6CCS3PRJ Final Year
ProvViz: an intuitive PROV editor and visualiser

Final Project Report

Author: Ben Werner

Supervisor: Professor Luc Moreau

Student ID: 1811051

April 9, 2021

Abstract

In an era where the ease of data collection, storage and consumption has evolved rapidly, data provenance - records describing the entities and processes involved in data production [4] - has become more important than ever. The World Wide Web Consortium (W3C) introduced the PROV model in 2013 to standardise the modelling of data provenance, defining an ontology of entities, activities, agents and the relationships between them. A key component of measuring a standard's success is to what extent it has been adopted. Therefore tools that lower the barrier to adopting a standard - by making it easier to understand, produce and interact with models - could play an important role in the success of the standard. This report introduces *ProvViz*, which aims to undertake this role by introducing an intuitive in-browser PROV editor and visualiser that doesn't require prior knowledge of PROV document syntax.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Ben Werner

April 9, 2021

Acknowledgements

I would like to give special thanks to my project supervisor Prof. Luc Moreau, for his guidance and support that made this project possible. I would also like to thank the Informatics Department at King's College London, and my partner for moral support throughout the project.

Contents

Chapter 1

Introduction

Data provenance is a valuable aspect of many data workflows; providing insight in the origins and processes that influenced its formation so that the quality and reliability of the data can be better assessed. The W3C standard PROV [12] models data provenance in terms of *Entities*, *Activities* and *Agents* and the relationships between them. A PROV document can therefore be visualised as a graph composed of nodes and edges.

This work introduces *ProvViz*, an in-browser application that enables intuitive editing and visualisation of PROV models. The tool is targeted towards both users that are well-versed in the PROV ontology, and those less familiar with the PROV ontology or PROV document syntax who are seeking to comprehend, modify or visualise existing documents to improve their familiarity with the ontology.

1.1 Project Motivations

As data collection and storage technologies have evolved exponentially in the information age [7], modelling the production processes of data items such as articles and datasets has become critical when it comes to assessing their reliability. The PROV standard could therefore play an important role in improving how such assessments are made.

This is why the primary motivation of the *ProvViz* tool is to make PROV documents accessible to more users - regardless of their familiarity with the standard - in the hopes of promoting and increasing the understanding and usage of data provenance.

1.2 Scope

The *ProvViz* project is divided into two components:

- the *ProvViz Application* responsible for loading and exporting PROV documents, and providing the primary interface for editing and visualising PROV documents; and
- the *ProvViz Visualiser* responsible for visualising the provenance and providing intuitive editing functionality.

As common amongst web-applications, this project will make use of existing open-source code modules taken primarily from the Node Package Manager[1] (NPM) to prevent the need for implementing features from scratch where existing solutions can provide equal if not better functionality. This not only reduces the scope of the project, but allows for better maintainability as popular packages are updated regularly to improve performance, adopt new standards or implement new features. For example, the open-source project *Monaco* [2] - a text editor maintained by Microsoft - will be used to provide direct PROV document editing functionality, rather than implementing a text editor from scratch.

1.3 Objectives

The accessibility of the *ProvViz Application* is a primary concern, so the application should be easy to access and its GUI easy to understand and use. The remaining primary objectives of the application can be summarised as follows:

- **Objective 1:** the application enables PROV document upload, download and storage functionality
- **Objective 2:** the application enables direct PROV document editing functionality in the form of a text editor
- **Objective 3:** the application enables intuitive PROV document editing functionality that does *not* require prior knowledge of PROV document syntax
- **Objective 4:** the application provides a a customisable visualisation of the provenance graph that continuously updates to represent the state of the PROV document, and can be exported as an image

[1] https://www.npmjs.com/
[2] https://microsoft.github.io/monaco-editor/

An additional objective of the project is to release the devised *ProvViz Visualiser* component as a standalone NPM package, to allow other developers to incorporate the tool in their own projects. This will require additional design considerations to be made to ensure the component is sufficiently decoupled from the *ProvViz Application* so that it can be used in a variety of application contexts.

1.4 Why target the Web Browser

One of the primary objectives of the project is to make the PROV visualisation and modification tool as accessible as possible, which is where modern web-browser applications excel. This is because a wide range of personal computers support modern web-browsers, including desktops, laptops, tablets and smartphones of all popular platforms. As well as being accessible on essentially all modern devices, web-applications don't require any downloads or installations (apart from the browser itself) to function, making them instantly accessible to users when the web-page is loaded. This lowers the barrier to using the application.

Modern browser APIs additionally provide features that historically have been available only to native applications, such as local storage[3] which enables data storage that persists across browser sessions, WebAssembly[4] which allows binary modules from languages such as C and Rust to be run in the browser with near native performance, or the Web Share API [5] which invokes the native share API of the current device.

1.5 Development Approach

The waterfall model was deemed as the most appropriate software development cycle. The model was chosen, as opposed to a more iterative and agile development cycle, because the requirements of the project were clearly established at the start and not expected to change significantly throughout the development phases.

The implementation phase of the project was partitioned by the two deliverables (the *ProvViz Visualiser* and the *ProvViz Application*). The *ProvViz Application* was implemented after the *Visualiser*, as it depends on the *Visualiser* component to fulfill its required functionality (discussed further in the requirements and design chapters).

[3] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[4] https://developer.mozilla.org/en-US/docs/WebAssembly
[5] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/share

1.6 Report Structure

The report will start by discussing the background of the project. The structure of the following sections closely resembles the development approach of the project. The requirements and specifications will be outlined, followed by a thorough summary of the design of the application and its architecture. The implementation and testing section will review the implementation of the *ProvViz Visualiser* and *ProvViz Application* and the implemented test-suite, followed by a section dedicated to evaluating the developed software system and the project as a whole in the context of its objectives. Finally, the conclusion will conclude the report with a discussion of future work. Terminology used commonly throughout the report is defined in a section dedicated to definitions (chapter 9).

Chapter 2

Background

2.1 Browser Applications

Over the past decade, web browsers have developed considerably through the introduction of HTML5 and browser APIs that have empowered web-apps with many of the features expected from native applications. Because modern web browsers are ubiquitous across popular personal computing devices, developing a single browser-based application has become an attractive alternative to creating several platform specific applications. Examples include the browser-based word processor *Google Docs*[1], the web-based coding IDE *Repl.it*[2] and the web-based simulation platform *HASH*[3].

2.1.1 TypeScript

TypeScript is a statically typed programming language that is developed and maintained by *Microsoft*, and is a superset of the JavaScript programming language. The typing functionality provided by the language makes it a popular choice for individuals and organisations that want to build robust applications that compile to JavaScript, making TypeScript a great alternative when developing modern web-apps of non-trivial complexity.

2.1.2 React

React is a modern open-source JavaScript library for building user interfaces and UI components. According to the 2020 Stack Overflow Developer Survey, it has become the second

[1] https://docs.google.com/
[2] https://repl.it
[3] https://hash.ai

most-used web framework[4] coming narrowly after the 14 year old framework jQuery. Further research indicates the jQuery web-framework is in the process of a decline in contrast to React, evidenced by its rapid decrease in the monthly Stack Overflow questions as shown in Figure 2.1.

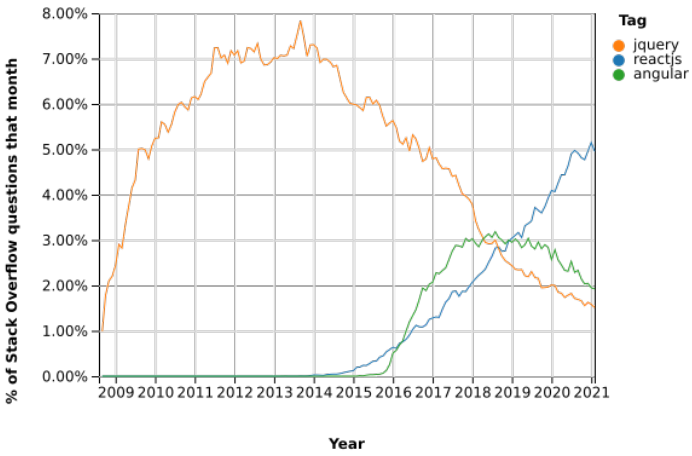


Figure 2.1: Stack Overflow Questions per month of jQuery, React and Angular

The primary features that make React a modern alternative to jQuery include its declarative approach to writing UI components that encapsulate their internal state and logic, and the ability to build complex applications composed of a component hierarchy. The framework is primarily used for developing browser-based applications, but can also be used to develop native applications on desktop[5] and mobile[6] devices. React additionally provides comprehensive TypeScript support, enabling the inputs of UI components to be statically type-checked at compile time.

2.2 The PROV Data Model

The PROV standard was published to the World Wide Web Consortium (W3C) in 2013 [12], and models data provenance so that the quality, reliability and trustworthiness of the data can be better assessed. The aspects of the PROV ontology relevant to this project will now be briefly outlined. The full definition of the ontology is made available by the W3C organisation [2].

The primary components of the model are *Entities*, *Activities* and *Agents*. *Entities* are

[4] https://insights.stackoverflow.com/survey/2020#technology-web-frameworks
[5] https://www.electronjs.org/
[6] https://reactnative.dev/

the physical, digital, conceptual or other kinds of things related to the provenance of an item. *Activities* are the processes of how entities come into existence and change to become new entities. And finally, *Agents* represent the actors responsible for an *Activity*, the existence of an *Entity* or another *Agent* [2].

There are several directed relationships between these components that are modelled in PROV, where the primary relationships include: *wasGeneratedBy*, *wasDerivedFrom*, *wasAttributedTo*, *used*, *wasInformedBy*, *wasAssociatedWith* and *actedOnBehalfOf* [2]. *Entities*, *Activities*, *Agents* and the relationships between them provide the basis for visualising provenance as a directed graph of nodes and edges.

The PROV ontology is also made of up additional components and relationships that will be used in the *ProvViz* application. A PROV *Bundle* is a set of *Agents*, *Entities*, *Activities* and relationships which may itself have provenance [2]. Some components in the PROV model can have additional defined attributes. For instance an *Activity* can have attributes such as *startedAtTime* and *endedAtTime* which represent timestamps indicating the start and end times of the activity. In addition to defined attributes, *Entities*, *Activities*, *Agents* and relationships can have custom attributes that act as a key-value store enabling the user to capture additional information.

2.3 Related Work

2.3.1 Provenance Visualisation

There are a variety of existing tools that facilitate provenance visualisation.

Provenance Explorer [8] is a standalone Java-based desktop application that visualises RDF provenance data, with an emphasis on security features that produce a variety of views based on a user's access privileges. The target user group is scientists who benefit from visualising the provenance of their scientific workflows. It uses the ABC ontology model, lacking support for PROV documents, and focuses exclusively on provenance events associated with laboratories or manufacturing.

Prov Viewer [14] is also a standalone Java-based desktop application, with the aim of enabling visual exploration of PROV documents. The application facilitates filtering nodes and edges, zooming and panning to navigate the graph, basic visualisation editing functionality including editing colours and shapes, and merging multiple provenance graphs into a single visualisation. The GUI of the application presumes prior knowledge of the PROV ontology,

and no modification functionality is implemented.

AVOCADO [18] is another visualisation tool which generates provenance visualisations based on data flows with the goal of assisting the reproducibility of bioinformatic workflows by allowing the user to introspect the provenance visually. It supports very large provenance graphs through clustering methods that expand only the parts of the graph that are relevant to the user.

More general-purpose tools such as the *ProvToolBox* Java library [15] and the *prov* Python library [13] also implement provenance visualisation generation functionality, producing an SVG image for a corresponding PROV document.

In general, the provenance visualisation tools that exist to date focus on scientists as their target users because of the important role provenance plays in the scientific discovery process as a means of reproducing and peer-reviewing experimental procedures. None of the applications are directed towards users who have no more than a passing interest in data provenance, and simply want to interact with existing PROV documents to learn more about the standard and how it can be used.

Some are limited to certain domains (*Provenance Explorer* and *AVOCADO*), and all require downloading and installing software packages on a desktop computer before they can be used. The applications that support the PROV ontology focus on provenance visualisation, and don't facilitate intuitive editing functionality relying on the user to manually modify the underlying PROV document if they wish to do so, hence requiring familiarity with PROV document syntax. This means **Objective 3** of the *ProvViz Application* is not addressed by any of the existing visualisation tools. In addition it can be argued that none of the existing applications focus on an easy-to-use interface for users unfamiliar with the PROV ontology. These are the primary aspects of the *ProvViz Application* that set it apart from the existing PROV visualisation tools available today.

Chapter 3

Requirements & Specifications

3.1 Requirements

The functional and non-functional requirements of the *ProvViz Application* and *ProvViz Visualiser* will now be examined.

