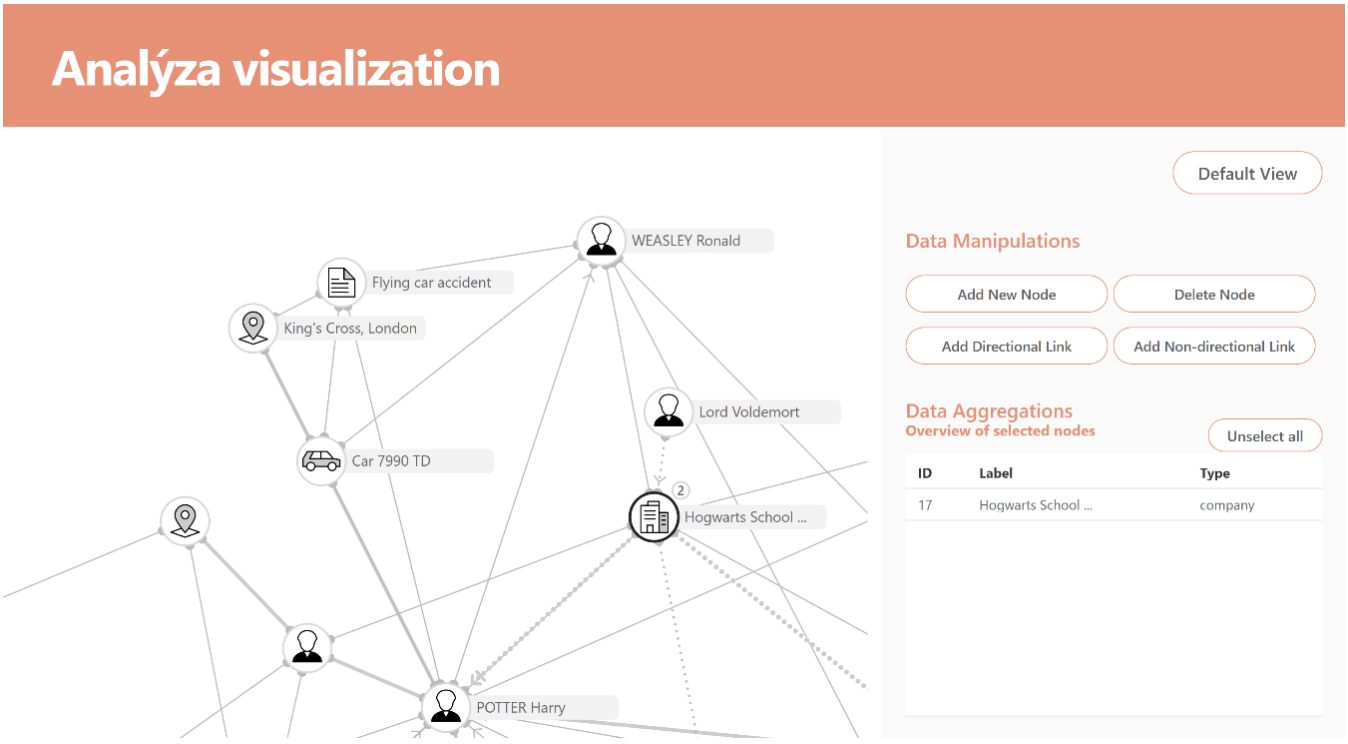


Figure 10: Zákopčanová's network visualization prototype for the project Analysis.

4.2.1 Previous work

In her diploma thesis, Zákopčanová [7] studied the workflow of the analysts in order to design the new tool. She had analyzed the requirements for the visual analysis component and then created a comprehensive system specification of the tool's features and visuals. To test her concept, she has built a web-based prototype demonstrating the core feature of the module: the *network visualization*, shown above on Figure 10.

4.2.2 Network visualization

The datasets that analysts work with contain several types of entries modeling real-world phenomena, such as information about people or their communication, documents, organizations, or financial transactions. Zákopčanová [7] describes that the data entries can take two general forms (illustrated in Figure 11):

i. **entities** representing the real-world objects (such as people, buildings, or documents)
ii. **relationships** between entities (such as personal connections, actions, ownership, or usage).

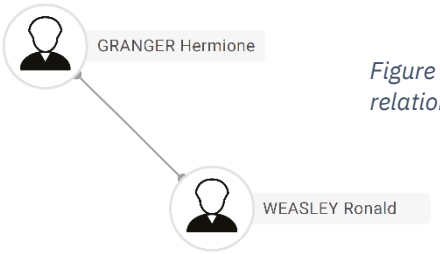


Figure 11: Visual representation of two people in a relationship. Zákopčanová [7] Figure 4.5

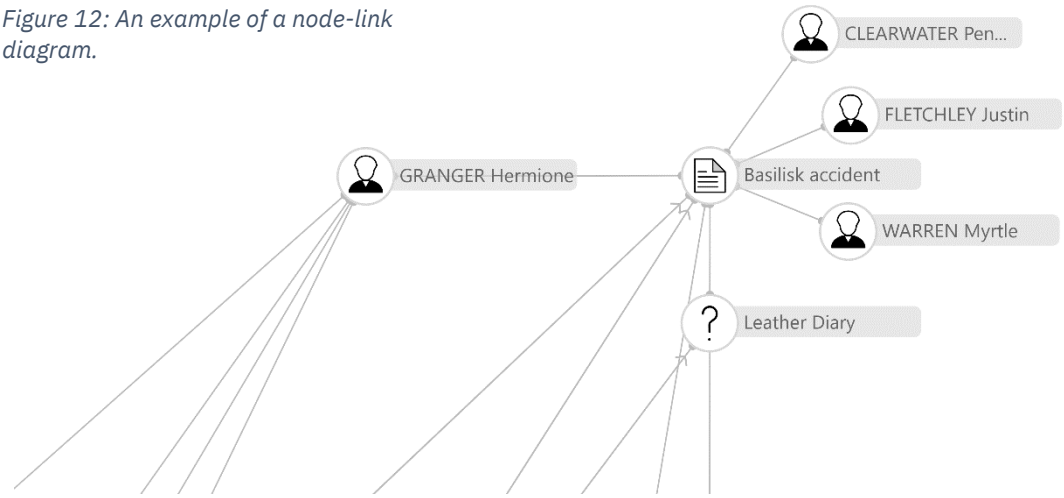
The entries of these two forms are then further specified by additional attributes. For instance, an entity can have an attribute specifying its *type*, such as *person*, and then have several other attributes, specifying the person's *name, national ID, birthdate,* or *address.* After the analysts choose a dataset, they can open and explore it using the tool's primary feature: the *network visualization*, depicted in Figure 12. It takes a form of a node-link diagram, a method well-suited for the two types of data entries described earlier. The entities in the network are shown as nodes (circles), and relations between them are shown as links (straight lines).

4.2.3 Previous prototype

The prototype has successfully demonstrated the design and key features of the visualization. Its implementation used the blackbox approach mentioned in Section 3.1.2, and the code[25] had several issues.

[25] The original code is available at https://is.muni.cz/auth/th/ujzyx/.

Figure 12: An example of a node-link diagram.



One of the issues was visible on the complexity and size of the *ForceLayout* component, which had 2680 lines of code and was difficult to refactor into smaller components. The component size amplified the **invisible document structure problem** state in Section 2.1.3.

Furthermore, the visualization was supposed to allow users to manipulate the data in several ways. As the data were bound to the DOM, there was no clear separation between them and the visualization itself. This called for the abstraction of data into a separate layer, in order to allow cleaner data manipulation, outside of the visualization.

The next goal in the project (and one of the goals of this thesis) was to evaluate the development approach and tools used in the building of the earlier mentioned prototype, and come up with a new approach, that would support all the requirements for the visual analysis component and that will easily allow feature extensions.

4.3 Building the network visualization module

The following sections describes the development of the network visualization module of the project Analysis' visual analysis tool. First stated are the software requirements, secondly the selected technologies, and lastly the system architecture design of the tool.

4.3.1 Requirements for the network visualization module

In the section 3.1.3 of her thesis, Zákopčanová [7] defines high-level requirements for the visualization component of project Analysis. She further states which of the requirements for the component also apply to its network visualization module, and then defines specific functional requirements for the network visualization, in the 4th chapter of her thesis called "Network Visualization." We will now focus on the specific requirements and primarily interactivity, as the goal of this case study

is to show that the new architecture will support the designed interaction techniques.

