FACULTY
OF SCIENCES

A Three-Part View on Solid Web Forms

Ieben Smessaert
Student number: 01804149

Supervisors: Prof. dr. ir. Ruben Verborgh, Dr. ir. Ruben Taelman
Counsellors: Patrick Hochstenbach, Prof. dr. Pieter Colpaert

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2022-2023

GHENT
UNIVERSITY

Preface

This PDF version of the thesis was generated from the original HTML version at https://thesis.smessie.com. Some functions might not be available in the PDF version due to limitations of the conversion from HTML to PDF.

Samenvatting

We gebruiken allemaal wel eens een webformulier, dat is omdat je deelneemt aan een enquête, of omdat je een of andere aanvraag moet invullen. Wanneer iemand gegevens invoert in een webformulier, doet hij dat m.b.v. een app ontwikkeld voor dat specifieke doel. Wanneer iemand een soortgelijk formulier nodig heeft, zal hij een nieuwe app vanaf nul moeten ontwikkelen, zonder de mogelijkheid om van die bestaande app te beginnen en die aan te passen. Bovendien zullen de gegevens worden opgeslagen op de app's server, en de gebruiker zal er niet meer bij kunnen of ze ergens anders kunnen opslaan.

Solid is een nieuw webdecentralisatie-initiatief dat gebruikers controle wil geven over hun gegevens. Deze thesis probeert deze problemen aan te pakken door een driedelige kijk op Solid Web forms te onderzoeken. Door webforms te decentraliseren en te ontkoppelen wordt de definitie van de form gescheiden van de app, en worden de gegevens zelf, inclusief de beschrijving, gedecentraliseerd opgeslagen. Hiertoe wordt een architectuur voorgesteld met de Solid Web Forms opgesplitst in drie delen: weergave, validatie en redenering. Door een beschrijving te hebben los van de app en een redeneergedeelte dat beschrijft wat er met de gegevens moet gebeuren, wordt decentralisatie bereikt. Meerdere apps werden geïmplementeerd, elk verschillende aspecten van deze architectuur tonend.

Een reasoner app werd gemaakt die toont hoe redeneren kan gebeuren in de browser en vanop afstand m.b.v. een HTTP reasoner server. Daarnaast wordt onderzocht hoe er op een abstraherende manier kan worden geredeneerd, wat gemakkelijk wisselen tussen verschillende reasoners toelaat dankzij de voorgestelde uniforme reasoner interface.
Een to-do app werd ontwikkeld om te tonen hoe een gegevensbron kan worden vertaald. Dit laat zien hoe de gegevens kunnen worden losgekoppeld van de app door schema alignment te gebruiken om ze te vertalen naar het vocabulaire dat de app begrijpt.
De implementatie van een FormGenerator app toont hoe een RDF beschrijving kan worden gedefinieerd m.b.v. de driedelige architectuur. Met de app kan men op declaratieve wijze beschrijven welke elementen de form bevat en wat er moet gebeuren bij indiening. Dit maakt decentralisatie mogelijk, omdat de beschrijving kan worden opgeslagen op een andere server, en de policies die beschrijven wat er met de ingevulde gegevens moet gebeuren, kunnen bepalen dat ze worden opgeslagen op een andere server.
De implementatie van de FormRenderer laat zien hoe schema alignment werkt in een meer algemeen proof-of-concept met formulieren. Bovendien wordt getoond hoe de in de beschrijving gedefinieerde policies kunnen worden uitgevoerd als footprint tasks.
Een geïmplementeerde FormCli app toont hoe de beschrijving kan worden gerenderd in een andere omgeving. Samen met de FormRenderer toont dit hoe het weergavegedeelte onafhankelijk is van de omgeving, omdat dezelfde beschrijving kan worden weergegeven in een tekstgebaseerde terminal, terwijl bij de FormRenderer in een GUI m.b.v. HTML.

Samen met een evaluatie van de gebruikerservaring van de FormGenerator en FormRenderer toont deze thesis aan dat de driedelige architectuur een werkbare oplossing is voor de bovengenoemde problemen. We zullen echter zien dat er nog enkele uitdagingen overblijven, die een uitgangspunt vormen voor toekomstig onderzoek.

Summary

We all use a kind of web form once in a while, that is because you take part in a survey, or maybe because you have to fill in some kind of request. When a user enters data in a web form, it does so by using an application that is developed for that specific purpose. When someone needs a similar form, he will have to develop a new application from scratch, without the possibility to start from that existing one and adapt it to his needs. Furthermore, the data will almost always be stored on the application's server, and the user will not be able to access it again or choose to store it somewhere else.

Solid is a new web decentralization initiative that aims to give users control over their data. This thesis attempts to address these problems by investigating a three-part view on Solid web forms. By decentralizing and decoupling web forms, the definition of how the form should look is separated from the app, and the data itself, inclusively the form description, is stored in a decentralized way. To do this, an architecture is proposed where the Solid Web Forms are split into three parts: display, validation, and reasoning. By having a form description that is detached from the app and a reasoning part that describes what to do with the submitted data, decentralization is achieved. Multiple proof-of-concept apps were implemented all showing different aspects of this architecture.