3.1.1 *ProvViz Application* Requirements

***ProvViz Application* Functional Requirements**

- AF1 - Users can upload a PROV document in a common format (PROV-N, PROV-JSON, RDF, XML, TriG)
- AF2 - Users can download a modified PROV document in a common format (PROV-N, PROV-JSON, RDF, XML, TriG)
- AF3 - Users can export visualisations as an SVG image
- AF4 - Users can export visualisation in other common image formats (JPEG, PNG)
- AF5 - Users can create an "empty" PROV document
- AF6 - Users can open an example PROV document to modify or visualise
- AF7 - Users can open previously uploaded PROV documents without needing to re-upload them
- AF8 - Users can view and modify a PROV document using a text editor
- AF9 - Users can view and modify a PROV document using its visualisation

***ProvViz Application* Non-Functional Requirements**

- ANF1 - Users can start using the system without downloading or installing any software
- ANF2 - The application is touch accessible so that it can be used on tablets, smartphones and other touch devices
- ANF3 - The application is responsive, and can be used on devices with a variety of screen sizes

3.1.2 *ProvViz Visualiser* Requirements

***ProvViz Visualiser* Functional Requirements**

- VF1 - Users can view a PROV document as a visualisation
- VF2 - Users can zoom and pan to navigate/explore the visualisation
- VF3 - Users can filter the specific namespaces, *Agents*, *Activities*, *Entities* or relationships that are displayed in the visualisation without modifying the underlying PROV document
- VF4 - Users can switch between different provenance *views* (Responsibility, Data Flow, Process Flow) to filter the *Agents*, *Activities*, *Entities*, *Bundles* and relationships that are displayed in the visualisation without modifying the underlying PROV document
- VF5 - Users can access documentation of *Agents*, *Entities*, *Activities*, *Bundles* and PROV relationship types
- VF6 - Users can intuitively create and delete a namespace declaration
- VF7 - Users can intuitively modify the prefix and the value of a namespace declaration
- VF8 - Users can intuitively create and remove an *Agent*, *Activity*, *Entity* or *Bundle*
- VF9 - Users can intuitively modify the identifier of an *Agent*, *Activity*, *Entity* or *Bundle*
- VF10 - Users can intuitively create and remove relationships between *Agents*, *Activities* and *Entities*
- VF11 - Users can intuitively add or remove an *Agent*, *Activity* or *Entity* from a *Bundle*
- VF12 - Users can intuitively set PROV attributes or other custom attributes for *Agents*, *Activities*, *Entities* and relationships

- VF13 - Users can intuitively edit the default colour of an *Agent*, *Activity* or *Entity*
- VF14 - Users can intuitively edit the colour of a single *Agent*, *Activity* or *Entity*
- VF15 - Users can intuitively edit the shape of a single *Agent*, *Activity* or *Entity*
- VF16 - Users can intuitively edit the position of a single *Agent*, *Activity* or *Entity*

***ProvViz Visualiser* Non-Functional Requirements**

- VNF1 - The system should react to user input *instantaneously* (user input is followed by feedback within 100 milliseconds [10])

3.2 Specifications

To discuss the specification of the *ProvViz Application* and *Visualiser*, the MoSCoW prioritisation technique will be used to evaluate the priority of each requirement. This means partitioning the requirements into *must have*, *should have*, *could have* and *will not have* groupings. The devised application cannot be considered as completed unless all the *must have* requirements have been implemented.

3.2.1 *ProvViz Application* specifications

Requirement	Specification	Priority
AF1	The *Application* must support uploading a PROV document in a common PROV format, translating it if necessary to the serialisation format of PROV-JSON using the *ProvToolBox API*	MUST
AF2	The *Application* must support downloading a PROV document in a common PROV format, translating the serialisation of the document to the format if necessary using the *ProvToolBox API*	MUST
AF3	The *Visualiser* must support translating the PROV document into an SVG image format, so that it can then be directly downloaded	MUST
AF4	The *Visualiser* could support translating the SVG image into other image formats natively using a HTML Canvas attribute to draw the SVG and then export it in a desired image format	COULD
AF5	The *Application* should let users create an empty PROV document with no *Agents*, *Entities*, *Activities*, *Bundles* or relationships so that the application can be used to create a PROV document from scratch	SHOULD
AF6	The *Application* should store example PROV documents so that the tool can be used without needing your own PROV document	SHOULD
AF7	The *Application* should store uploaded, created or modified example PROV documents in the browser's local storage so that they can be re-opened if the browser tab or the browser is closed	SHOULD
AF8	The *Application* should use the *Monaco* text editor[a] to display the current state of the PROV document and allow the user to modify it directly	SHOULD
AF9	The *Application* must use the *ProvViz Visualiser* component to visualise the PROV document and allow for intuitive editing functionality	MUST
ANF1	The *Application* must be browser-based and accessible at `https://provviz.com` so that it can be directly used without any prior downloads or installations	MUST
ANF2	The *Application* could implement the relevant touch-events required to be considered fully touch accessible	COULD
ANF3	The *Application* could implement a responsive design that adjusts according to screen width and height	COULD

[a] https://microsoft.github.io/monaco-editor/

3.2.2 *ProvViz Visualiser* specifications

Requirement	Specification	Priority
VF1	The *Visualiser* will use the *GraphViz* framework[a] to generate a visualisation of the provenance graph in an SVG format, which can be displayed in the browser's Document Object Model (DOM)	MUST
VF2	The *Visualiser* will use the D3 JavaScript library[b] to manipulate the *GraphViz* SVG visualisation and provide zooming and panning functionality using D3-Zoom[c]	MUST
VF3	The *Visualiser* must provide the relevant checkbox inputs and buttons to allow users to filter namespaces, *Agents*, *Activities*, *Entities* or relationships that are displayed in the *GraphViz* SVG visualisation	MUST
VF4	The *Visualiser* should provide a select input that enables users to switch between different provenance views, filtering the relevant *Agents*, *Activities*, *Entities*, *Bundles* and relationships	SHOULD
VF5	The *Visualiser* should display documentation for *Agents*, *Entities*,*Activities*, *Bundles* and each PROV relationship type to the user, citing the W3C PROV Namespace Documentation[d]	SHOULD
VF6	The *Visualiser* must allow users to create and remove namespaces by the press of a button	MUST
VF7	The *Visualiser* must allow users to modify the prefix and value of a namespace using text input fields	MUST
VF8	The *Visualiser* must allow users to intuitively create or delete *Agents*, *Activities*, *Entities* or *Bundles* by the click of a button	MUST
VF9	The *Visualiser* must allow users to intuitively modify the namespace of an *Agent*, *Activity*, *Entity* or *Bundle* using a select input field, and modify their names using a text input field	MUST
VF10	The *Visualiser* must allow users to intuitively create and remove relationships between *Agents*, *Activities* and *Entities* by using an auto-complete text input field	MUST
VF11	The *Visualiser* must allow users to intuitively add or remove *Agents*, *Activities* or *Entities* from a *Bundle* using a rearrangeable "Tree View" of the provenance graph, and making use of the existing `react-sortable-tree` NPM package[e]	MUST
VF12	The *Visualiser* must allow users to intuitively set a common PROV attribute or a custom attribute for *Agents*, *Activities* and *Entities* using the relevant text input fields	MUST
VF13	The *Visualiser* should allow users to intuitively edit the default colour of an *Agent*, *Activity* or *Entity* by using a color picker storing the value in a global settings data structure	SHOULD

[a] https://graphviz.org/
[b] https://d3js.org/
[c] https://github.com/d3/d3-zoom
[d] https://www.w3.org/ns/prov
[e] https://www.npmjs.com/package/react-sortable-tree

VF14	The *Visualiser* should allow users to intuitively edit the colour of a specific *Agent*, *Activity* or *Entity* by using a color-picker and storing the value as a PROV attribute with the key `provviz:color`	SHOULD
VF15	The *Visualiser* should allow users to intuitively edit the shape of a specific *Agent*, *Activity* or *Entity* by using a select input field and storing the value as a PROV attribute with the key `provviz:shape`	SHOULD
VF16	The *Visualiser* could allow users to intuitively edit the position of a specific *Agent*, *Activity* or *Entity* by using a select input field and storing the value as a PROV attribute with the key `provviz:position`	COULD
VNF1	The *Visualiser* should enable all user input to be followed by feedback within 100 milliseconds by utilising established techniques, such as "debouncing" CPU intensive or asynchronous tasks that are triggered by text inputs or other events that can be rapidly triggered sequentially [9]	SHOULD

Chapter 4

Design

In this chapter the design of the *ProvViz Application*, the *ProvViz Visualiser* and the architecture as a whole will be discussed. The work done in the design phase will strongly support the implementation phase of the project.

4.1 Use Cases

UML use case diagrams help summarise how actors interact with a devised software system. The use case diagram depicted in Figure 4.1 summarises how an actor interacts with the primary functionality of the *ProvViz* system. The sole actor in the system is the user of the application.

As depicted in the use case diagram, the user can load a PROV document using four methods:

- the user may load an empty PROV document; or
- the user may upload a PROV document from their device; or
- the user may choose a PROV document from a list of example documents; or
- the user may choose a PROV document from a list of recent documents (documents the user has previously loaded)

Once a document is loaded, it will be translated into the *ProvViz Application*'s serialisation format. The serialisation format used by the *ProvViz Application* and *ProvViz Visualiser* is the PROV-JSON format, as JSON objects can be natively modified in JavaScript and are therefore more straightforward to manipulate programmatically than the other PROV formats available.

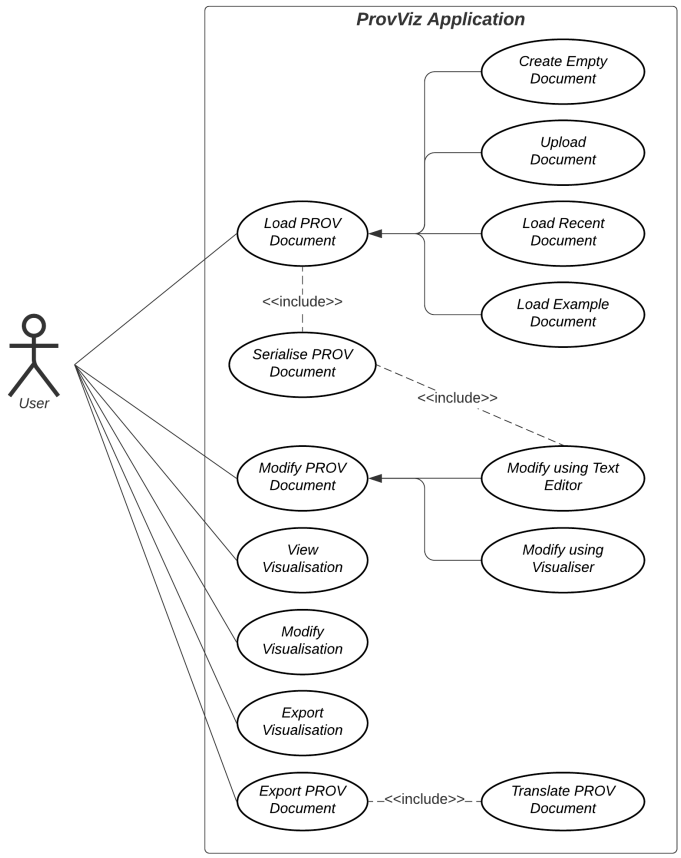


Figure 4.1: UML Use Case Diagram of the ProvViz Application

The user may seek to modify the PROV Document, in which case they can either modify the PROV document directly using a text editor, or intuitively make modifications using the *ProvViz Visualiser*. When modifying the PROV document directly, it needs to be translated into the serialisation format.

The user can view a visualisation of a PROV document, using the GUI of the *ProvViz Visualiser*. This visualisation can also be modified by interacting with the *Visualiser*'s visualisation editing functionality (modifying colors, shapes, filtering items, etc). The user may then seek to export the devised visualisation in a common image format.

Finally, the user may seek to export the PROV document in a desired PROV format, translating the serialised document in the process.

4.2 System Architecture

Because both the *ProvViz Application* and the *ProvViz Visualiser* can be considered as standalone software systems (a web application and a React component NPM package), their architectures will be discussed individually.

4.2.1 ProvViz Application

The system architecture diagram depicted in Figure 4.2 depicts the architecture of the *ProvViz Application* as a high-level abstraction consisting of its primary components and their dependencies. A dependency between two components is depicted by a directed edge, where the source node depends on the destination node.