We need to consider the defined **higher-level requirements** in the design of the tool's architecture. Several of the requirements suggest that the resulting application will become complex and will inherently need a well-considered supporting structure. These requirements include **workspaces** (personal spaces for users to manage their work), **customization** (ability of the user to modify their workspace to their needs and workflow), **team collaboration** (sharing of information, new evidence, and investigation progress between analysts), or **report visualization** (to present evidence to third parties). We believe that the new architecture, which uses a frontend framework, will be suitable and efficient in accomplishing these tasks. Even though these requirements were considered in the design of the architecture, we will not discuss them further in detail, as they are part of future work.

The new architecture (and the corresponding implementation) is focused on finding ways how to support the interactivity of the visualization. By interactivity, we mean "analytical navigation, real-time exploration of data, easier orientation in the visualization, and interactive manipulation with the data and the visualization" [7]. Besides this abstract definition, Zákopčanová [7] also described the interactivity of specific tasks that the visualization needs to support. The following section lists four of these tasks, which we chose as the relevant ones to test out the architecture, and their basic functionality is implemented in the new prototype of the tool. The functionality will be implemented to the full extent in the future of the project. In the next section, each of the four tasks is briefly explained, illustrated on an image with a description of the user interaction along with information on what components of the implementation relate to them.

4.3.2 Functional overview

4.3.2.1 Local exploration of data (details-on-demand[26]27)

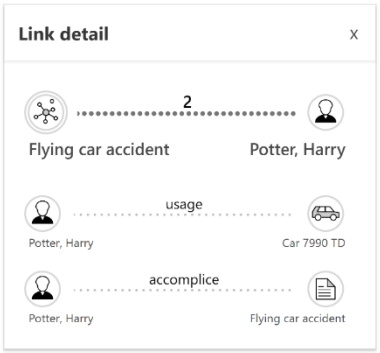


Figure 13: The implementation. Link detail card illustrating the details-on-demand technique.

Users can request detailed information about visualized entities and relationships.

In the prototype: Double-clicking on nodes or links opens a card with detailed information about the related entities, as depicted for a link in Figure 13. Hovering a node opens a card that lists attributes of the entity.
Related components: *NodePopover, Tools/NodeDetailCard, Tools/LinkDetailCard, data.store.*

4.3.2.2 Selections

Users can create selections of nodes and later operate on them.

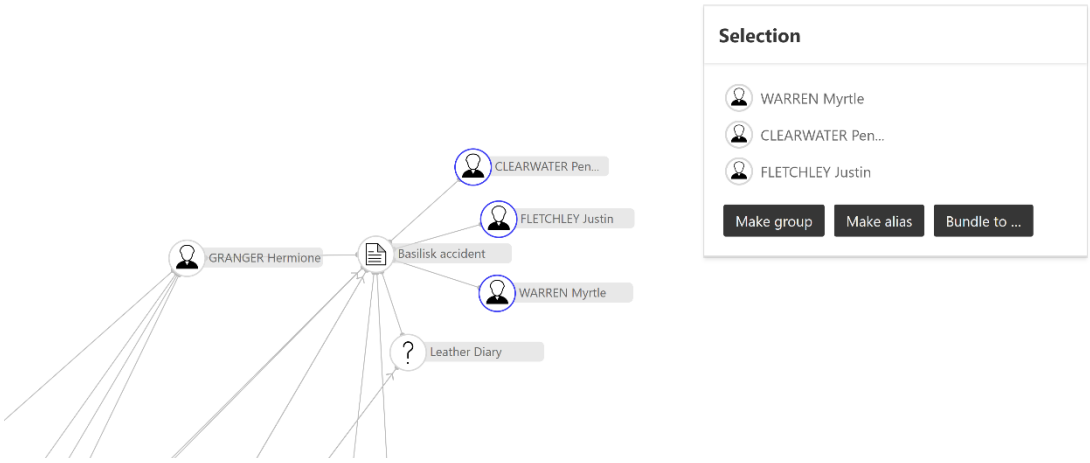


Figure 14: The implementation. Node selection card with three selected nodes.

In the prototype: clicking on nodes adds them to the current selection and allows the user to *aggregate* them, *as shown in* Figure 14.
Related components: *NodeSelection, NetworkGraph/GraphNode, interaction.store.*

---

[26] Details-on-demand is an interaction technique that allows users to request more detailed visualization of selected parts of data.

4.3.2.3 Node aggregations and labelling

Users can organize nodes into *groups* or *aliases* and label the joined nodes with a caption that is displayed alongside the node. The users can also *bundle* the nodes, hiding them inside the others. The meaning of these three types of *node aggregations* will be explained in detail later in Section 4.4.1.

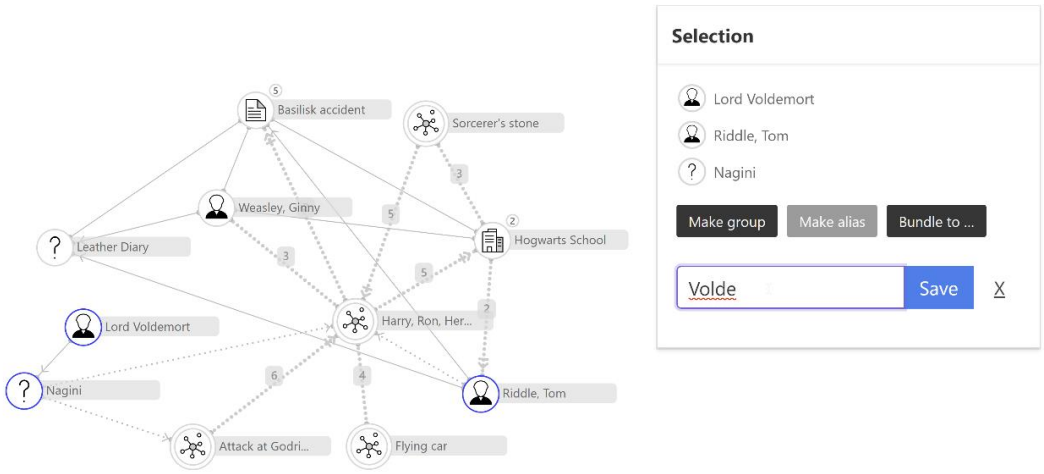


Figure 15: The implementation. Graph illustrating several node aggregations. The user is creating an alias for Voldemort.

In the prototype: selecting at least two nodes allows the user to create a group or alias them in the *NodeSelection* card, selecting at least one node allows to bundle it (Figure 15).

Related components: *NodeSelection, NetworkGraph, data.store*.

4.3.2.4 Focus and context together

Users can request visual hints on neighboring nodes, which facilitates an analyst's orientation in the visualization, especially when it is crowded.

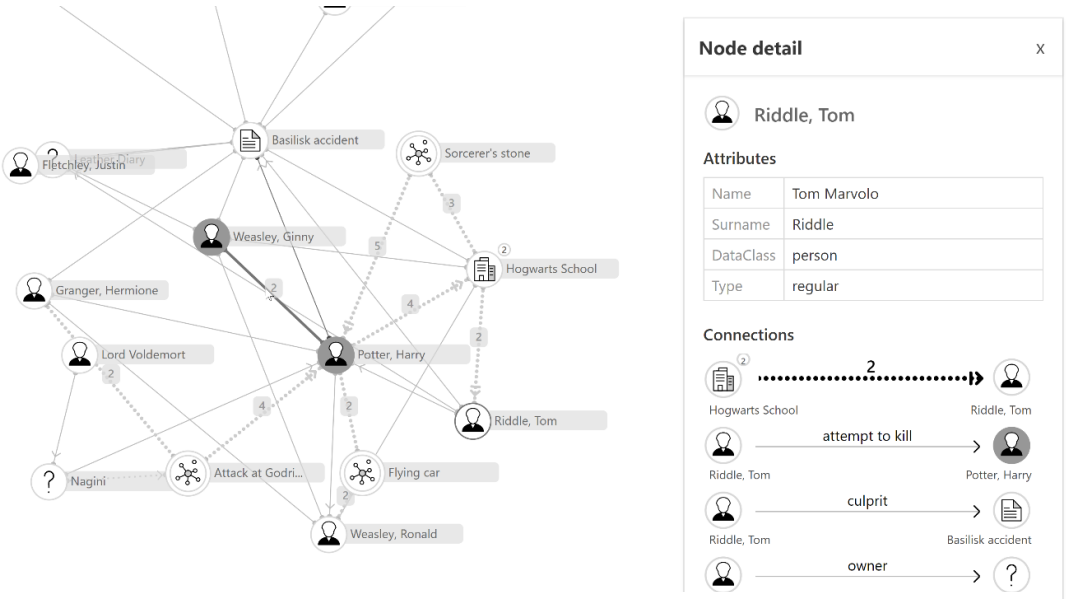


Figure 16: The implementation. Using a cursor, we have focused on the link between Harry and Ginny. The visualization highlighted the related entities.