A reasoner app was built to show how reasoning can be done in the browser and remotely with the use of an HTTP reasoner server. Additionally, this investigates how reasoning can be done abstractly, allowing one to easily switch between different reasoners thanks to the uniform reasoner interface proposed.
A to-do app was built to show how a data resource can be translated into a different language. This shows how the data can be decoupled from the application by using schema alignment to translate the data into the vocabulary that the application understands.
A FormGenerator app was implemented to demonstrate how to define a form description in RDF using the three-part architecture. The app allows one to describe what elements are contained in the form and what should happen in case of submission, both in a declarative way. This allows for decentralization, since the form description can be stored on a different server than the application, and the policies that describe what to do with the submitted data can specify that it be stored on a different server than the application.
The implementation of the FormRenderer shows how schema alignment can be done in a more general proof of concept with forms. Additionally, it shows how the policies defined in the form description can be executed as footprint tasks.
A FormCli app was built to show how the form description can be rendered in a different viewing environment. Together with the FormRenderer, this shows that the display part is independent of the viewing environment as the same form description can be rendered in a text-based terminal while the FormRenderer renders it in a GUI using HTML.

Together with a user-experience evaluation of the FormGenerator and FormRenderer, this thesis shows that the three-part architecture is a viable solution to the problems mentioned above. However, we will see that some challenges remain, providing a starting point for future research.

A Three-Part View on Solid Web Forms

Ieben Smessaert , Prof. dr. ir. Ruben Verborgh , Dr. ir. Ruben Taelman ,
Patrick Hochstenbach , Prof. dr. Pieter Colpaert

ABSTRACT

Web forms are used all the time, but they lack basic, yet important, features such as controllability, reusability, and decentralization. With traditional centralized forms, we cannot decide where to store the submitted data, and we cannot reuse existing forms. To solve this, we need to describe the form and its actions, i.e. what to do with the submitted data, independently of the app, so that we can edit and copy this description. We created a three-part view on Solid Web Forms by decoupling a form description into the display, validation, and reasoning parts. In this paper, we demonstrate how such a declarative form description can be created and used without making assumptions about the viewing environment or data storage. Using a declaratively written form description, we can render a form with our favorite renderer application in any viewing environment and perform the actions described in the form description using reasoning performed through a uniform reasoner interface. This decoupling allows us to reuse forms and store data in a decentralized way. By enabling users to modify form descriptions, we reinstate their authority over the data. The first results of this paper are promising. Further research will have to show how further abstractions may push the need for Linked Data knowledge even further aside.

1. INTRODUCTION

Current web forms are meant to be used against one endpoint (1), often used for one (web) display (2), with one particular workflow in mind (3), without a means to send and receive the data in another way (4). We all use a kind of web form once in a while, that is because you take part in a survey, or maybe because you have to fill in some kind of request. When one need a similar form to one that already exists, they will have to develop a new application from scratch, without the possibility to start from that existing one and adapt it to their needs. Furthermore, the data will almost always be stored on the server of the service provider, and the user will not be able to access it again or choose to store it somewhere else (footprint).

To tackle this problem, this thesis introduces a solution by looking at Solid web forms as a whole of 3 separate parts: *display*, *validation*, and *reasoning*. With the use of Solid [1] and Linked Data, several solutions have already been proposed, but none of them consider web forms as a whole of 3 separate parts, except for the Design Issue by Berners-Lee [2] and the blog post on *Shaping Linked Data apps* by Verborgh [3]. In addition, to fully decouple the description from the application, schema alignment is required to map the description to the vocabulary of the application. This makes it possible to use the same description for different applications, even if they use different vocabularies. This, along with the execution of the footprint tasks, is done using reasoning.

Therefore, we propose an architecture that splits the Solid web forms into 3 parts and implement proof-of-concept applications to show how this can be done in practice. In this paper, the following three research questions are examined:

- How can machines be controlled in a declarative way to create forms for producing RDF in **multiple viewing environments** (such as the web and text-based via a command line)?
- How can machines be controlled in a declarative way to perform **schema alignment and footprint tasks** by the use of reasoning?
- How can an **abstraction** be made to **run**

reasoning in the browser or remotely?

In Section 2, the high-level architecture of the proposed three-part view is discussed after which each research question is answered in Section 3, Section 4, and Section 5. Next, an evaluation is done with the help of a user-experience evaluation in Section 6. Finally, in Section 7, the conclusion is given.

2. HIGH-LEVEL ARCHITECTURE

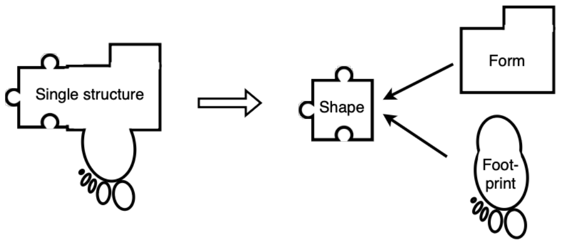


Figure 1: Transition from the traditional single structure where all the data is defined using a single vocabulary, to a three-part view, consisting of a form (for display), shape (for validation), and footprint (for reasoning) part.