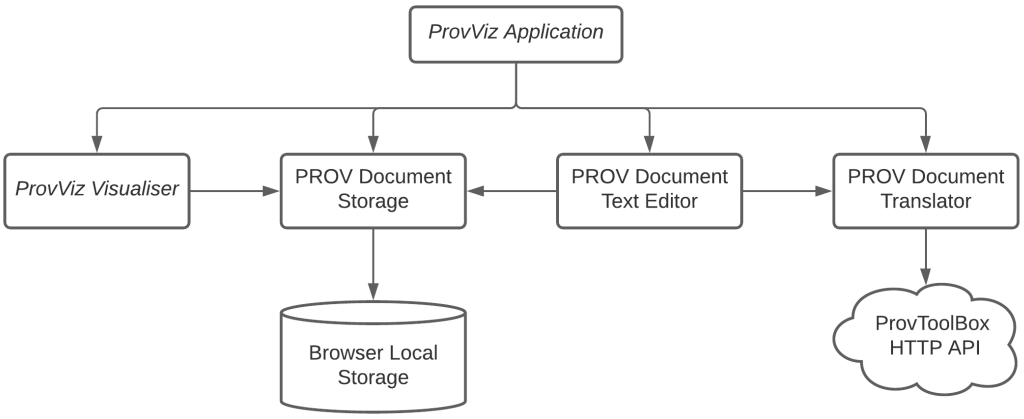


Figure 4.2: System Architecture Diagram of the ProvViz Application

PROV Document Translator

The PROV Document Translator component is responsible for transforming a PROV document from one format to another. This component is used when uploading a PROV document or editing a PROV document using a text editor, as the document in question must be translated into the serialisation format PROV-JSON. It is also used when exporting a PROV document into a desired format.

The PROV Document Translator makes use of the *ProvToolBox* HTTP API [15], which is publicly hosted at the endpoint `https://openprovenance.org/services/provapi` by the Open Provenance organisation. The HTTP API implements a variety of PROV manipulation and validation tools, but the PROV Document Translator will only require the HTTP route

hosted at endpoint `/documents2`. As detailed in its documentation[1], this route accepts a HTTP POST request containing the PROV document in its HTTP body, where the desired return PROV document format is specified in its HTTP header. The route then responds with the HTTP response status code 301 and a redirect URL to a temporary hosted version of the translated document.

The use of an HTTP API dictates that the PROV Document Translator operates asynchronously, an important design consideration.

At the time of writing, a PROV translation library either implemented in JavaScript or compiled to WebAssembly could not be obtained. Therefore a solution for translating PROV documents synchronously in the browser was not devised for this project.

PROV Document Storage

The PROV Document Storage component is used to retrieve and store PROV documents using the local storage API of the browser. This will allow PROV documents to be stored across browser sessions, so documents are not lost when the the browser tab is closed, the application is quit, or the user's device is restarted.

PROV Document Text Editor

The PROV Document Text Editor component is used to edit the PROV document directly. Once modified, the updated PROV document must be translated into the serialisation format and stored which is why the component depends on the PROV Document Translator and Storage components. Therefore when the Text Editor modifies the document, saving the changes cannot occur synchronously (as serialising the document depends on the *ProvToolBox* HTTP API and therefore occurs asynchronously). This will require further design considerations when devising the Graphical User Interface of the *ProvViz Applications*, as there should be visual feedback whilst the asynchronous save request is occurring, and if the request fails an appropriate error message should be displayed. This will be further outlined in the Graphical User Interface section.

4.2.2 ProvViz Visualiser

The final dependency of the *ProvViz Application* depicted in Figure 4.2 is the *ProvViz Visualiser*, the component responsible for visualising PROV documents and providing intuitive

[1] https://openprovenance.org/services/view/api#/provapi/submit2

editing functionality that enables the user to modify the PROV document and the generated visualisation.

Name	Type	Necessity	Description
document	*object*	Required	The PROV document to be visualised, in the PROV-JSON format
`onChange`	*object→void* or *null*	Required	When a function is provided, the PROV document state is controlled by its parent component and the supplied function is called when the PROV document is modified in the *Visualiser* component. When *null* is provided instead of a function, the PROV document state is controlled by the *Visualiser* component itself. The state management of the *Visualiser* component will be further discussed in section 4.2.2 dedicated to discussing the PROV document state management.
`wasmFolderURL`	*string*	Required	The endpoint where the required GraphViz WASM module can be accessed
`width`	*number*	Required	The width in pixels of the *Visualiser* component when rendered in the browser DOM
`height`	*number*	Required	The height in pixels of the *Visualiser* component when rendered in the browser DOM
`initialSettings`	*object*	Optional	When supplied, these are the visualisation settings used to initialise the *Visualiser* component
`onSettingsChange`	*object→void*	Optional	When supplied, the function is called when the visualisation settings are modified in the *Visualiser* component
`documentName`	*string*	Optional	The name of the current PROV document, used as a filename when the visualisation is exported as an image

Table 4.1: Props of the *ProvViz Visualiser* React component

A key design decision of the *ProvViz Visualiser* is that it is a standalone React UI component, that will be distributed as an NPM package so that it can be used in a variety of applications. When designing a React UI component intended for distribution, clearly defining its "props" (the required and optional inputs of the component) is an important aspect to improve usability for third-party developers. Table 4.1 outlines the props of the *ProvViz Visualiser* React component, which are also made publicly available in the `README.md` file of

the *ProvViz Visualiser*'s GitHub repository[2] and the homepage of its NPM package[3].

PROV Document State Management

From the perspective of the *Visualiser* component, the current state of the PROV document can either be managed by the component itself or by its parent as illustrated in Figure 4.3.

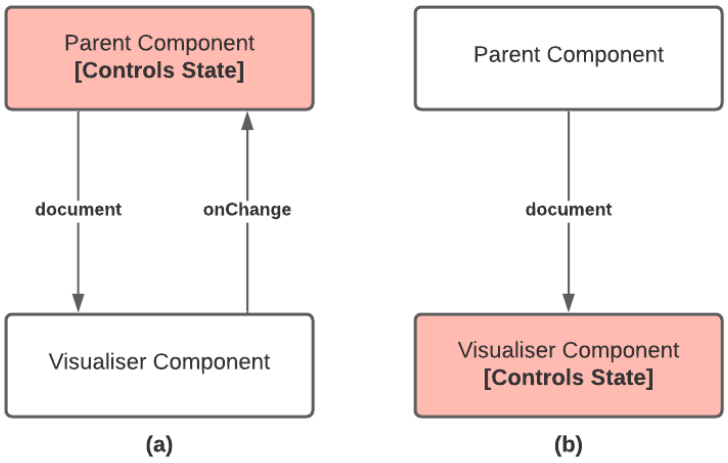


Figure 4.3: State Management of the *Visualiser* Component

When the state is controlled by the parent component (Figure 4.3.a), the current state is reflected by the `document` React prop. The `onChange` prop of the *Visualiser* component must be a function, which is called whenever the PROV document is modified in the *Visualiser* component. Changes to the PROV document do not have any effect until the parent component updates the `document` prop.

When the state is controlled by the *Visualiser* component (Figure 4.3.b), the `document` prop of the component is used to initialise the state of the *Visualiser* component, but does not necessarily reflect the current state of the PROV document (this is stored and managed internally by the *Visualiser* component). The `onChange` prop must be set to `null`, as the state is not managed by the parent component.

The *Visualiser* component can therefore determine who is responsible for managing the state of the PROV document by checking whether the `onChange` prop is a `function`, or `null`. In the context of the *ProvViz Application*, the PROV document state will be managed by the parent so that the current state can be reflected by both the text editor and the *Visualiser*.

[2] https://github.com/benwerner01/provviz
[3] https://www.npmjs.com/package/provviz

Visualisation Settings

The visualisation settings of the *ProvViz Visualiser* are the settings that define how the provenance visualisation is presented to user. Serialisation of the visualisation settings refers to the process in which user-defined visualisation settings are saved alongside or embedded within the PROV document, so that they can be restored in a later life-cycle of the application.

Some of the visualisation settings can be embedded within the PROV document as custom PROV attributes of *Agents*, *Entities* or *Activities*, which include:

- the color of a specific node (using the `provviz:color` attribute key, and a valid color in the hexadecimal notation as the attribute value)
- the shape of a specific node (using the `provviz:shape` attribute key, and a supported shape as the attribute value)
- whether or not a specific node is hidden in the visualisation (using the `provviz:hide` attribute key, and a Boolean as the attribute value)
- whether or not a specific node's attributes are hidden in the visualisation (using the `provviz:hideAttributes` attribute key, and a Boolean as the attribute value)

The benefit of embedding these settings as PROV attributes is that when the PROV document is exported from the *ProvViz Application* and re-uploaded at a later point as a different document, the visualisation settings are preserved.

Other visualisation settings that are not related to a single *Agent*, *Entity* or *Activity* cannot be embedded as attributes, and must therefore be stored in a separate visualisation settings data-structure. These visualisation settings include:

- the default colors of *Agents*, *Entities*, *Activities* and *Bundles*;
- whether or not to hide all PROV attributes in the visualisation;
- which namespaces are hidden in the visualisation;
- whether or not to present a provenance view (*Responsibility*, *Data Flow* or *Process Flow* view); and
- whether the direction of edges flow horizontally (right-to-left), or vertically (bottom-to-top).

These visualisation settings are stored in a JavaScript object, that is then serialized as JSON and stored by the *ProvViz Application* in the browser's local storage. Although these settings cannot be exported alongside the PROV document, they are preserved as long as the PROV document remains in the browser's local storage, enabling the settings to be persisted when the browser tab is closed, the browser is quit, or the user's device is restarted.

4.3 Programming Paradigms

4.3.1 Declarative Programming

Declarative programming is a design paradigm that focuses on the logic of programming expressions, in contrast to the imperative paradigm which focuses on the control-flow of the program. Where possible, the declarative programming paradigm will be used to implement functionality throughout the project on the highest level of abstraction possible, to avoid complex application logic.

4.3.2 React UI Components

The user interface of the *ProvViz Application* and *Visualiser* will be composed of a hierarchy of declarative functional React UI components. High-level components tend to be composed primarily of other React UI components, whereas lower-level components tend to be composed of a combination of HTML and Material-UI components such as buttons, input fields, etc. Constructing a hierarchy of components where each component is related to a specific piece of UI functionality also allows UI components to be reused throughout the hierarchy, preventing code duplication.

React UI components were historically defined using JavaScript classes. Recently however defining React UI components as JavaScript functions was made possible[4], to help make UI components more declarative and functional. Features such as *React Hooks* provide functional React components with the same functionality as class React components in the form of functional side-effects, which is why they are becoming modern best-practice for writing UI components in React. This is why the *ProvViz Application* and *ProvViz Visualiser* will be strictly composed of functional React UI components.

[4] https://reactjs.org/docs/components-and-props.html#function-and-class-components

4.4 Graphical User Interface

The user interface is an important aspect of many software applications, but is an especially vital aspect of applications where accessibility and ease-of-use are primary objectives. Therefore the design of the *ProvViz Application* and *Visualiser* user interfaces will be carefully considered.

4.4.1 ProvViz Application GUI

The *ProvViz Application* GUI is the most high-level interface of the application. There are two primary interfaces it displays to the user: the "Start View" and the "Editor View". The common aspects of the views is the menu bar which is present in every state of the application, providing functionality relevant to each view. Throughout all states of the application it enables the user to "Open" a document using a drop-down menu. The wire-frame depicted in Figure 4.4 illustrates the minimal view of the *ProvViz Application*, with a menu bar and the "Open" drop-down menu.

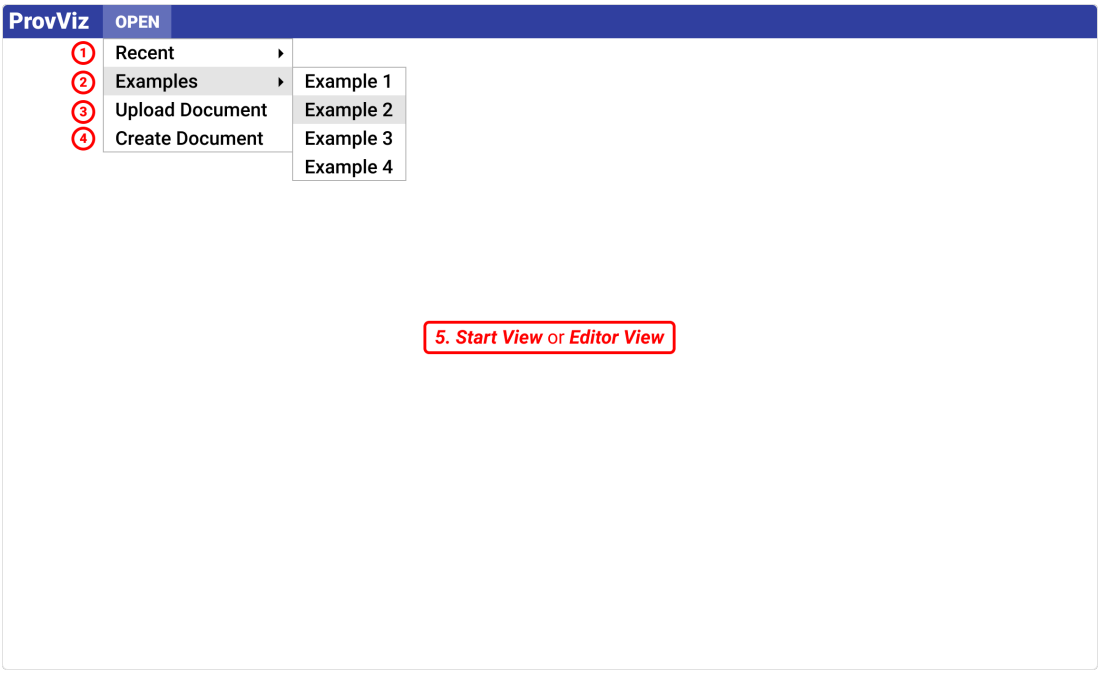


Figure 4.4: *ProvViz Application* Wire-Frame

By interacting with the "Open" drop-down menu the user has the option to select a recent document if there are any (Figure 4.4.1) (requirement AF7), select an example document (Figure 4.4.2) (requirement AF6), upload a document using the "Upload Document Dialog" (Figure 4.4.3) (requirement AF1), or create a new document using the "Create Document Dialog" (Figure 4.4.4) (requirement AF5).