In the prototype: holding *Control* key and hovering over nodes/link triggers visual aid (Figure 16).
Related components: *NetworkGraph/GraphNode, NetworkGraph/GraphLink*, *interaction.store.*

4.3.3 Technologies

This work focuses on the implementation of the network visualization, however, in the future, the visual analysis tool of the Analysis system is intended to provide several visualization modules which would connect to the Analysis system's other components. Therefore, we take into account that we are designing a large-scale application, and we are choosing the technologies accordingly.

4.3.3.1 Vue.js (Vue): frontend framework

We have chosen Vue.js as the core of the new visual analysis tool, as this case study wants to prove it can be used to effectively build complex data visualizations. Vue fits our use case well, as we have shown in Section 2.3.

4.3.3.2 Vue CLI: project scaffolding tool

Vue CLI is a tool that helps the developer to start a new project quickly. Using Vue CLI, the developer can avoid dealing with complex configurations of build tools, such as *Webpack*, as it provides him with a deliberately pre-configured starting point. It acts as an abstraction over the build-related tools. The implementation featured in this thesis did not require changes to the default configuration.

4.3.3.3 Vuex: state management

A problem well-known to the development of large-scale web applications is managing their state. When the application consists of multiple components that share data, "the complexity of their interconnections will increase to a point where the state of the data is no longer predictable or understandable" and "the application becomes impossible to extend or maintain" [35].

Vuex is a state management library[28] for Vue.js applications that solves these issues and is fundamental to our application. The library provides centralized storage for the application's data with a set of programming principles, which ensure that the state can only be mutated in an orderly fashion. Vuex features a ***store***, an object, which the programmer uses to access the data. One of the benefits of using Vuex is that it is reactive, the same as Vue components: any components reading data from the store will be updated automatically, if the underlying data change.

[28] Vuex can be primarily seen as a partial implementation of the *Flux pattern* (an observer-like pattern) and its principles.

4.3.3.4 Jest: testing library

Major part of the application's store data module and primarily the functions related to the aggregation algorithms (later introduced in Section 4.4.2) were developed using a test-driven approach using the testing framework Jest. We found the framework very approachable, as it worked out-of-the-box and offered valuable context when tests failed.

4.3.3.5 D3: visualization

Three of the D3 modules are used, yet minimally and only in the component *NetworkGraph*:

- *d3-force*, to calculate layout for the nodes and links
- *d3-zoom*, to allow zooming and panning of the <svg> rendering the network
- *d3-select*, to aid in the setup of the zoom/pan behavior

4.3.3.6 Buefy: user interface library

Buefy is a UI component library with Vue components created on top of the CSS framework *Bulma*.

4.3.3.7 SVG: 2D graphics

The network visualization is rendered using SVG. It enables us to do a minimal setup for the zoom/pan behavior and click/hover/touch interactivity when compared to HTML5 *canvas*. However, its rendering effectivity may limit the size of the displayed dataset, and when needed, we will consider switching to the canvas. The modularity of the implementation supports that.

4.3.4 System architecture

A high-level overview of the network visualization module's architecture is depicted on Figure 17. Currently, the application does not use routing, and we can consider *Workspace.vue* as its root component, from which we can start to explore the application's structure. The **Workspace** contains the **NetworkGraph**, which represents the node-link diagram; and **Tools**, components, which provide users with the functionality described earlier in Section 4.3.2, such as selecting nodes or requesting a more detailed visualization.

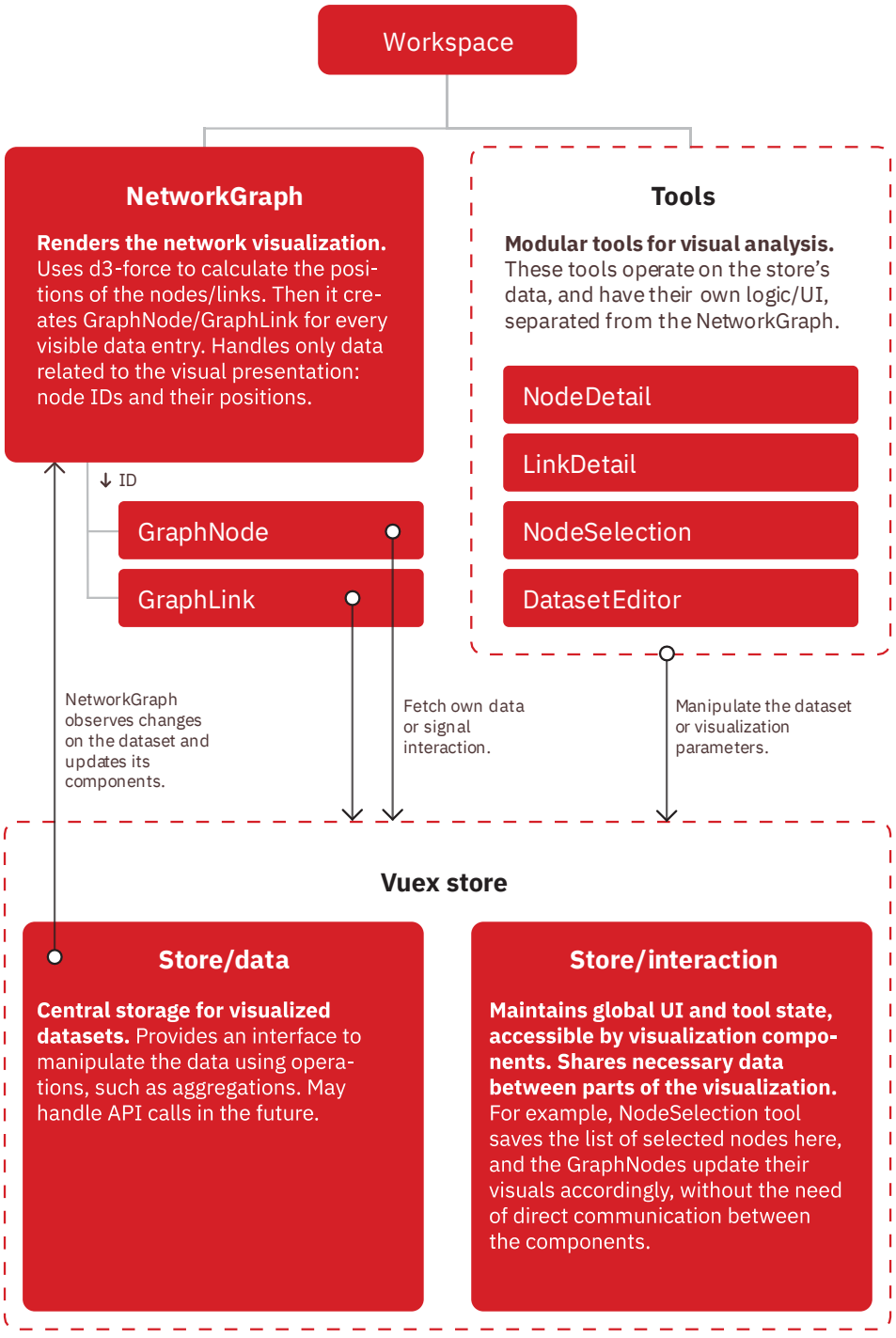
Furthermore, the implementation uses a Vuex store to manage shared state. The first reason to use this centralized state management is that all components must access the visualized dataset through the store, and thus can be reactively updated when data change. For example, if a user groups a set of nodes using the *NodeSelection* component, the *NetworkGraph* will remove the old nodes and renders the new group node without a need to have a direct order from the *NodeSelection*.
The second reason is that it allows user interactions to affect several parts of the visualization. For example, double-clicking a node, which is handled in the *GraphNode* component, toggles the visibility of the *NodeDetailCard* component,

which reads the currently "selected" node from the global state. Therefore, we divided the store into two modules:

- ***store/data***, which abstracts the access to the visualized dataset; and provides methods to mutate the data, such as adding new nodes/links, updating their attributes, or aggregating them

- ***store/interaction***, which contains the state of the application UI and data shared between visualization tools, such as currently selected/hovered nodes or links

Figure 17: A high-level overview of the network visualization module. The Workspace is the root component.



4.3.5 Data flow

As we have previously mentioned, the visualized dataset is stored in the *store/data* module. This module represents a data layer, to which components can bind to access the data. This module is the ***single source of truth***[29] for the dataset it holds, as components reference to the data entries using their IDs, and retrieve the data using the module's getters. The getters allow the components to have read-only and reactively updated instances of the data entries.