Upload Document Dialog

The "Upload Document Dialog" allows the user to upload a PROV document in one of the supported PROV formats (PROV-N, PROV-JSON, PROV-XML, TriG or Turtle). Using the dialog, the user can trigger the native file system explorer, so that they can select a PROV document located on the device's file system. Once the file is uploaded, it is translated to the serialisation format, and the PROV document's name and PROV format is parsed from uploaded file's filename.

An input text-field is automatically filled with the parsed name, so that the user has the option to modify it. The PROV document name field is required, and must be unique amongst all previously loaded PROV documents. Appropriate error messages will inform the user if one of the validation constraints is violated.

When the input field is validated, the user can then click an "Upload" button which loads the PROV document and transitions the interface to the "Editor View" if necessary.

Create Document Dialog

The "Create Document Dialog" allows the user to create an empty PROV document in one of the supported PROV formats. An empty PROV document consists of no *Entities*, *Agents*, *Activities*, *Bundles* or relationships, and contains the `prov` and `xsd` namespaces.

An input text-field is automatically filled with a unique PROV document name that satisfies all validation constraints. A select input field let's the user choose the PROV format of the document, where PROV-N is the default chosen format.

When all input fields are validated, the user can click a "Create" button that loads an empty PROV document, and transitions the interface to the "Editor View" if necessary.

Start View

The "Start View" is the interface the user encounters when no PROV document is loaded in the application, and is therefore the first interface of the application a user sees. As user retention is an important aspect of this project, the "Start View" should feature the functionality available to load a PROV document so that the *Text Editor* and *Visualiser* are as accessible as possible. The wire-frame depicted in Figure 4.5 illustrates the devised interface of the "Start View".

When there are PROV documents stored in the browser's local storage, the "Start View" presents the three most recent locally stored documents at the top of the view (Figure 4.5,1). Clicking on a recent document will open it in the "Editor View". The recent document selector

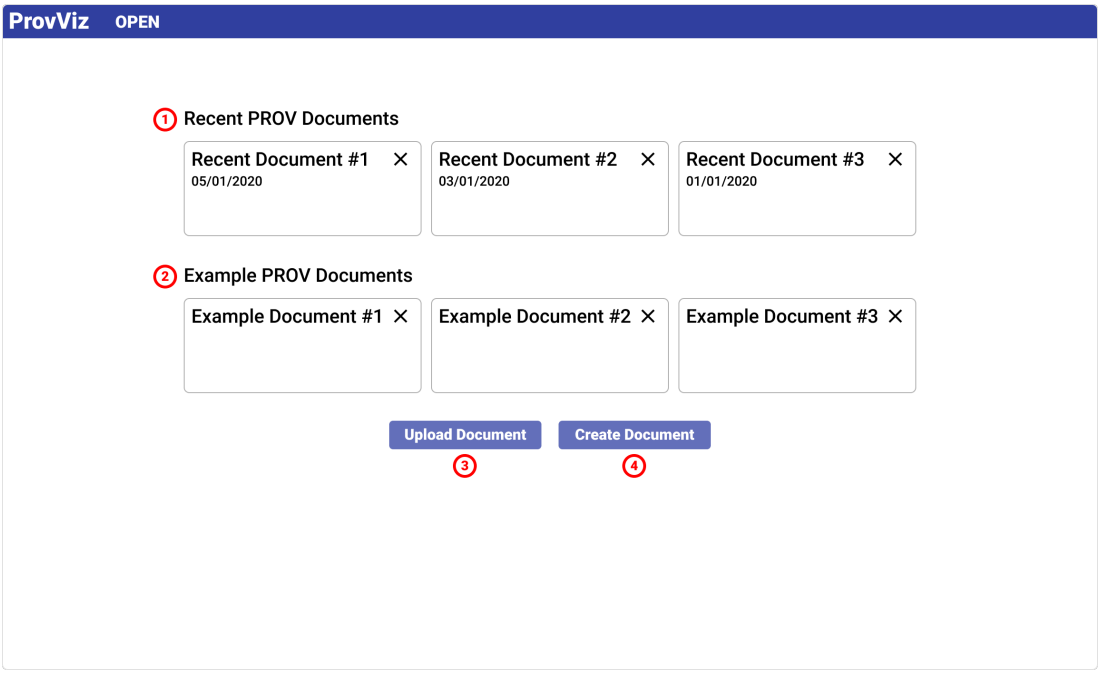


Figure 4.5: "Start View" Wire-frame

is located at the top of the "Start View", as this will be the most common interaction with the "Start View" when the user has used the *ProvViz Application* before.

Next, three example PROV documents are presented (Figure 4.5.2), that when clicked are opened in the "Editor View". This is available to all users, whether they have used the *ProvViz Application* before or not.

Finally, buttons triggering the "Upload Document Dialog" and "Create Document Dialog" are presented (Figure 4.5.3 and 4.5.4), if the user wants to upload or create a new PROV document.

All document loading functionality facilitated by the "Start View" is accessible by a single button click, as opposed to the two or more button clicks required when interacting with the menu bar's "Open" drop-down menu. This should should result a significant improvement in user retention, an important design consideration of the first interface users interact with.

Editor View

The "Editor View" is the interface that enables the user to modify and visualise PROV documents, and is therefore where the primary functionality of the application lies. The wire-frame depicted in Figure 4.6 illustrates the layout of the "Editor View".

The menu bar gains additional functionality when the current application view is the "Editor View". An "Export" button (Figure 4.6.1) opens the "Export Document Dialog", which enables

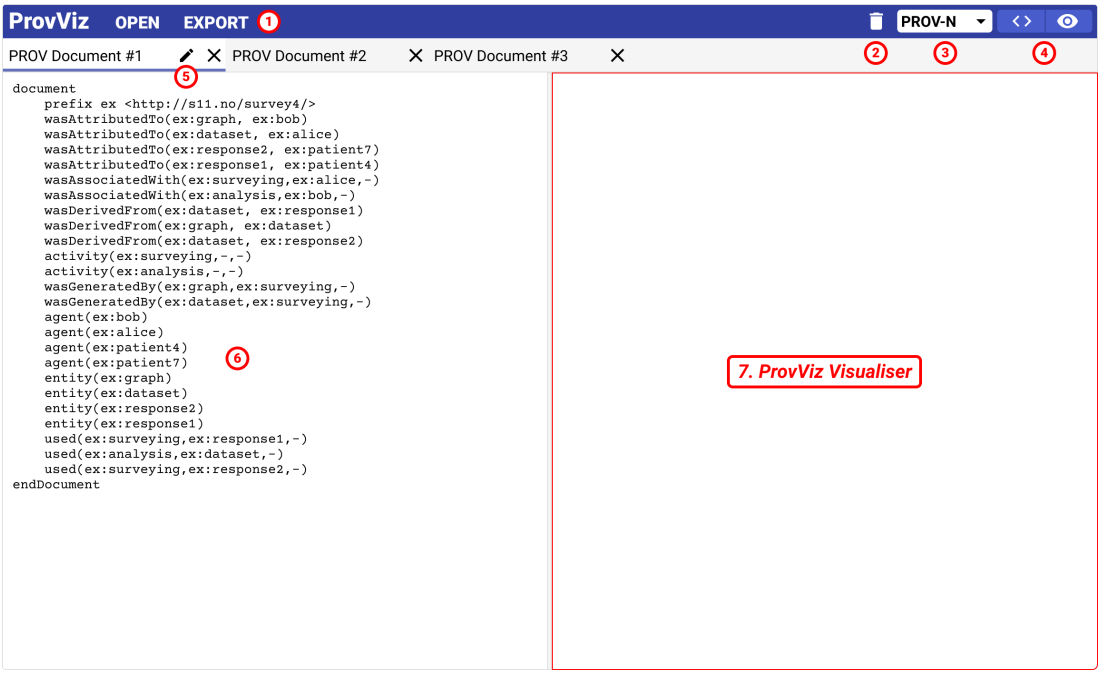


Figure 4.6: "Editor View" Wire-frame

the user to export the current PROV document in a desired PROV format (requirement AF2). A delete icon button enables the user to permanently delete the current PROV document (Figure 4.6.2).

A select input field (Figure 4.6.3) enables the user to change the PROV format of the current PROV document. Because the *ProvToolBox* API can only translate valid PROV documents, the serialisation is used instead of the text-editor's content, as it represents the most up-to-date valid version of the current PROV document.

The layout of the "Editor View" itself is split into three main components:

- a "Tabs" component (situated directly beneath the menu bar),
- a text editor component that enables the user to directly manipulate the current PROV document in the currently selected PROV format (Figure 4.6.6) (requirement AF8), and
- the *ProvViz Visualiser* component (Figure 4.6.7).

The "Tabs" component enables the user to navigate loaded PROV documents, modify the name of the currently open PROV document by clicking the edit name icon button (Figure 4.6.5), and close a tab by clicking its close icon button. The "Tabs" component also contains the saving status of the PROV document. When the user modifies the PROV document using the text editor component, the document must be translated to the serialisation format

asynchronously using the *ProvToolBox* API. For the duration of this saving process, a loading icon is displayed on the right side of the "Tabs" component. If an error is encountered (because the current state of the text editor contains syntactical errors, or a network error occurs in the HTTP request) an appropriate error message is displayed.

This layout takes inspiration from a variety of popular existing code editors, such as *Visual Studio Code*[5] or *Sublime Text*[6], which will help make the interface seem more familiar to users who have previously used those applications.

To facilitate a responsive layout of the components that functions at a variety of screen dimensions and use-cases, the user can select whether to display the text editor and visualiser simultaneously next to each other (as displayed in the wire-frame in 4.6), or just one at a time filling the entire width of the application, by interacting with a button group in the menu bar (Figure 4.6.4). Toggling each icon button displays or hides the text editor and visualiser component in the layout. In addition the line dividing the text editor and visualiser when they are displayed simultaneously can be dragged to change the proportion of each component's width in the layout. At small screen widths, presenting the text editor and visualiser simultaneously will not be an option to prevent overflowing UI elements (requirement ANF3).

Export Document Dialog

The "Export Document Dialog" allows the user to export the current PROV document in a desired PROV format (requirement AF2). A select input field allows the user to choose the PROV format of the exported document, where the default chosen format is the one currently displayed in the "Editor View". The user can click an "Export" button that creates a new file with the appropriate filename, file contents, and filename extension (according to the selected PROV format).

4.4.2 ProvViz Visualiser GUI

The *ProvViz Visualiser* is responsible for visualising a PROV document, and providing intuitive editing functionality for modifying the PROV document and the visualisation that is produced. The wire-frame depicted in Figure 4.7 illustrates the layout of the *Visualiser* component, when a non-empty valid PROV document is provided as input to the *Visualiser*.

The layout of the *Visualiser* component can be partitioned into three main components:

- the menu bar component,

[5] https://code.visualstudio.com/
[6] https://www.sublimetext.com/

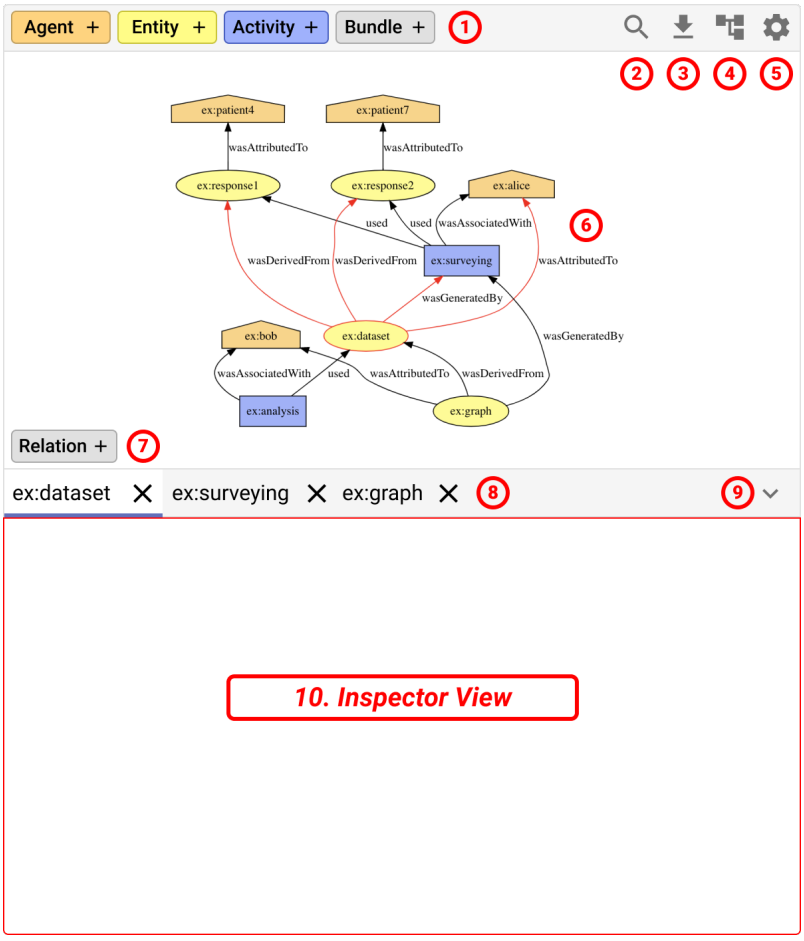


Figure 4.7: *Visualiser* Wire-frame

- the visualisation component, and
- the inspector component.

The menu bar component provides the functionality that is immediately accessible to the user. This includes four buttons for creating *Agents*, *Entities*, *Activities* and *Bundles* (Figure 4.7.1) (requirement VF8). A search icon button (Figure 4.7.2) enables the user to search for an *Agent*, *Entity*, *Activity* or *Bundle* by its name using a text input field. If the current view of the Visualisation component is the "Graph View", it is automatically transitioned to the "Tree View". A download icon button (Figure 4.7.3) enables the user to download the current visualisation in the SVG image format (requirement AF3). A view icon button (Figure 4.7.4) enables the user to toggle the current view of the visualisation component, where possible options include a "Graph View" (as displayed in the Figure 4.7.6) or a "Tree View". The distinct roles of the "Graph View" and "Tree View" will be discussed further when examining the visualisation component of the layout. The final item in the menu bar component is the

settings icon button (Figure 4.7.5) which enables the user to open the "Settings Inspector" in the inspector component.

Graph View

The default view of the visualisation component is the "Graph View", a provenance graph consisting of nodes and directed edges. The *GraphViz* visualisation library[7] is used to determine the layout of the nodes and edges in the graph, which produces a visualisation in the form of an SVG document (requirement VF1). The *D3* DOM manipulation library[8] is then used to provide additional functionality to the SVG document, such as enabling animated transitions between states in the GraphViz visualisation, enabling better inspection of the SVG by dragging and zooming (requirement VF2), and enabling the user to select *Activities*, *Entities*, *Agents* and *Bundles* by clicking on their corresponding nodes in the graph.

The "Graph View" additionally provides an intuitive PROV editing functionality, enabling the user to draw a relationship between two nodes using the create relationship button (Figure 4.7.7). First the user must select either an *Agent*, *Entity* or *Activity*, which triggers the create relationship button to appear. When the button has been pressed, a menu appears enabling the user to select the relation type (*wasAttributedTo*, *wasDerivedBy*, etc) that is a valid outgoing relation of the selected node. Then the user can draw the relation by selecting the destination node in the graph. This method for intuitively creating a PROV relation depends on the mouse hover event, and therefore is not available to touch users. The "Node Inspector" will provide touch-accessible PROV relationship creation functionality, which will be discussed in a later section.

Tree View

The "Tree View" is a hierarchical nested view, as illustrated in Figure 4.8. The motivation behind the "Tree View" is to allow for better visual navigation of PROV documents, as *Bundles* can be used to group and collapse items in the nested tree, and to provide an intuitive interface for moving *Agents*, *Entities* and *Activities* from one *Bundle* to another (requirement VF11) by simply drag-and-dropping any node in the tree. The `react-sortable-tree` NPM package[9] implements a Tree React component, with the desired drag-and-drop functionality, and will therefore be used as an existing maintained solution.

[7] https://graphviz.org/
[8] https://d3js.org/
[9] https://www.npmjs.com/package/react-sortable-tree

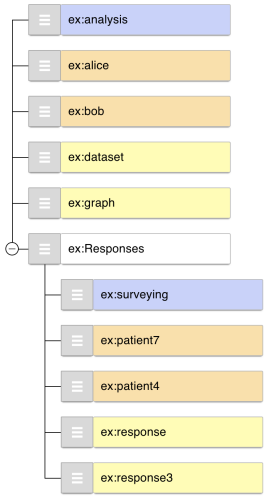


Figure 4.8: "Tree View" of PROV document

Settings Inspector

The "Settings Inspector" is opened as a tab in the inspector component when the settings icon button is pressed in the menu bar component, and accommodates two types of functionality: global namespace and global visualisation settings editing functionality.

The global namespace refers to all the namespace declarations that are globally accessible in the PROV document (that is all namespace declarations that are not made from within a *Bundle*). The editing functionality for *Bundle* specific namespace declarations will be implemented in the "Bundle Inspector", which will be discussed in a later section. Namespace declarations can be created by the press of a button (requirement VF6), and any existing namespace declaration can be modified by editing its prefix or value using text input fields (requirement VF7). Existing namespace declarations can also be removed by the press of a button (requirement VF6), but only when they are not being referenced by any existing *Agent*, *Entity*, *Activity*, *Bundle* or attribute. An additional visibility icon button can be pressed to toggle whether or not the *Agents*, *Entities*, *Activities* or *Bundles* in a specific namespace are displayed in the visualisation (requirement VF3).

The global visualisation settings are defined as the visualisation settings that are not related to a specific *Agent*, *Entity*, *Activity*, *Bundle* or namespace. This includes the following visualisation settings:

- selecting the current provenance view of the "Graph View" (requirement VF4),
- selecting the default color of *Agents*, *Entities* and *Activities* (requirement VF13),

- whether or not to hide the attributes of all *Agents*, *Entities* and *Activities* in the "Graph View", and
- whether the direction of edges in the "Graph View" flow horizontally (right-to-left), or vertically (bottom-to-top).

The final functionality the "Settings Inspector" provides is a "Reset Visualisation Settings" button, which enables the user to restore the default visualisation settings.

Node Inspector

The "Node Inspector" provides editing functionality related to a specific *Agent*, *Entity* or *Activity* in the PROV document, and is opened when an *Agent*, *Entity* or *Activity* is selected in the visualisation component or in the "Bundle Inspector". The editing functionality provided includes editing the node's identifier, attributes, outgoing relationships with other nodes, and its visualisation settings.

Identifier editing functionality allows the user to select a prefix from the available namespace declarations to the node using a select input field (requirement VF9), and edit the name of the identifier using a text input field. The identifier is validated by ensuring that the name is not equal to the empty string, and that the identifier is unique amongst all other nodes in the PROV document. Upon validation, the identifier is automatically updated in the PROV document.

Attribute editing functionality enables the user to capture additional information associated with the selected node. These attributes are separated into *defined* attributes which are those defined in the PROV ontology [2], and *custom* attributes which are those defined by the user. A single attribute is made up of a *key*, a *value* and a *value type*. Defined attributes have predefined keys and value types. For instance, *Activities* have a "Started At Time" attribute with the key `prov:startTime` and a date-time value type. Custom attributes on the other hand let the user choose the key and value type, where the key cannot clash with the key of other defined attributes.

The outgoing relationships of the node can be edited using a group of auto-complete inputs for each possible outgoing relationship type of the node. This enablesx the user to create or delete an outgoing relationships (requirement VF10). The possible outgoing relationships for a node are all the relationships with a domain of the node's *type* (*Agent*, *Entity* or *Activity*). The auto-complete input for each relationship lets the user then create relationships between the domain node and all the specified range nodes in the auto-complete input, which enables the

user select from existing possible range nodes or create new ones. To provide documentation to the user for each relationship type (requirement VF3) an information icon button can be toggled to show or hide the documentation for each relationship, which is taken from the W3C PROV Namespace definitions [1] and cited accordingly.

The visualisation editing functionality available in the "Node Inspector" is the visualisation editing functionality related to the selected *Agent*, *Entity* or *Activity*, which includes:

- editing the color of the specific node in the "Graph View" using a color-picker (requirement VF14),
- editing the shape of the specific node in the "Graph View" using a select input field (requirement VF15),
- toggling whether or not the specific node is hidden in the "Graph View" using a checkbox input field (requirement VF3), and
- toggling whether or not the attributes of the node are hidden in the "Graph View" using a checkbox input field.

The final piece of functionality available in the "Node Inspector" is a delete button which enables the user to remove the selected *Agent*, *Entity* or *Activity* and all its outgoing and incoming relationships from the PROV document (VF8). A warning message informs the user if any outgoing or incoming relationships will be deleted as a consequence of this action.

Bundle Inspector

The "Bundle Inspector" provides editing functionality related to a specific *Bundle* in the PROV document, and is opened as a tab in the inspector component when a *Bundle* is selected in the visualisation component.

Similar to the "Node Inspector", the "Bundle Inspector" provides identifier editing functionality of the selected bundle (requirement VF9). The "Bundle Inspector" also provides the namespace declaration editing functionality outlined in the "Settings Inspector" section for the namespace declarations made inside the selected *Bundle*.

The *Agents*, *Entities* and *Activities* defined in the selected *Bundle* are also listed as clickable links, where clicking an *Agent*, *Entity* or *Activity* will select the node in the visualisation component and open its corresponding "Node Inspector" tab in the inspector component.

Finally, a delete button will enable the user to remove the selected *Bundle* and its contents from the PROV document (VF8). A warning message informs the user if any *Agents*, *Entities*

or *Activities* will be deleted as a consequence of this action.

4.5 Other Third-Party Content

To appropriately limit the scope of the project, and to make use of existing maintained solutions where possible, some additional third-party content will be used.

4.5.1 Material UI

Material UI[10] is a popular React component library, with nearly 2 million weekly downloads, that will be used to streamline the development of a responsive user interface for both the *ProvViz Application* and *Visualiser*. Material UI also features a collection of icons[11] which will used throughout the application. Both the react components and icons can be used for free under the Apache license[12].

4.6 Limitations

The design outlined for the *ProvViz Application* system comes with the following limitations:

a. The translation of PROV document formats depends on the *ProvToolBox* HTTP API, and therefore occurs asynchronously in the application. Not only does the HTTP protocol lead to a less efficient translation of PROV documents, but potential network errors have the ability to interrupt the user-flow of the application. Therefore these errors have to be accounted for to not disrupt or confuse the user when a PROV translation failed.

b. Use of the *GraphViz* visualisation library limits the extent to which the "Graph View" provenance visualisations can be customised by the user. The library accepts input graphs in the form of the *dot* language. Therefore, the visualisation editing functionality is limited to what can be expressed in the *dot* language.

c. Although the *dot* language does provide some rudimentary custom positioning functionality of nodes in the layout[13], it is only available when using the *neato* or *fdp GraphViz* layout engines. The *dot* layout engine was chosen for this project because it better produces directed graphs with non-overlapping but densely populated nodes in the graph. In

[10] https://material-ui.com/
[11] https://material-ui.com/components/material-icons/
[12] https://www.apache.org/licenses/LICENSE-2.0
[13] https://www.graphviz.org/doc/info/attrs.html#d:pos

addition the positioning parameter's units don't translate to the pixel dimensions of the produced SVG rendered in the browser, making drag-and-drop editing functionality hard to implement, and leading to unexpected results if the user were to modify the parameter manually. Therefore, an additional short-coming of the devised *ProvViz* design is the lack of support for modifying the position of *Agents*, *Entities* and *Activities* (requirement VF16).

Chapter 5

Implementation & Testing

This chapter is dedicated to outlining the implementation phase of the project, which was guided by the designs laid out in the previous chapter. The implementation phase of the project was partitioned into first implementing the functionality of the *ProvViz Visualiser*, followed by the implementation of the *ProvViz Application*. This is because the functionality of the latter depends on the implementation of the former.

5.1 ProvViz Visualiser Implementation

5.1.1 PROV Queries & Mutations

As the *Visualiser* provides intuitive PROV editing and visualisation functionality, the PROV document's serialisation needs to be queried and modified in many of the implemented React components. Therefore a collection of composable query and mutation functions were devised, written primarily in the functional programming style to adhere with the declarative programming paradigm. The devised functions are pure, meaning they have no side-effects, always producing the same result when provided with identical inputs.