4.3.6 Data model

The following section describes ***JSON***[30] structure of the visualized dataset. Earlier in this chapter, we described that the network visualizations present entities and relations between them using nodes and links. These are general representations, and both nodes and links need additional attributes for visualization purposes, and also to specify their meaning in the context of our use case, criminal investigations. Figure 21 and Figure 20 show examples of the data entries in JSON.

4.3.6.1 Node

There are three categories of nodes: regular nodes, groups, or aliases. Group and alias nodes are created during aggregation, and they temporarily replace the aggregated nodes. Any of these nodes can hold a bundle or be bundled under another node. The following attributes model the node:

- *number* **ID**: a unique identification in the set of nodes
- *string* **type**: *regular*, *group*, or *alias*
- *string* **label**: the caption displayed alongside the node
- *object* **vis**: stores attributes related to the visualization
    - *number* **replacedBy** ID of a presentational node which replaced this one during aggregation (meaning this node is inside group or alias)
    - *number* **bundledIn** same as replacedBy, but related to bundling
    - *number[]* **bundledNodes** IDs of nodes bundled behind this one
    - **Graph** layout data
        i. *number* **x** position
        ii. *number* **y** position

[29] In the theory of information systems and design, the s*ingle source of truth* is a practice, where data have only one instance, and if we want to access it, we reference.

[30] *JSON* is a popular data format, which has a human-readable self-describing syntax to describe arrays and objects made of attribute-value pairs. This format represents JavaScript's prototype objects.

Figure 18: The component hierarchy of the application.

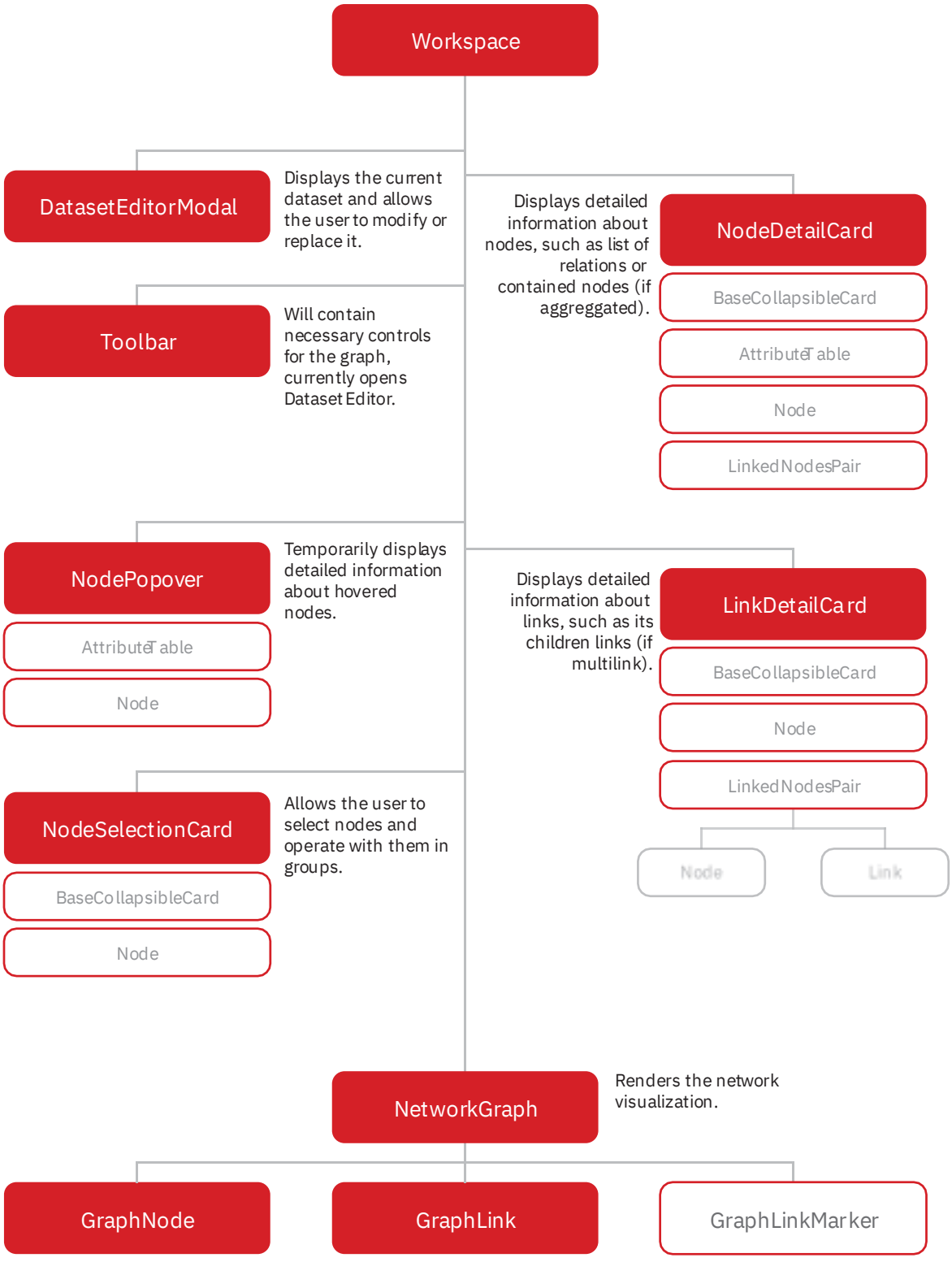
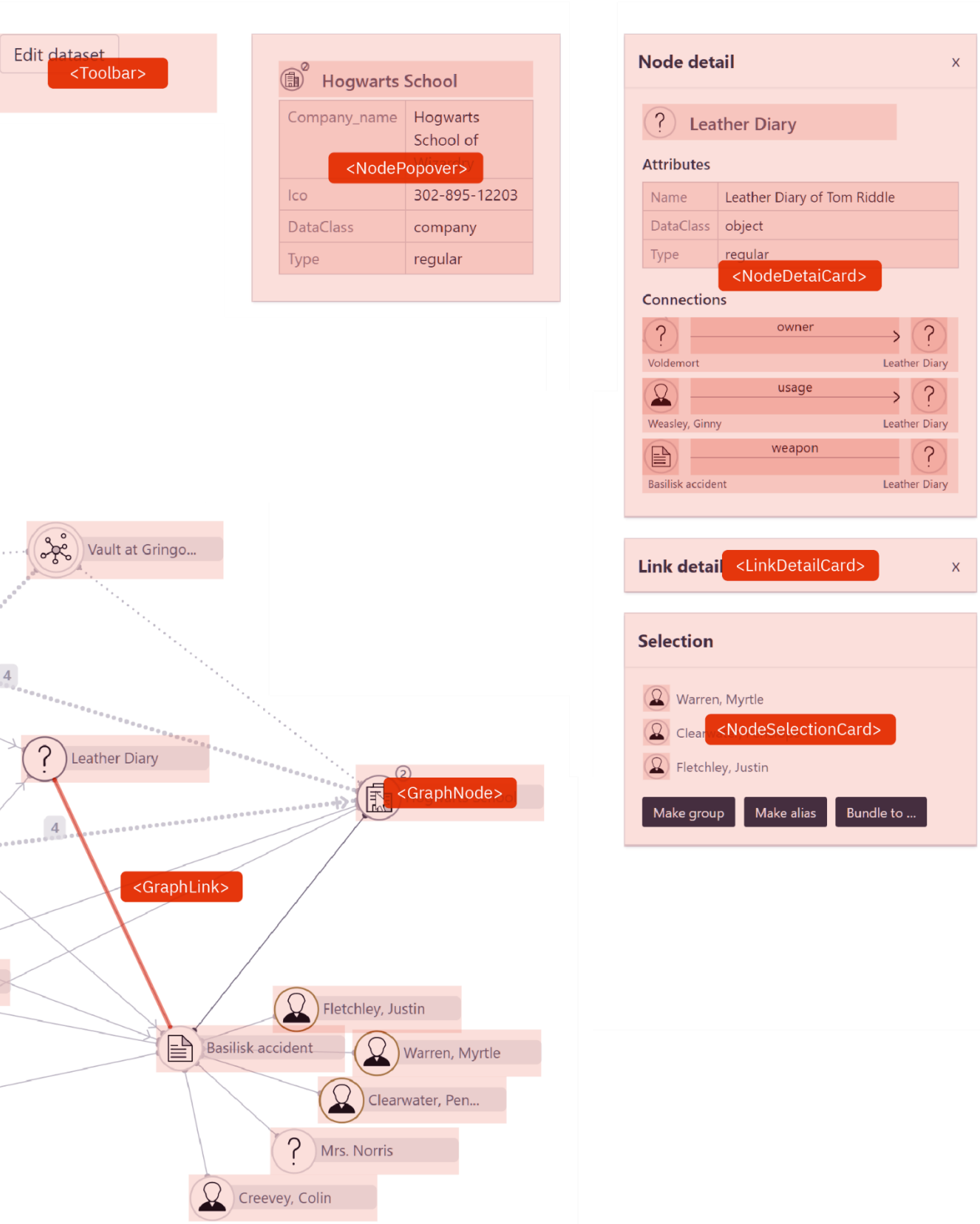


Figure 19: The UI of the application with visualized components.