Most of the mutation functions are higher-order functions, taking various parameters as input, and returning a function that takes a PROV-JSON document as input returns the modified PROV-JSON document as the final output.

$$(...\text{arguments}) \rightarrow (\text{Input PROV Document}) \rightarrow \text{Output PROV Document}$$

The TypeScript function implemented in Listing 5.1 is an example of a higher-order mutation

function for creating an *Agent* in an PROV-JSON document. The function takes the identifier of the *Agent* as an argument, and returns a function which takes a PROV-JSON document as its input and returns the same PROV-JSON document, except with the new *Agent*. The `lodash.clonedeep` NPM package[1] is used to clone the input PROV-JSON JavaScript object, so that the output PROV-JSON JavaScript object is completely detached from the input. This ensures the function is pure, always returning the identical PROV-JSON JavaScript object when supplied with the identical input.

```
// import the cloneDeep function from the 'lodash.clonedeep' NPM package
import cloneDeep from 'lodash.clonedeep';


// mutation function for creating an agent
const createAgent = (
    // the arguments
    id: string
) => (
    // the input PROV document
    document: PROVJSONDocument
): PROVJSONDocument => cloneDeep({
    // spread the input document over the return object...
    ...document,
    // ...and add the identifier to the existing agents
    agent: { ...document.agent, [id]: { } },
});
```

Listing 5.1: Create Agent TypeScript Function

5.1.2 Graph View Visualisation

The "Graph View" visualisation was implemented by generating the layout of nodes and edges using the *GraphViz* library, as outlined in the design phase. A collection of mapping functions were implemented that map the PROV-JSON serialisation to the *dot* language taking any visualisation settings into consideration. The produced *dot* statements are then used as the input for the *GraphViz* WebAssembly module, producing the desired SVG output. The W3C

[1] https://www.npmjs.com/package/lodash.clonedeep

PROV Model Primer [5] example visualisations were taken as inspiration for the default "Graph View" visualisation, so that produced visualisations are familiar to users who have read some of the W3C articles about the PROV model.

5.1.3 PROV-JSON Document Validation

As depicted in Figure 4.1, the `document` prop of the *ProvViz Visualiser* accepts a JavaScript object. During the implementation phase however two implications of this decision became apparent:

1. the `document` parameter could contain invalid PROV-JSON syntax; and
2. if the `document` represents a valid PROV-JSON document syntactically, some additional assumptions need to be checked before the document can be visualised (for example, whether or not a relationships references undefined *Entities*, *Agents* or *Activities*).

Therefore, validating the `document` prop and displaying appropriate error messages to the user became an additional implementation task.

JSON Schema Validation

To ensure that the `document` prop represents a syntactically correct PROV-JSON document, the document is validated using a JSON schema. An existing JSON schema for PROV-JSON documents was found in the W3C PROV-JSON Serialization submission [3], and can be accessed directly here[2]. The JSON Schema used by the *ProvViz Visualiser* can be found in Appendix A.1, which was adapted from the JSON schema submitted to the W3C organisation to conform to the specifications of JSON-Schema Draft-07[3], the latest release of the JSON-Schema specification at time of writing.

The `ajv` NPM package[4] was used as the JSON schema validation library. It is one of the most popular libraries, and the most performant validator for Draft 7 of the JSON-Schema specification [11].

Additional Validation Checks

Once the PROV-JSON document has been validated as syntactically correct using its JSON schema, several other checks are made:

[2] https://www.w3.org/Submission/2013/SUBM-prov-json-20130424/schema
[3] https://json-schema.org/draft-07/json-schema-release-notes.html
[4] https://www.npmjs.com/package/ajv

1. For each relationship, the domain and range identifiers are checked whether a corresponding *Agent*, *Entity* or *Activity* exists with that identifier. If either the domain or the range identifier does not have a corresponding *Agent*, *Entity* or *Activity* the "Graph View" component will encounter an error when attempting to draw the relationship in the "Graph View". Therefore an error message is displayed informing the user an undefined *Agent*, *Entity* or *Activity* has been referenced in a relationship.

2. The identifier of each *Agent*, *Entity*, *Activity* and *Bundle* is checked whether or not it is in a declared namespace. If its namespace is not declared, an error message is displayed informing the user.

3. The attribute keys of each *Agent*, *Entity*, *Activity* and relationships is checked whether or not it is in a declared namespace. If its namespace is not declared, an error message is displayed informing the user.

4. If the required namespaces `prov`[5], `xsd`[6], or `provviz`[7] are undefined, they are automatically appended to the namespace of the PROV-JSON document.

5.1.4 Debouncing Text-Field Input

To ensure text-fields are responsive to user input, the PROV document serialisation is not immediately updated when a user interacts with a text-field. This is because typing into a text-field triggers a rapid succession of keystroke events. If each event were to modify the PROV document serialisation individually, the text-field input may become unresponsive to the user for large provenance graphs. This is because JavaScript is a single-threaded language, meaning updating text-field inputs to reflect user input can be delayed by CPU intensive activities such as re-rendering the *Visualiser*. The performance of rendering large provenance graph in the *Visualiser* is further discussed in the performance evaluation (section 7.1.1).

A common technique to ensure text-fields are responsive to user input is "debouncing" any CPU intensive activities triggered by updating the text-field (such as updating the PROV document serialisation). Debouncing a function means delaying when a function is invoked until a certain amount of time has elapsed since the last time the function was called. In practice, this means that the PROV document serialisation is only updated when a user has not typed into a text-field for a certain amount of time, making the text-field feel more responsive to the

[5] http://www.w3.org/ns/prov#
[6] http://www.w3.org/2001/XMLSchema#
[7] https://provviz.com/ns/provviz#

user.

The `lodash.debounce` NPM package[8] was used to implement the debounce functionality for text-field inputs.

5.1.5 Continuous Integration

Continuous Integration is the automated process of ensuring that the latest stable version of an application is distributed. This is an important aspect of open-source projects, so that third-party code contributions can be released without requiring the maintainer of the project to perform a tedious sequence of steps. When continuous integration is implemented effectively, code contributions and bug fixes are more swiftly integrated into the distributed version of the application.

As an NPM package, the *ProvViz Visualiser* needs to be built, tested and published to the NPM package registry[9] in order to be released. This process includes numerous steps, which were codified into an automated GitHub Actions workflow. GitHub Actions[10] is a free to use[11] continuous integration tool, popular amongst open-source projects to build, test and deploy software applications. In addition, the `semantic-release` NPM package[12] was used to automate the process of selecting a version number, generating release-notes and publishing the *ProvViz Visualiser* to the NPM package registry. The continuous integration workflow is triggered on every git commit to the `main` branch of the project, and performs the following steps:

1. **Install** any external dependencies (NPM packages)
2. **Test** the *ProvViz Visualiser* by executing the implemented test-suite (if any tests fail, the workflow is aborted to prevent distributing a failing version)
3. **Build** the *ProvViz Visualiser*, which includes compiling the TypeScript source-code into native JavaScript and a "tree-shaking" process that removes any dead code (code that is in the scope of the project, but never used)
4. **Publish** the *ProvViz Visualiser* to the NPM package registry[13] using the `semantic-release` NPM package

[8] https://www.npmjs.com/package/lodash.debounce
[9] https://www.npmjs.com/
[10] https://github.com/features/actions
[11] at time of writing, 2000 monthly run-time minutes
[12] https://www.npmjs.com/package/semantic-release
[13] https://www.npmjs.com/package/provviz

5.2 ProvViz Application Implementation

5.2.1 Example PROV Documents

As specified in the design, example PROV documents can be loaded to start using the *ProvViz Application*. The origin of each chosen example document is cited on the homepage of the application.

The first chosen example document was taken from the W3C PROV Model Primer [5], where the data provenance of an example newspaper article is explored with the aim of introducing the reader to modelling provenance using the PROV model. This example was chosen so that users who read the PROV Model Primer have a familiar starting point when using the *ProvViz Application*, and that those who have not yet read the PROV Model Primer are made aware of its existence.

The second example document models the provenance of an example survey, and was taken from the *ProvStore* [17], a service hosted by the *Open Provenance* organisation that provides free storage, viewing and collaboration on provenance documents. The purpose of this example is to demonstrate a different provenance domain, and introduce the user to the large collection of PROV documents made available by the *ProvStore*.

The final example document models the provenance of the *Provenance: An Introduction to PROV* book [16]. This document was chosen both because it introduces users to a book that provides a comprehensive introduction to PROV, and because it is a large PROV document relative to the other examples, showcasing how the *ProvViz Application* performs with a large document composed of 85 *Agents*, *Entities*, *Activities* and relationships.

5.2.2 Continuous Integration

Similar to the *ProvViz Visualiser*, the *ProvViz Application* benefits from a continuous integration workflow. Because the *ProvViz Application* is a web-app, it needs to be deployed as a web-service in order to be released. The hosting provider *Vercel*[14] was chosen as it supports free hosting and continuous deployments of React applications. The *ProvViz Application* is automatically built, tested and deployed to the `https://provviz.com` domain on every commit to the `main` branch of its code repository.

[14] https://vercel.com/

5.3 Changes and Additions from the Design

5.3.1 Relationship Inspector

During the implementation phase of the project, it became apparent a "Relationship Inspector" tab was a logical addition to the "Node Inspector", "Bundle Inspector" and "Settings Inspector" tabs outlined in the design phase. This would provide intuitive editing functionality for the defined and custom attributes for a particular relationship. It would also enable the implementation of visualisation settings that would let the user hide a specific relationship, or hide a specific relationship's attributes.

5.4 Implementation Issues

5.4.1 Cross-Browser Support

Supporting multiple browsers is one of the main difficulties when developing a web-app, as the APIs supported by different browsers can differ substantially. To mitigate the extent to which web-developers have to account for different browser environments, it is considered best-practice to call native browser APIs as little as possible, using open-source "wrappers" to implement functionality instead. Material-UI components can be considered as such a wrapper, abstracting a significant amount of browser-compatibility functionality, one of the reasons as to why Material-UI components were used heavily throughout the project.

However other packages were used in addition to Material-UI components. For example, to export PROV documents in the *ProvViz Application*, the browser API for downloading files has to be called. This browser API is implemented differently across the various browsers that exist. Some browsers support exporting files in the form of a `Blob` data structure[15], others require the file to be encoded into a Data URI[16], some support specifying the filename whilst others don't, and some browsers don't support saving files at all. The open-source `file-saver` NPM package[17] provides a simple `saveAs` function, which when called by the *ProvViz Application* handles the nuances of the various browser environments, saving significant amounts of development time. In addition, because the package is open-source, when a relevant browser API changes in the future the package will be updated, improving the maintainability of this project.

[15] https://developer.mozilla.org/en-US/docs/Web/API/Blob
[16] https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs
[17] https://www.npmjs.com/package/file-saver

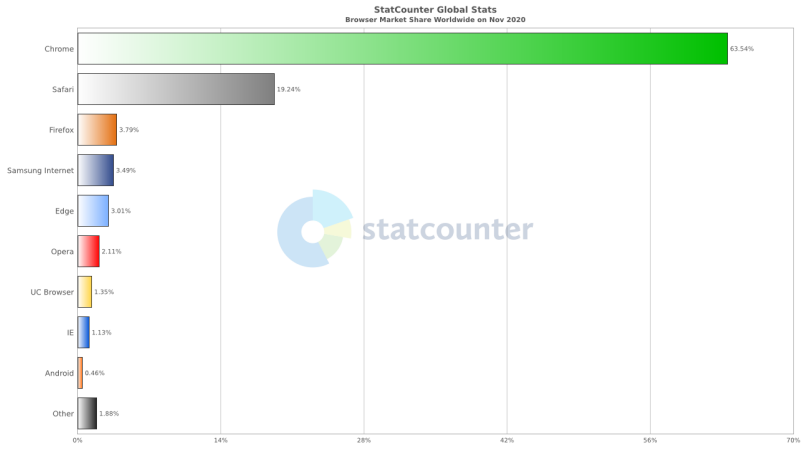


Figure 5.1: Worldwide Browser Market Share on November 2020 according to `https://gs.statcounter.com/`

In addition to these best-practices, the application was additionally tested in the two most popular browsers: Chrome and Safari (as indicated by Figure 5.1). This helped reveal unaccounted for cross-browser support issues. For instance, the `datetime-local` input element[18] was used, but cross-browser testing revealed that this input type is not yet been supported by the *MacOS* version of *Safari*. Therefore the `@material-ui/pickers` NPM package[19] was used instead to provide this functionality to both browsers.

Although these mitigation techniques were followed during the development process of the *ProvViz Application* and *ProvViz Visualiser*, cross-browser support of functionality remains a challenge worth acknowledging. Browser APIs change over time, and future updates of the application may be necessary to ensure its functionality remains well-supported across the majority of popular web-browsers.

5.5 Testing

An important component of software systems is not just the implemented solution, but testing that it works. This is especially important in this project because the continuous integration pipelines of the *ProvViz Visualiser* and *ProvViz Application* are configured to release committed code changes immediately, where the test-suite is what is used to prevent deploying broken changes. This section outlines the testing methodology adopted by the software systems.

[18] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/datetime-local
[19] https://www.npmjs.com/package/@material-ui/pickers

5.5.1 Unit Testing

Unit testing is the process of testing individual code modules used by other components, to ensure they behave as expected.

The mutations and queries implemented in the *ProvViz Visualiser* are an integral component of the devised software system, used frequently throughout the component hierarchy of the *ProvViz Visualiser*. The individual query and mutation functions were therefore deemed ideal candidates for unit tests, to help catch edge cases and ensure they behave as expected.

5.5.2 Integration Testing

Integration testing is the process where software units are combined and tested as a collective, to ensure they work together as expected.

As stated in the React testing documentation, for React UI components "the distinction between a 'unit' and 'integration' can be blurry" [6]. Some UI components such as buttons can be thought of as individual units of functionality, whereas more higher-level components such as the menu bar are composed of many units of code. Testing the latter is therefore in general considered to be integration testing, and is where the PROV queries and mutations are combined to provide a variety of functionality for components. The most critical React UI components devised for the system are tested to see if they render the expected elements based on their input, where user interactions are additionally tested to see if the component behaves as expected. This was done for the *ProvViz Visualiser* and the *ProvViz Application*.

5.5.3 Requirement Based Testing

Requirement based testing is the process of ensuring the requirements have been satisfied by the implemented software system, so that the devised software can be considered as complete. This was the final stage of the testing phase, where the implementation of each requirement was verified. The requirement based testing conducted will support the evaluation chapter of the report.

Application Requirements

Must have requirements:

- AF1 - Clicking on the "Upload Document" button in the "Open" menu bar drop-down, or on the homepage, opens a "Upload Document Dialog" that lets the user upload a

document.

- AF2 - Clicking on the "Export" button in the menu bar opens a "Export Document Dialog" that lets the user export a document
- AF3 - Clicking the download icon button in the *Visualiser* downloads the current visualisation as an SVG image.
- AF9 - The application uses the `provviz` NPM package[20] to visualise PROV documents and provide intuitive editing functionality.
- ANF1 - The application can be accessed at `https://provviz.com` and can be used directly without downloading or installing any prerequisite software (apart from the browser itself).

Should have requirements:

- AF5 - Clicking on the "Create Document" button in the "Open" menu bar drop-down, or on the homepage, opens a "Create Document Dialog" that lets the user create a new empty document.
- AF6 - Three example documents are featured on the homepage and accessible in the "Examples" sub-menu of the "Open" menu bar drop-down, as discussed in section 5.2.1.
- AF7 - Any loaded documents are automatically stored in the browser's local storage, and can be reopened on the homepage or in the "Recent" sub-menu of the "Open" menu bar drop-down.
- AF8 - The current PROV document can be modified directly using a text editor in any of the supported PROV formats (PROV-N, PROV-JSON, TriG, PROV-XML and Turtle).

Could have requirements:

- AF4 - The "Graph View" visualisation **cannot** be exported in image formats other than the SVG image format, meaning this requirement has not been met.
- ANF2 - The application is fully touch accessible, with alternative input methods provided for features that are not touch accessible. For instance the relationships auto-complete input in the "Node Inspector" serves as a touch accessible alternative to the create relationship functionality in the "Graph View", as outlined in the design.

[20] https://www.npmjs.com/package/provviz

- ANF3 - The application implements a responsive design, fully accessible for screen sizes with a width above 560 pixels. The "Start View" of the application is also accessible for screen sizes with a width of 360 pixels.

Visualiser Requirements

Must have requirements:

- VF1 - The *Visualiser* generates a "Graph View" visualisation using the *GraphViz* framework and renders it in the browser's DOM
- VF2 - Using the pointer device (a mouse/touch-pad for non-touch devices, or the finger for touch devices) the "Graph View" visualisation can be panned and zoomed
- VF3 - Namespace declarations can be hidden in the "Graph View" by clicking a checkbox in the "Settings Inspector" for global namespace declarations, and in the "Bundle Inspector" for the namespace declarations made in a bundle. *Agents*, *Activities* and *Entities* can be hidden in the "Graph View" by clicking a checkbox in the "Node Inspector". Relationships can be hidden in the "Graph View" by clicking a checkbox in the "Relationship Inspector".
- VF6 - Namespace declarations can be created by clicking the "Create" button in the namespace section of the "Settings Inspector" or "Bundle Inspector". They can be deleted by clicking the delete icon button in the same sections.
- VF7 - The prefix and value of a namespace declaration can be modified by interacting with the corresponding text-fields in the namespace section of the "Settings Inspector" or "Bundle Inspector".
- VF8 - *Agents*, *Entities*, *Activities* and *Bundles* can be intuitively created by pressing the corresponding button in the *Visualiser*'s menu bar. *Agents*, *Entities* and *Activities* can be additionally created when interacting with an outgoing relationship auto-complete input in the "Node Inspector".
- VF9 - The namespace of an *Agent*, *Entity*, *Activity* or *Bundle* can be modified by selecting an item from a select input, and the name can be modified by interacting with a text-field.
- VF10 - When having selected an *Agent*, *Entity* or *Activity* an outgoing PROV relationship to another node can be created by clicking the create relationship button in the "Graph

View", or interacting with the outgoing relationship auto-complete input in the "Node Inspector". Relationships can be deleted from the same auto-complete input, or by clicking the "Delete" button in the relationship's corresponding "Relationship Inspector" tab.

- VF11 - *Agents*, *Entities* and *Activities* can be moved from one *Bundle* to another by drag-and-dropping the corresponding nodes in the "Tree View" visualisation.
- VF12 - A defined PROV attribute can be modified in the "Node Inspector" for *Agents*, *Activities* and *Entities*, and in the "Relationship Inspector" for relationships. Custom PROV attributes can be created, modified and removed in the "Custom Attributes" section of the "Node Inspector" and "Relationship Inspector".

Should have requirements:

- VF4 - The "Settings Inspector" of the *Visualiser* provides a select input that enables the user to select a provenance view for the "Graph View" visualisation.
- VF5 - Documentation for *Entities*, *Activities* and *Agents* can be accessed by clicking the info icon button in the "Node Inspector". Documentation for *Bundles* can be accessed by clicking the info icon button in the "Bundle Inspector". Documentation for relationships can be accessed by clicking the info icon button next to the relationship auto-complete input in the "Node Inspector", or by by clicking the info icon button on the "Relationship Inspector".
- VF13 - The default colour of *Agents*, *Activities* and *Entities* in the "Graph View" and throughout the application can be modified using a colour picker in the "Visualisation" section of the "Settings Inspector".
- VF14 - The colour of a specific *Agent*, *Activity* or *Entity* can be modified using a colour picker in the "Visualisation" section of its "Node Inspector" tab.
- VF15 - The shape of a specific *Agent*, *Activity* or *Entity* in the "Graph View" can be modified using a select input in the "Visualisation" section of its "Node Inspector" tab.
- VNF1 - Most user input is followed by feedback *instantaneously* within 100 milliseconds in the software system. For text-field inputs, updating the PROV document serialisation is "debounced" to ensure feedback remains instantaneous for every keystroke as explained in section 5.1.4. The only scenario in which feedback may not be instantaneous is for the duration the *Visualiser* re-renders when the PROV document serialisation is updated.

The performance of this re-rendering process is further discussed in the evaluation chapter (section 7.1.1).

Could have requirements:

- VF16 - The position of a specific *Agent*, *Activity*, *Entity* **cannot** be modified, meaning requirement VF16 was not met as explained in the limitations of the design (section 4.6)

Chapter 6

Legal, Social, Ethical and Professional Issues

Throughout the development of the *ProvViz* software system the key principles of the code of conduct outlined by the *British Computer Society* (BCS) were followed closely. The primary BCS principle that applies to the *ProvViz* project is producing software that is for everyone. A key aspect of the project is to make the PROV standard accessible to more people, on a platform that can be accessed on any modern personal computing device.

The system is composed of my own work and open-source libraries and frameworks. Third-party software was used to decrease development time, improve maintainability, and provide better overall software quality. Intellectual property was respected when using third-party software, where the licenses of the software modules were adhered to. Use of third-party packages is explicitly listed in the `package.json` files of both code repositories, and summarised in this report. The code repositories are public, and can be introspected at:

- `https://github.com/benwerner01/provviz-web` (*ProvViz Application*)
- `https://github.com/benwerner01/provviz` (*ProvViz Visualiser*)

The `package.json` files can also be found in the source code attached to Appendix C.

6.1 Terms of Use

The "Terms of Use" of the *ProvViz Application* are the terms the user agrees to when using the application, which are outlined at the bottom of the homepage of the application.

The *ProvViz Application* is a statically hosted *React* application without any server-side storage components, and therefore **does not** and **cannot** collect personal data about its users (such as cookies, IP addresses, etc). Any uploaded or created PROV document is stored client-side in the user's web-browser, and processed remotely only by the ProvToolBox API in order to provide the PROV translation functionality. Therefore by agreeing to the terms of using the *ProvViz Application*, the user must also agree to the terms of using the Open Provenance Web Services that provide the ProvToolBox API, which can be found at `https://openprovenance.org/ethics/provenance-web-services/` and are linked at the bottom of the homepage of the application. The security of any PROV document stored in the application is dependent on the security of the browser's local storage API implementation, and the security of the HTTP requests being sent to the ProvToolBox API.

Chapter 7

Evaluation

This section will evaluate the produced *ProvViz* software system and the project's development methodology in separate sections.

7.1 ProvViz Application Evaluation

As demonstrated by the requirement based testing (section 5.5.3), the *ProvViz* application fulfills all the *must have* requirements and can therefore be considered as a complete implementation. In addition it fulfills most of the *should have* and *could have* requirements. However, no software system is perfect, and the devised *ProvViz* application comes with several drawbacks worth acknowledging in its evaluation.

7.1.1 Limitations

Position Editing Functionality of *Entities*, *Activities* and *Agents*

One of the missed *could have* requirements is requirement VF16, changing the position of an *Agent*, *Entity* or *Activity* in the "Graph View". This is a consequence of using the *GraphViz* framework for determining the layout of nodes and edges, as discussed in the design limitations section 4.6.

Exporting the Visualisation in Alternate Image Formats

Another missed *could have* requirement is requirement AF4, exporting the "Graph View" visualisation in alternate image formats such as PNG and JPEG. As this was a *could have* requirement, it was one of the last requirements considered for implementation at which point

there wasn't enough remaining time in the development phase. Other *could have* requirements were prioritised because other applications already exist that let the user translate an SVG image to PNG, JPEG and other image formats.

Moving Relationships from one *Bundle* to Another

Late in the development phase the realisation was made that the PROV relationships should be treated as first-class citizens like *Entities*, *Activities*, *Agents* and *Bundles*, and be given its own inspector tab. This lead to the implementation of the "Relationship Inspector" as an addition to the design, as discussed in section 5.3.

Other design additions however could not be made in time, which includes adding relationships to the "Tree View" of the application so they can be moved from one *Bundle* to another, just like *Entities*, *Activities* and *Agents*. This feature will be further described in the future work (section 8.2). It is important to note that although the *ProvViz* application would benefit from this functionality, it remains an edge case piece of functionality. In the majority of use cases a user will be seeking to move *Entities*, *Activities* and *Agents* not relationships. In addition, the lack of this functionality does not break other functionality. Relationships in or outside of *Bundles* can be selected and modified. Finally, if the user needs to move a relationships from one *Bundle* to another they can always do so by directly modifying the PROV document using the text editor.

ProvViz Visualiser Performance for Large Documents

On every modification of the PROV document serialisation, the *ProvViz Visualiser* must be re-rendered to display any changes. This process includes validating the updated document and updating the visualisation. Of these processes, the most time-consuming one is updating the "Graph View" visualisation where the *GraphViz* engine is used to generate the updated layout. To better understand how the *GraphViz* layout engine render time scales with PROV documents, a performance analysis was conducted in which the provenance graph was scaled using nodes and relationships in separate tests.

The first test was to see how the *GraphViz* engine performs as the number of nodes in a PROV document (*Entities*, *Activities* and *Agents*) increase, with no relationships in the graph. The results are graphed in Figure 7.1, which depict a linear correlation between the render time and the number of nodes in the PROV document. The render time remains below 0.5 seconds for graphs with 75 nodes.