During aggregations, the nodes which are being aggregated stay in the dataset and are marked using the *replacedBy* and *bundledIn* attributes. The attributes show that they have been replaced, but are still accessible, and the related aggregation can be reversed.

4.3.6.2 Link

Link represents relations between two entities and can either take a form of a ***singlelink*** or a ***multilink***. Singlelinks represent exactly one specific relation, and multilinks represent many, as they aggregate singlelinks. Multilinks are generated by aggregations in order to prevent overlapping links in the visualization but can also carry information from the analyst. The following attributes model the link:

- ID: a unique identification in the set of nodes
- *number* source: ID of the first end node
- *number* target: ID of the second end node
- *boolean/object* direction:
  if the link is multilink, it contains *object* aggregating the directionalities of contained singlelinks; otherwise it is *boolean* specifying, if the relation is directed from source to target
  - *boolean* nonDirectional
  - *boolean* sourceToTarget
  - *boolean* targetToSource
- *object* vis: stores attributes related to the visualization
  - *number* newSource: if the link was transferred during aggregation, this number replaces the ID of the source node, i.e., *newSource* points to the node which the link is currently connected to
  - *number* newTarget: same as *newSource*, but for the second end node
  - *number* inMultilink: ID of the link which replaced this one during aggregation
  - *boolean* isMultilink: if the link aggregates other links (changes the type)
  - *boolean* isArchived: if multilink, this attribute specifies, that this link was replaced during aggregation, but stays archived, as it could be later recovered when reversing aggregations
  - *boolean* forceTransferred: forces the visuals of a transferred link; aggregation algorithm sets this attribute in case of bundling, as we cannot calculate this attribute afterwards only from the data

```
{
    id: 4,
    type: 'regular',
    label: 'Riddle, Tom',
    name: 'Tom Marvolo',
    surname: 'Riddle',
    dataClass: 'person',
    vis: {
        replacedBy: 27,
    },
},
{
    id: 27,
    type: 'alias',
    label: 'Voldemort',
    vis: {
        graph: {
            x: 969.872726512079,
            y: -29.195455516806426,
        },
    },
}
```

Figure 21: JSON example of a regular and a group node. The first object is a regular node, which was aliased during aggregation and replaced by the second object.

```
{
    id: 16,
    source: 4,
    target: 1,
    direction: true,
    description: 'attempt to kill',
    vis: {
        newSource: 27,
        newTarget: 1,
        inMultilink: 72,
    },
},
{
    id: 72,
    source: 27,
    target: 1,
    vis: {
        isMultilink: true,
    },
    direction: {
        nonDirectional: false,
        sourceToTarget: true,
        targetToSource: false,
    },
}
```

Figure 20: JSON example of the two types of links. The first object is a singlelink, aggregated in the multilink, the second object.

4.4 Aggregations

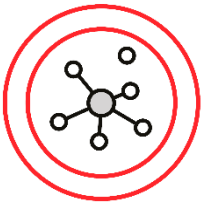
In Section 4.3.2.3, we briefly mentioned the functional requirement for node aggregations – a function vital to the exploratory analysis of data, which Zákopčanová [7] describes in Section 4.2.2.5 of her thesis. One of the goals of this thesis was to design an algorithm to support these aggregations and implement it in the network visualization module. The following sections, therefore, explain the concepts behind them and present an algorithm that enables them.

4.4.1 Definition of the functionality

Zákopčanová [7] defined three possible types of node aggregations in the visualization, and this section explains each of them:

1. *general group*
2. *alias*
3. *bundle*

4.4.1.1 General group



A general group (Figure 23) is formed by joining any nodes; for example, it can be used to group all the evidence of a specific criminal act. After creating the group, links connected to the joined nodes are *transferred* to the group node and visually differ from regular links. Grouping is illustrated on Figure 22.

Figure 23: Visual representation of a general group. Zákopčanová [7], Figure 4.22.

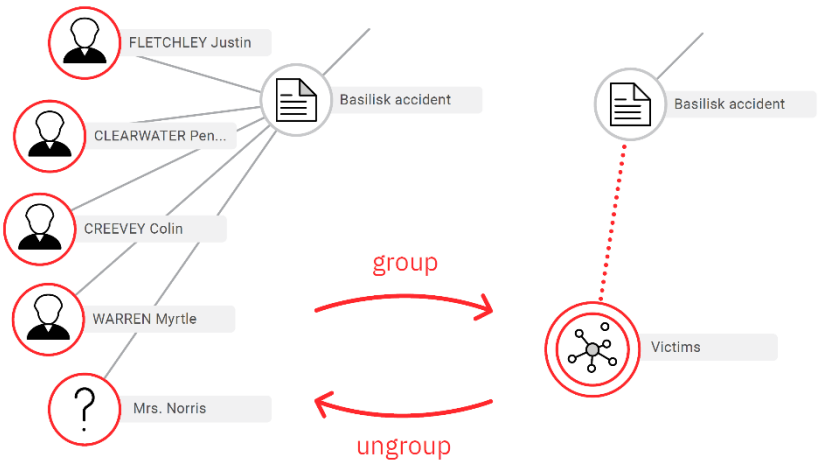


Figure 22: Node grouping. Zákopčanová [7], Figure 4.23.

4.4.1.2 Alias

Alias (Figure 24) is very similar to the general group and is used to join nodes that the analyst believes are representing a single entity. Implementation-wise, the alias is the same as the general group without transferred links. Links connected to joined nodes do not differ visually. Creating an alias is illustrated on Figure 25.

Figure 24: Visual representation of an alias. Zákopčanová [1], Figure 4.18.

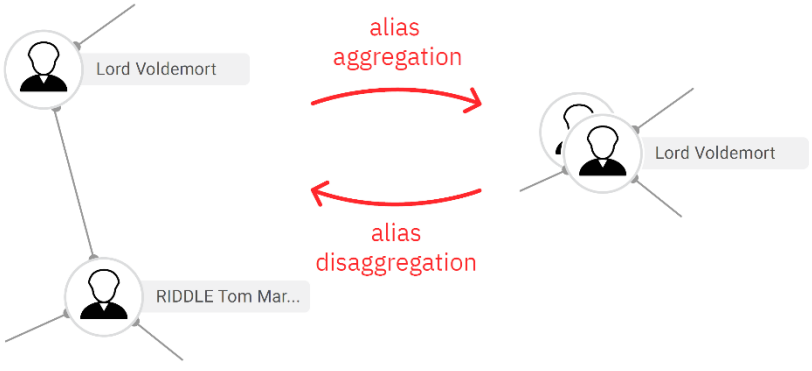


Figure 25: Node aliasing. Zákopčanová [1], Figure 4.19.

4.4.1.3 Bundle

Figure 26: Visual representation of a node with a bundle. Zákopčanová [1], Figure 4.20.

The last type of node aggregation is bundling, which allows the analyst to hide currently unimportant nodes by moving them to a node, which has a hierarchical or a logical connection to them. For example, in Figure 27, we do not want to see, which vault has *Hagrid* visited, when only his visit to the bank is important for us: we, therefore, bundle the *Vault 713* to the *Gringotts Bank*.

Implementation-wise, bundling is different from both the grouping and aliasing, as it does not create a new node. The links of the node with the bundle appear transferred if they now contain a link from the node we hid.

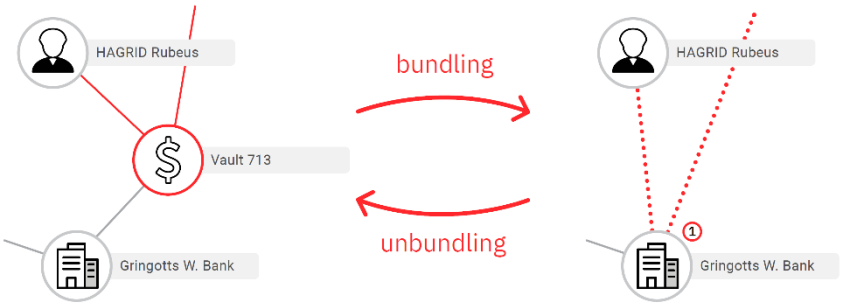


Figure 27: Node bundling. Zákopčanová [1], Figure 4.21

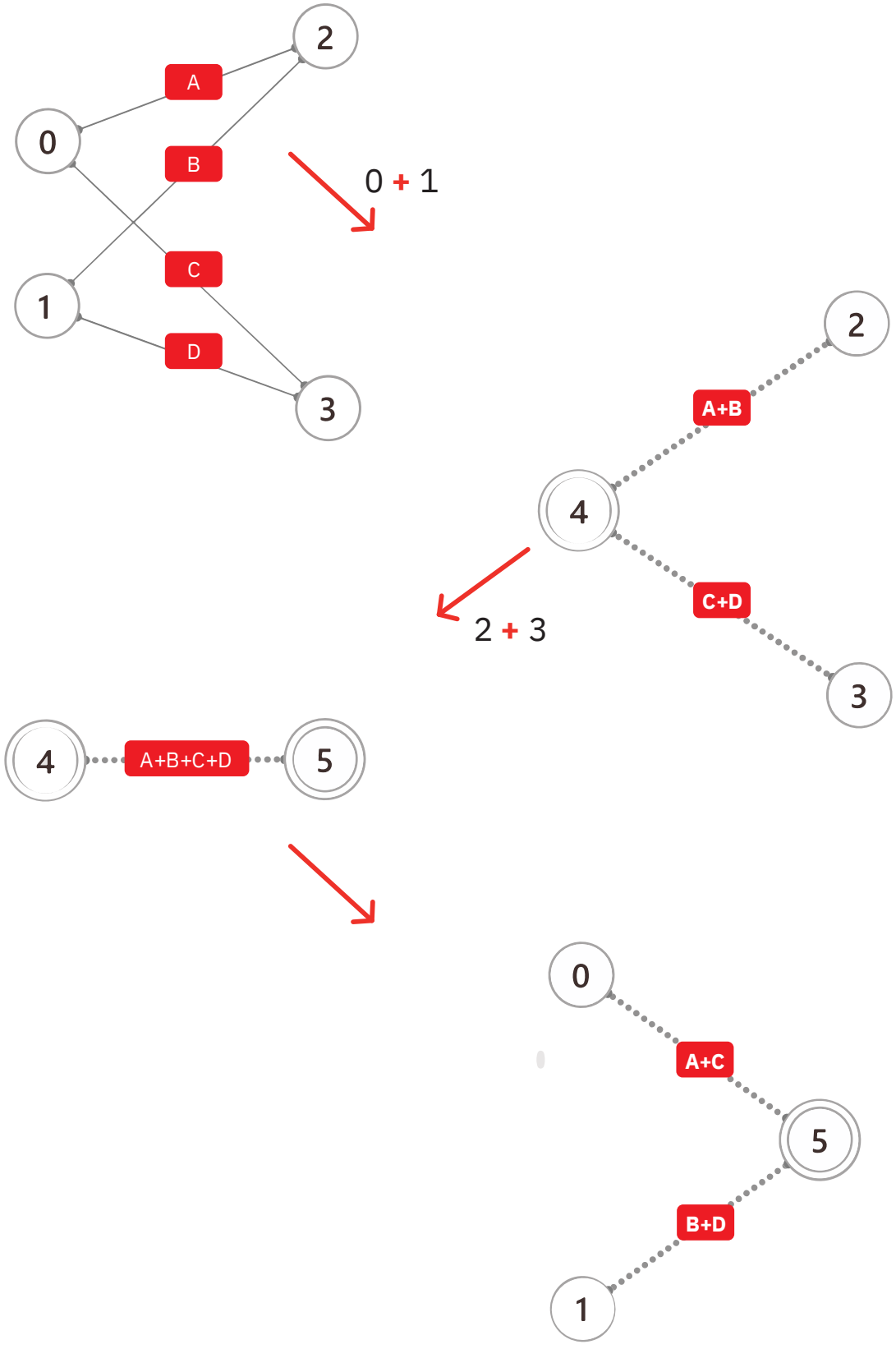
4.4.2 Algorithm design

This section explains the details needed to understand the design of the aggregation and disaggregation algorithms and how they work in the implementation.

4.4.2.1 Key points

- The aggregations need to be **reversible** operations. However, a naïve approach has shown that reversing the nodes in a different order they were created in poses a problem as illustrated on Figure 28, and the solution needs to be well-thought to avoid this.
- The three types of aggregations should be generalized to **avoid code repeat**.
- The data structure on which the algorithm operates is flat: meaning that aggregated objects **do not form a recursive hierarchy** in the data; instead, they have attributes that mark their direct successor or predecessor.
- The algorithm only needs to **work on a subgraph**, which is affected by the aggregation to be more optimal.

Figure 28: The order of operations matters. Disaggregating groups in different order can create links which did not exist in the graph beforehand, and the algorithm needs to account for that. To illustrate this, in the first step, we will group nodes 0 and 1. This will create links A+B and C+D, which will become aggregated into AB+CD after we group nodes 2 and 3. When we will then try to first disaggregate group 4, links A+C and B+D need to form, which did not exist.



4.4.2.2 The concept

The algorithm operates on two data structures, which store nodes and links. This section assumes the reader has a knowledge of the structure of dataset, described in Section 4.3.6, as it closely relates to the algorithm. The following description is slightly generalized to understand the concept more clearly.

Firstly, **nodes** of the dataset are represented by a hash map[31], which keys are the nodes' IDs, and its values are either empty if the nodes are visible and not placed in an aggregation, or contain a node ID of the successor node, which replaced them during aggregation. To find currently visible nodes, we list entries whose values are not set. If the entry has a value, we need to do recursive lookups to find the node which replaced it and is currently visible. For these lookups, we will benefit from the average-case constant search in hash maps.

Secondly, **links** are represented by a set. The keys are links' IDs, and their values are link's connections – IDs of end nodes, source and target. After we aggregate a node, the connections between the new node and the rest of the graph are internally represented by the same links, which are *transferred* to the new group nodes (by using visualization attributes *newSource*, *newTarget*, replacing the current *source* and *target*). Only multilinks are created, if necessary. Then during disaggregation, we calculate the link's new connection by using the nodes hash map. We take the original connection (attributes *source*, *target*), and we are recursively looking in the hash map to find the currently visible nodes to which the link is connected to. This method solves the earlier stated problem of reversing the aggregations in a different order.

A further complication in the design of the algorithm is caused by **multilinks**: links representing an aggregation of multiple links. Multilinks are formed when several links overlap in our network visualization – the multilink then hides the individual links. The algorithm needs to handle a lot more cases, as it needs to handle these multilinks.

4.4.3 Implementation details

The detail worth mentioning is the output of the algorithms in the context of the implementation. The output is a set of rendering instructions for the *NetworkGraph* component, such as which nodes should be added or removed. These instructions are then stored in an object, a ***payload***, which Vuex actions pass to mutations as input. As the NetworkGraph component is subscribed to the mutations, it consequently allows it to access the mutation's payload and retrieve the instructions. This is not a standard or documented Vuex approach, and in order to be correct, the payload should not contain any references to data

[31] The implementation (found in *data.store.js*) does not store nodes in a hash map, as they cannot support reactivity in the current version of JavaScript/Vue.js. However, during disaggregation, a hash map is created to optimize the recursive lookups of the *findReplacementNode* function.

stored in the store module. The implementation defensively uses only primitive types or deep copies of objects.

This approach was proposed as it was not possible to share data between Vuex store and d3-force, the D3 module, which calculates the layout of the graph in the NetworkGraph component. However, the benefit is that it allows the NetworkGraph to be replaced modularly, as it only receives the general instructions, not the data.

Furthermore, the algorithms were implemented using a test-driven approach. The tests for the *store/data* module can be found in the file *tests/unit/data.spec.js*.

4.4.4 Pseudocode of the aggregation algorithm

The implementation of this algorithm can be found in the store module *data.store.js* as the mutation *AGGREGATE_NODES*.