Figure 7.1: *GraphViz* Rendering Time vs. Number of Nodes in the PROV Document

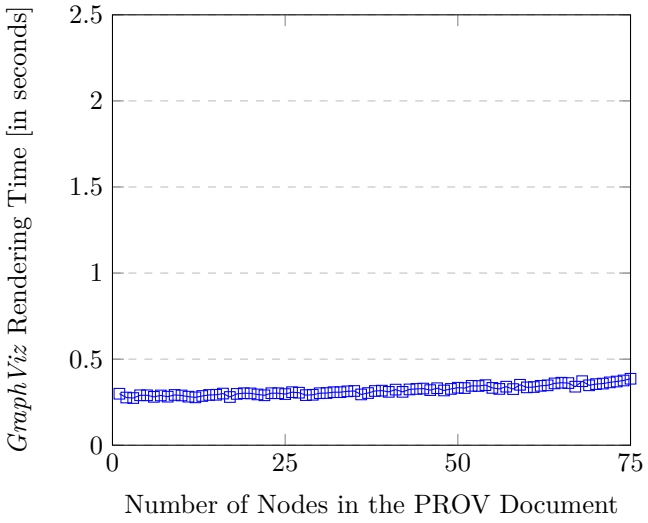
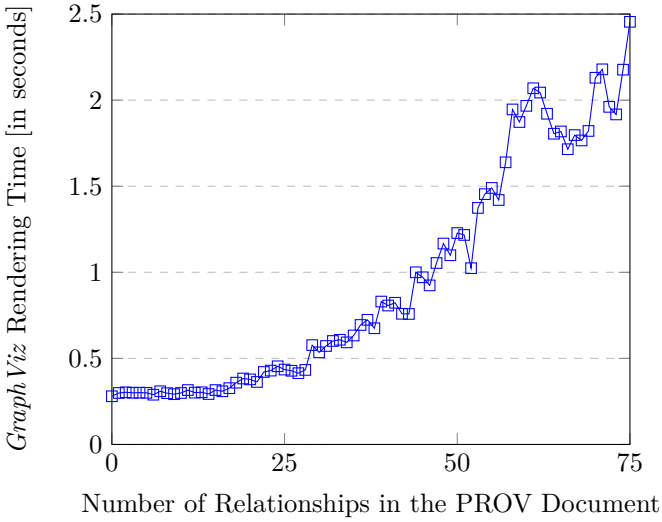


Figure 7.2: *GraphViz* Rendering Time vs. Number of Relationships in the PROV Document



A second test was conducted to better understand how the *GraphViz* engine performs as the number of relationships in the PROV document increase, with a constant number of nodes. An algorithm was devised for updating a PROV document with 5 *Entities*, *Activities* and *Agents* that on each iteration picks a possible relationship at random and adds it to the graph. The results for this test are graphed in Figure 7.2, which reveals the render time increases dramatically as the PROV document exceeds 50 relationships, with render times above 1 second. This is likely because the *GraphViz* engine is trying to find a layout with the least possible overlapping relationships, which becomes increasingly time-consuming when the number of relationships is high.

This is a significant limitation in the context of the *ProvViz* application, as the intuitive editing functionality provided relies on the "Graph View" visualisation to reflect the most recent version of the PROV document. Therefore when a modification to the serialisation is made, further user input is blocked until the *GraphViz* engine has finished devising the updated layout of the provenance graph. For large provenance graphs with a high number of relationships this can mean the interface is unresponsive for multiple seconds. Because JavaScript is a single-threaded programming language, multi-threading cannot be used to resolve this input blockage.

7.1.2 Overall Evaluation

The objective of the *ProvViz* application was to introduce an intuitive PROV document editing and visualisation tool, that is accessible and easy-to-use. With all the *must* and *should have* functionality and most of the *could have* functionality implemented in the *ProvViz* system, the application fulfills its requirements. None of the mentioned limitations prohibit the application from fulfilling its primary objectives. For large PROV documents the application remains functional, but users may encounter brief input blockage. However as a tool that aims to introduce users to the PROV model, small to medium PROV documents are expected to be the majority of use cases.

7.2 Project Evaluation

The first phase of the project was researching the background and related work, which proved to be the most important phase of the project. This is because one of the most challenging aspects of this project was gaining familiarity with the PROV model, and the variety of functionality it can provide. Dedicating more time to learning about the PROV model before the design phase

would have helped complete the design process in a more timely manor, and could have ensured the designs were more complete preventing changes and adaptations in the implementation phase.

Despite the previously mentioned shortcomings of the design phase, elements such as the architecture diagrams and the devised wire-frames significantly supported the implementation phase of the project.

The implemented test-suites during the final stages of the implementation phase played an important role ensuring that any changes beyond that point did not break critical components of the application. Failing integration tests helped prevent deploying broken code on multiple occasions, and will continue to serve this purpose if any future changes are made.

This project managed to produce a web-app that meets the objectives outlined for the *ProvViz* application. It enables users to modify and visualise PROV documents, without prior knowledge of PROV document syntax. Therefore the project is considered to be successful overall.

Chapter 8

Conclusion & Future Work

8.1 Conclusion

As in most software projects, the devised *ProvViz* application could benefit from additional development time and resources. This could help address some of the limitations outlined throughout the report, and help make some "quality-of-life" improvements throughout the application. It is worth mentioning that the *ProvViz* project's source code is publicly accessible under the MIT license, and could therefore be further developed in an open-source capacity from third-party developers in the future.

Overall, the application produced in this project fulfils a significant role in the PROV tooling ecosystem. It provides intuitive PROV editing and visualisation functionality well-suited for those already familiar with the PROV ontology and beginners. The *ProvViz* application could therefore play a significant role in promoting not only the PROV data model, but data provenance as whole.

8.2 Future Work

Although the *ProvViz* software system meets the goals defined for the project, the scope of development (as in most projects) was limited so that deadlines were met on time. During the development process certain limitations and areas for additional functionality came to light, which could be explored in future work. This includes:

- **Moving Relationships from one *Bundle* to Another**: In a future update to the *ProvViz Visualiser*, relationships could be added to the "Tree View" visualisation so that

they can be moved from one *Bundle* to another just like *Entities*, *Activities* and *Agents*.

- **Exporting Visualisations in Alternate Image Formats**: In a future update to the *ProvViz Visualiser*, functionality for exporting the "Graph View" visualisation as a PNG or JPEG image could be implemented.

- **JavaScript PROV Translation Library**: One of the primary limitations of the devised *ProvViz* system outlined in the design is its reliance on the *ProvToolBox* HTTP API for the PROV document translator. There are a variety of possibilities when it comes to developing a *synchronous* PROV translation library that can be run within a browser by the *ProvViz* application. A native JavaScript library could be developed from scratch that implements translation functionality for the PROV model, similar to the functionality provided by the *prov* Python library[1] or the *ProvToolBox* Java library[2]. A less time-consuming alternative could be to compile the *ProvToolBox* Java library to WebAssembly using a tool such as *JWebAssembly*[3], which could then be imported and called synchronously in the browser. Given JavaScript is the most commonly used programming language (according to the last 8 annual Stack Overflow Developer Surveys[4]), a PROV library that can be called synchronously using *JavaScript* would make a great addition to the existing collection of PROV tools, and would promote the usage of the standard in browser applications and other *Node.js* applications.

- **Integrate Cloud Storage Providers**: Users could link a cloud storage provider such as *DropBox*[5] or *Google Drive*[6] to load and save modified PROV documents directly. This would better integrate *ProvViz* into a user's personal file-storage system, without the need for explicitly uploading and downloading PROV documents in the browser, which would provide a more seamless user experience.

- **PROV-N Tokenising the Text-Editor**: The text-editor component of the *ProvViz* application currently only implements syntax highlighting for the JSON and RDF PROV formats, where existing tokenisation rules could be taken from the *monaco-languages* GitHub repository[7]. Additional tokenisation rules for the prominent PROV format *PROV-N* would serve as a "quality-of-life" improvement for users of the application.

[1] https://pypi.org/project/prov/
[2] https://lucmoreau.github.io/ProvToolbox/
[3] https://github.com/i-net-software/JWebAssembly
[4] https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages
[5] https://dropbox.com
[6] https://drive.google.com
[7] https://github.com/microsoft/monaco-languages

- **Implement a Custom Provenance Graph Layout Engine**: The *ProvViz* application currently makes use of the *GraphViz* layout engine to determine the layout of the "Graph View" visualisation. This brings some previously identified limitations, such as a lack of position editing functionality of nodes in the graph and performance set-backs for large PROV documents. A custom provenance graph layout engine could be devised that takes a PROV document as input alongside other visualisation parameters, and produces an SVG as an output, where position editing functionality is implemented accordingly in an efficient manor. Although deemed beyond the scope of this project, this could provide a more customisable PROV visualisation and could be integrated into the *ProvViz* application.

Chapter 9

Definitions

Key terminology used throughout the report:

1. “W3C” - an abbreviation for the World Wide Web Consortium, the international standards organization for the World Wide Web
2. “NPM” - an abbreviation for Node Package Manager, the package manager for the JavaScript programming language
3. “NPM package” - a package published to the NPM registry
4. “API” - an abbreviation for Application Programming Interface, representing a set of definitions that allow software systems to interact with one another
5. “Browser API” - the API provided by a browser
6. “Web-App” - short for “Web-Application” and equivalent to “Browser Application”, a software application that runs in the browser
7. “Native Application” - a software application that runs on a particular platform or device, typically developed using the APIs provided by the platform/device manufacturer
8. “DOM” - an abbreviation for Document Object Model, the interface that represents HTML and XML documents so that programs can change its document structure, style, and content[1]
9. “GUI” - an abbreviation for Graphical User Interface, and represents the interface the user interacts with

[1] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

10. “WebAssembly” - a compilation target for programming languages that allows them to be executed in environments such as the browser

11. “SVG” - an abbreviation for Scalable Vector Graphics, a vector image format that can be rendered natively in the browser’s DOM

References

[1] Provenance Working Group . The PROV Namespace. 04 2013. https://www.w3.org/ns/prov.

[2] Timothy Lebo, and Satya Sahoo, and Deborah McGuinness . PROV-O: The PROV Ontology. 04 2013. https://www.w3.org/TR/prov-o/.

[3] Trung Dong Huynh, and Michael O. Jewell, and Amir Sezavar Keshavarz, and Danius T. Michaelides, and Huanjia Yang, and Luc Moreau . The PROV-JSON Serialization. 04 2013. https://www.w3.org/Submission/2013/SUBM-prov-json-20130424/.

[4] Yolanda, Gil and James, Cheney and Paul, Groth and Olaf, Hartig and Simon, Miles and Luc, Moreau and Paulo, Pinheiro da Silva . Provenance XG Final Report. 2010. https://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/.

[5] Yolanda, Gil and Simon, Miles . PROV Model Primer. 04 2013. https://www.w3.org/TR/prov-primer/#the-complete-example.

[6] Dan Abramov, Alan Zhang, and Matt Cale. React Testing Overview. 2019. https://reactjs.org/docs/testing.html.

[7] Tawfik Borgi, Nesrine Zoghlami, Mourad Abed, and Naceur Mohamed Saber. *Big Data for Operational Efficiency of Transport and Logistics: A Review*. 07 2017.

[8] Kwok Cheung and Jane Hunter. Provenance Explorer – Customized Provenance Views Using Semantic Inferencing. In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora M. Aroyo, editors, *The Semantic Web - ISWC 2006*, pages 215–227, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[9] David Corbacho. Debouncing and Throttling Explained Through Examples. 04 2016. https://css-tricks.com/debouncing-throttling-explained-examples/.

[10] Dmitry Dragilev. Response Times: The Three Important Limits. 11 2011. https://zurb.com/blog/response-times-the-3-important-limits.

[11] Allan Ebdrup. json-schema-benchmark (draft7), 11 2020. https://github.com/ebdrup/json-schema-benchmark/tree/master/draft7.

[12] Paul Groth and Luc Moreau. PROV-Overview - An Overview of the PROV Family of Documents. 04 2013. https://www.w3.org/TR/prov-overview/.

[13] Dong Huynh. prov Python Library. https://github.com/trungdong/prov.

[14] Troy Kohwalter, Thiago Oliveira, Juliana Freire, Esteban Clua, and Leonardo Murta. Prov Viewer: A Graph-Based Visualization Tool for Interactive Exploration of Provenance Data. In Marta Mattoso and Boris Glavic, editors, *Provenance and Annotation of Data and Processes*, pages 71–82, Cham, 2016. Springer International Publishing.

[15] Luc Moreau. ProvToolbox. http://lucmoreau.github.io/ProvToolbox/.

[16] Luc Moreau and Paul Groth. *Provenance: an Introduction to PROV*. Morgan & Claypool, 2013. http://www.provbook.org/.

[17] Open Provenance Organisation. ProvStore. https://openprovenance.org/store/.

[18] Holger Stitz, S. Luger, Marc Streit, and N. Gehlenborg. AVOCADO: Visualization of Workflow-Derived Data Provenance for Reproducible Biomedical Research. *Computer Graphics Forum*, 35:481–490, 06 2016.