4.4.4.1 Input

- requested aggregation action: *group*, *alias*, or *bundle*
- a set of nodes to aggregate
- the target node

4.4.4.2 Output

- set of links, which should be removed from the visualization
- set of links, which should be newly drawn in the visualization

4.4.4.3 Pseudocode

1. Decide what is the target node and…
   1.1. if bundling, set *bundledNodes* on the target;
   1.2. if not bundling, create the new group/alias node in state;
   1.3. set *bundledIn/replacedBy* respectively on the nodes to be joined.
2. For every link which is being affected by the aggregation, i.e., connected to or in the subgraph:
   2.1. If the link is visible, set it as removed from the visualization.
   2.2. Calculate the link's new connections:
      2.2.1. skip the link, if it is inside the aggregation;
      2.2.2. calculate the *newSource, newTarget* as it is being transferred,
      2.2.3. if it is a multilink, archive it, and skip it.
   2.3. Update the link's connections.
      2.3.1. Set *forceTransffered* attribute if bundling.
   2.4. Release the link from multilink (if it belongs to one).
   2.5. If the source node is the same as the target, skip the link.
   2.6. Handle multilinks – the new connection may have created overlapping links.
      2.6.1. If the link overlaps and there exists a similar multilink, assign it to the multilink, and update the multilink's direction.

        2.6.2. Create a new multilink if the first option fail.
3. Update the links in the state and create the mutlilinks.
4. Prepare visualization payload: newly visible links, removed links.

4.4.5 Pseudocode of the disaggregation algorithm

Reverses node aggregations. The implementation of this algorithm can be found in the store module *data.store.js* as the mutation *DISAGGREGATE_NODE*.

4.4.5.1 Input

- the type of aggregation to revert: *group* (general group or alias) or *bundle*
- the target node: either the group/alias node to be disbanded, or a node of any type with a bundle to be unbundled

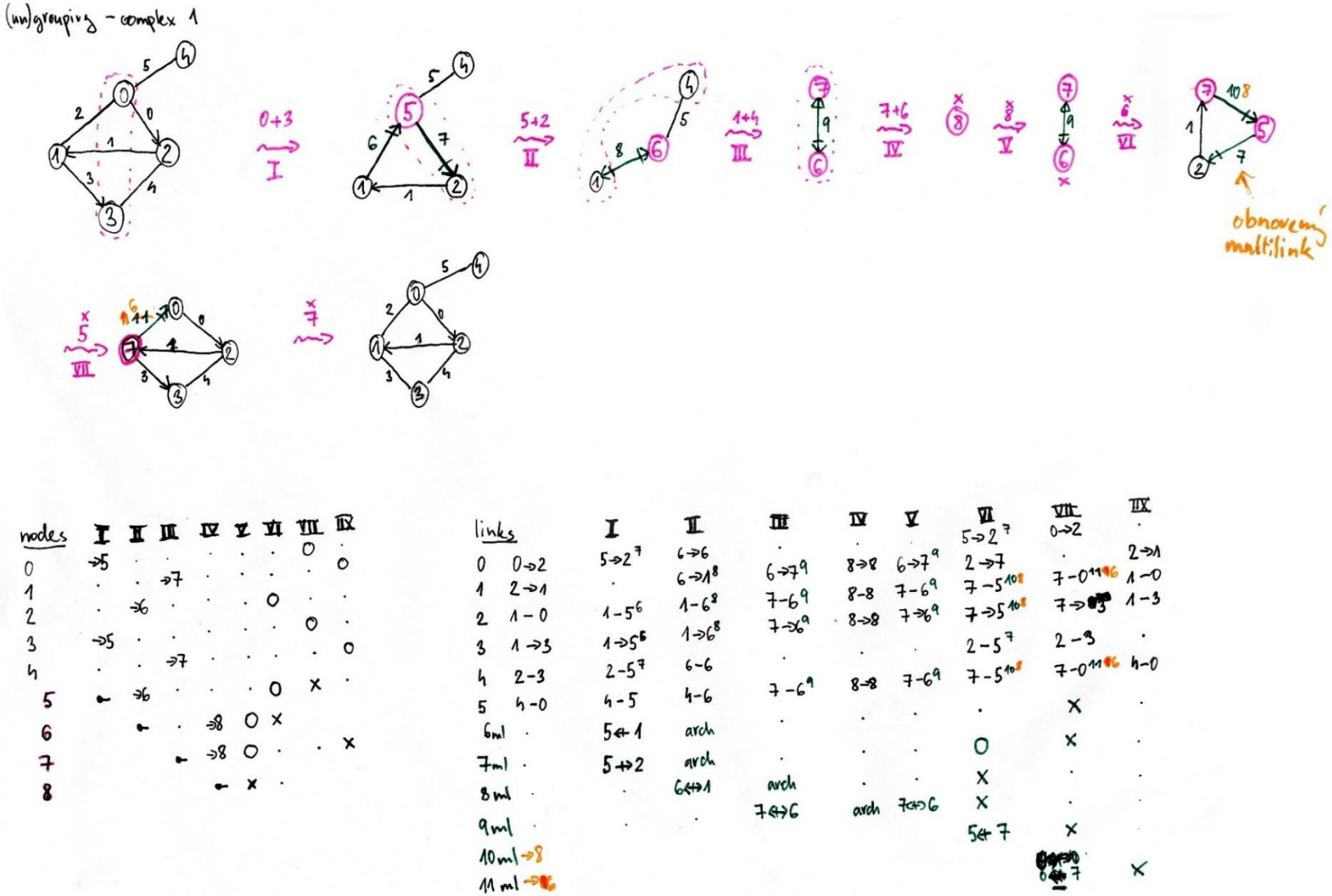
4.4.5.2 Output

- instruction for the renderer, whether it should remove the node from visualization (*true*, if disaggregating a bundle)
- set of links, which should be removed from the visualization
- set of links, which should be newly drawn
- set of nodes, which should be newly drawn

4.4.5.3 Pseudocode

1. Reveal original nodes.
2. If not reversing bundle, remove the target node.
3. Divide links into two sets:
   3.1. links forming the subgraph,
   3.2. other links, unaffected by the aggregation.
4. Prepare a set of archived multilinks, which could be restored.
5. For all subgraph links from step 3.1:
   5.1. Remove the link from visualization.
   5.2. Calculate the updated connections: for both of its end nodes, find the currently visible node which replaces them in the hash map of nodes.
   5.3. Update link's attributes: release the link from multilink, remove *forceTransferred*, and if the connection is now the same as in the beginning, remove *newSource* and *newTarget.*
   5.4. Handle multilinks – the updated connection may have created overlapping links:
      5.4.1. If the link overlaps...
         5.4.1.1. with multilink, assign it to the multilink, and update the multilink's direction;
         5.4.1.2. if there is no multilink, check, if we can recover it from the set prepared in step 4.
         5.4.1.3. Create a new multilink, if the first two options fail.
6. Update the links in the state by replacing them with merged sets from step 3.
7. Prepare visualization payload: newly visible links, removed links, newly visible nodes, and whether to remove the target node.

Figure 29: Visualizing the algorithm and illustrating its various corner cases was instrumental in the process of designing it. In fact, almost every test case was written using a hand-drawn illustration. This illustration depicts an ungrouping algorithm, how it handles multilinks and different order of aggregation, in 7 steps. The state of the data in each of the steps is then visible in the individual columns of the tables beneath the graph visualization. The table to the left is a hash map described earlier, which states whether the node has been replaced and by which node. To the right is a table with links, and the columns represent their current connection, and the green superscript number represents the multilink it belongs to.



5 Conclusion

The aim of this thesis was firstly to design a system architecture for building interactive data visualizations on the web using the frontend framework Vue.js and secondly to create a prototype of a network visualization evaluating the proposed architecture.

In the beginning, we have introduced data visualizations, described the different tools for creating them, and discussed visualizations on the web. We have then presented that most of the custom web-based visualizations are built using library D3.js. While discussing the possibly outdated aspects of the library that may hinder the development of complex visualizations, we have described recent concepts from the field of web development. In the third chapter, we have argued that we can use frontend frameworks for building visualizations and discussed the issues of using these frameworks simultaneously with the D3. Lastly, we have presented a case study of a network visualization prototype for project Analysis, which used the framework Vue.js as the core of the visualization. The study described the research project Analysis related to criminal investigations and the project's visualization component. We have listed the requirements on interactive functionality of the visualization and presented a system architecture of the application that will support them. This work also focused on the design and implementation of algorithms for the aggregation functionality of the visualization.

In the future work, we will focus on further development of the network visualization and evaluation of its features with the domain experts. Next, we will focus on developing additional modules for the visual analysis tool of project Analysis, including time-oriented and geospatial visualizations.

A: source code

The archive attached to the digital copy of this thesis contains source code of the prototype.

To run the code

Prerequisites

i. Install the JavaScript runtime Node.js, available: https://nodejs.org/en/.
ii. Install Python, which is required to compile node-sass, one of the dependencies.

Commands

Node.js includes NPM, a package manager used to handle dependencies in this project. To install the dependencies, run:

```
npm install
```

Install tooling for Vue, which will take care of building the application:

```
npm install -g @vue/cli
```

To compile the project and serve it on localhost, run:


```
npm run serve
```

Tests

Tests are located in */tests/unit/*. Run all tests with:

```
npm run test
```

To additionally generate a coverage report in */coverage*, run:

```
npm run test:coverage
```

6 Bibliography

[1] E. Meeks, “2019 Annual Data Visualization Survey Results,” Medium, 07-Aug-2019. [Online]. Available: https://medium.com/nightingale/2019-annual-data-visualization-survey-results-334d3523073f. [Accessed: 14-Dec-2019].

[2] S. Few, Now You See It: Simple Visualization Techniques for Quantitative Analysis. Oakland, Calif., USA: Analytics Press, 2009.

[3] S. Murray and O’reilly Media, Interactive data visualization for the web: an introduction to designing with D3, 2nd ed. O’Reilly Media, 2017.

[4] L. Meskanen-Kundu, “Making Data Accessible: An Overview of Interactive Data Visualization Using D3.js as Applied to a Scientific Dataset,” Bachelor Thesis, Helsinki Metropolia University of Applied Sciences, 2015.

[5] J. Steele and N. P. N. Iliinsky, Designing data visualizations. Farnham: O’Reilly Media, 2011.

[6] S. Stoppel, “User-Centric Parameter Specification for Interactive Virtual and Physical Visual Representations,” PhD Thesis, Universitetet i Bergen, 2018.

[7] K. Zákopčanová, “Visual Analysis of Data for Criminal Investigation,” Master’s Thesis, Masaryk University, Faculty of Informatics, 2018.

[8] S. Guldamlasioglu, “Web-based information visualization using JavaScript,” M.Sc. Thesis, University of Tampere, 2015.

[9] M. Bostock and J. Heer, “Protovis: A Graphical Toolkit for Visualization,” IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 6, pp. 1121–1128, Nov. 2009.

[10] Nathalie Henry Riche, Christophe Hurter, N. Diakopoulos, and Sheelagh Carpendale, Data-driven storytelling. CRC Press, 2018.

[11] M. Bostock, V. Ogievetsky, and J. Heer, “D3 Data-Driven Documents,” IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[12] P. Sweeney, “Why I no longer use D3.js,” Medium, 16-Dec-2018. [Online]. Available: https://medium.com/@PepsRyuu/why-i-no-longer-use-d3-js-b8288f306c9a. [Accessed: 14-Dec-2019].

[13] Mozilla, “Introduction to the DOM,” MDN Web Docs, 26-Aug-2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. [Accessed: 14-Dec-2019].

[14] E. Meeks, “D3 is not a Data Visualization Library,” Medium, 11-Jun-2018. [Online]. Available: https://medium.com/@Elijah_Meeks/d3-is-not-a-data-visualization-library-67ba549e8520. [Accessed: 14-Dec-2019].

[15] D. Scanlon, “How (and why) to use D3 with React,” Hackernoon.com, 30-Apr-2017. [Online]. Available: https://hackernoon.com/how-and-why-to-use-d3-with-react-d239eb1ea274. [Accessed: 14-Dec-2019].

[16] GitNation React Conferences, “D3 and React, Together - Shirley Wu,” YouTube. 24-Apr-2018.

[17] M. Bostock, “A Better Way to Code,” Medium, 28-Apr-2017. [Online]. Available: https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0. [Accessed: 14-Dec-2019].

[18] M. Iglesias, Pro D3.js: Use D3.js to Create Maintainable, Modular, and Testable Charts. Apress, 2019, pp. 22–23.

[19] E. Meeks, D3.js in Action: Second Edition. Shelter Island, N.Y., USA: Manning Publications Ltd, 2018.

[20] A. Smith, “Thinking in Components — Better from Front to Back,” Medium, 08-Sep-2017. [Online]. Available: https://blog.prototypr.io/thinking-in-components-better-from-front-to-back-7b43e4f2273f. [Accessed: 14-Dec-2019].

[21] P. Jang, “Modern HTML Explained For Dinosaurs,” Medium, 14-Nov-2018. [Online]. Available: https://medium.com/actualize-network/modern-html-explained-for-dinosaurs-65e56af2981. [Accessed: 14-Dec-2019].

[22] Vue.js, “Introduction — Vue.js,” Vuejs.org. [Online]. Available: https://vuejs.org/v2/guide/#Composing-with-Components. [Accessed: 14-Dec-2019].

[23] DotConferences, “dotJS 2016 - Evan You - Reactivity in Frontend JavaScript Frameworks,” YouTube. 21-Dec-2016.

[24] Kohei Mikami, “Understanding Rendering Process with Virtual DOM In Vue.js,” Medium, 13-Dec-2017. [Online]. Available: https://medium.com/@koheimikami/understanding-rendering-process-with-virtual-dom-in-vue-js-a6e602811782. [Accessed: 14-Dec-2019].

[25] R. Benitte, S. Greif, and M. Rambeau, “The State of JavaScript 2018,” Stateofjs.com, 2018. [Online]. Available: https://2018.stateofjs.com. [Accessed: 14-Dec-2019].

[26] Vue.js, “Comparison with Other Frameworks — Vue.js,” Vuejs.org. [Online]. Available: https://vuejs.org/v2/guide/comparison.html. [Accessed: 14-Dec-2019].

[27] E. Meeks, “Interactive Applications with React & D3,” Medium, 05-May-2017. [Online]. Available: https://medium.com/@Elijah_Meeks/interactive-applications-with-react-d3-f76f7b3ebc71. [Accessed: 14-Dec-2019].

[28] I. Mundy, “Declarative vs Imperative Programming,” Medium, 20-Feb-2017. [Online]. Available: https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2. [Accessed: 14-Dec-2019].

[29] S. Wu, “On D3, React, and a little bit of Flux,” Medium, 13-Jul-2015. [Online]. Available: https://medium.com/@sxywu/on-d3-react-and-a-little-bit-of-flux-88a226f328f3. [Accessed: 14-Dec-2019].

[30] J. Burget, “D4 — Declarative data-drive documents,” D4.js. [Online]. Available: https://d4.js.org/. [Accessed: 14-Dec-2019].

[31] E. Meeks, “Introducing Semiotic for Data Visualization,” Medium, Sep-2017. [Online]. Available: https://medium.com/@Elijah_Meeks/introducing-semiotic-for-data-visualization-88dc3c6b6926. [Accessed: 14-Dec-2019].

[32] Vue.js, “Single File Components — Vue.js,” Vuejs.org. [Online]. Available: https://vuejs.org/v2/guide/single-file-components.html#What-About-Separation-of-Concerns. [Accessed: 14-Dec-2019].

[33] M. Bostock, “D3.js - Data-Driven Documents,” D3js.org. [Online]. Available: https://d3js.org/. [Accessed: 14-Dec-2019].

[34] Masaryk University, “Complex Analysis and Visualization of Large-scale Heterogeneous Data,” Masaryk University, 2017. [Online]. Available: https://www.muni.cz/en/research/projects/35490. [Accessed: 14-Dec-2019].

[35] A. Gore, “WTF is Vuex? A Beginner’s Guide To Vue’s Application Data Store,” Medium, 30-Nov-2016. [Online]. Available: https://medium.com/js-dojo/vuex-for-the-clueless-the-missing-primer-on-vues-application-data-store-33fa51ffc3af. [Accessed: 14-Dec-2019].

[36] Mozilla, “Web APIs,” MDN Web Docs, 24-May-2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API. [Accessed: 14-Dec-2019].

[37] “Dynamic web page,” Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_web_page. [Accessed: 14-Dec-2019].

[38] S. K. Card, J. D. Mackinlay, and B. Shneiderman, Readings in information visualization: using vision to think. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 7.

[39] M. Iglesias, “Bringing Together React, D3, And Their Ecosystem,” Smashing Magazine, 21-Feb-2018. [Online]. Available: https://www.smashingmagazine.com/2018/02/react-d3-ecosystem/. [Accessed: 14-Dec-2019].

[40] M. Bostock, “Towards Reusable Charts,” 27-Feb-2012. [Online]. Available: https://bost.ocks.org/mike/chart/. [Accessed: 14-Dec-2019].

[41] Plotly, “More Basic Charts,” Plot.ly. [Online]. Available: https://plot.ly/javascript/basic-charts. [Accessed: 14-Dec-2019].

[42] R. P. L. Buse and W. R. Weimer, “Learning a Metric for Code Readability,” IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[43] Wikimedia Commons, Marketing Dashboard. 2015. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/1/17/Marketing_dashboard.png