A common way how web applications store their data is by using only one fixed structure. Because of that, it is not possible to use the data with another application that uses a different structure. This is even the case for many Solid apps that assume the data is stored in a fixed location in the pod with only one vocabulary. This paper proposes a solution to this problem by using a three-part view on Solid web forms. This shift from a single structure to a three-part view is shown schematically in Figure 1. The left part is the current situation where the data is stored in a single structure and the application is built on top of this structure. The right part is the goal of this research where the data is divided into three parts: a form (for display), shape (for validation), and footprint (for reasoning) part.

The high-level architecture can also be viewed from a different angle. In traditional centralized web applications, different users interact with the same centralized web server using different interfaces. These web interfaces are written for that server and only work for that single web server. Additionally, the data is stored on the server of the application, outside the user's control. The Solid protocol [4] provides a standardized interface, but still many apps are being built with assumptions about the data that is stored in the pod. The app is designed for one specific use case and the data is most of the time stored in a specific way.

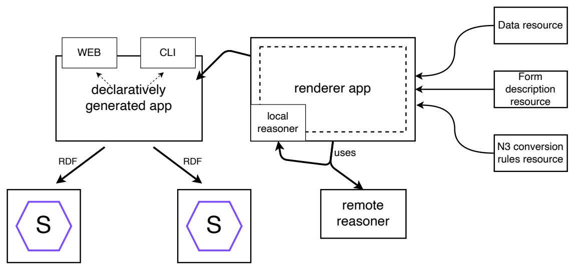
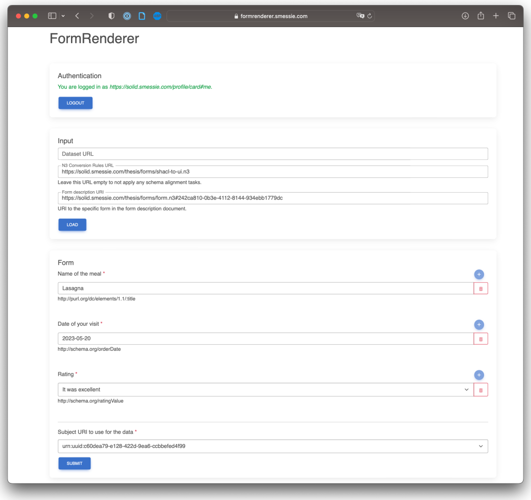


Figure 2: Users interact with a dynamically generated app built by a form renderer using the 3 inputs displayed on the right. This generic renderer app can build for multiple viewing environments without making assumptions about the interface and app itself. It uses a reasoner to apply the schema alignment and footprint tasks. The user can use the generated app to interact with one or more Solid pods.

This paper pushes this decentralized architecture a step further with the introduction of a declarative Solid app that makes no assumptions about the interface and app itself. The previous problem of needing a separate app for each use case is solved by describing the user interface in a declarative way: the *form description resource*. A schematic overview of the architecture is shown in Figure 2. The app still needs to understand the ontology of the form description. This problem is overcome with the use of *schema alignment tasks* translating it into an ontology the app understands. The third input as shown in Figure 2, the *N3 conversion rules resource*, is used by the renderer app to perform this mapping. Data stored in the provided *data resource* can be used to prefill the form. Next, reasoning is also used to apply *footprint tasks*: the execution of policies when a certain action occurs, such as submission. A remote or local reasoner can be used to perform these tasks. Lastly, no assumptions should be made about the app itself or the interface used to interact with this app. This declaratively generated app built by the renderer app can then be used to interact with one or more Solid pods. This concept of having a display part that is unrelated to the viewing environment is discussed in Section 3.

3. MULTIPLE VIEWING ENVIRONMENTS


formrenderer.smessie.com

FormRenderer

Authentication

You are logged in as *https://solid.smessie.com/profile/card#me.*

Input

Dataset URL

N3 Conversion Rules URL: https://solid.smessie.com/thesis/forms/shacl-to-ui.n3

Leave this URL empty to not apply any schema alignment tasks.

Form description URI: https://solid.smessie.com/thesis/forms/form.n3#242ca810-0b3e-4112-8144-934ebb1779dc

URI to the specific form in the form description document.

Form

Name of the meal *: Lasagna

http://purl.org/dc/elements/1.1/:title

Date of your visit *: 2023-05-20

http://schema.org/orderDate

Rating *: It was excellent

http://schema.org/ratingValue

Subject URI to use for the data *: urn:uuid:c60dea79-e128-422d-9ea6-ccbbefed4f99

Figure 3: Screenshot of the implemented FormRenderer.

The form description will provide the decoupling of the three parts: display, validation, and reasoning. The display part is the part that is responsible for rendering the form to the user. There already exist ontologies that can be used for this purpose, such as SHACL [5], Solid-UI [6], and RDF-Form [7].

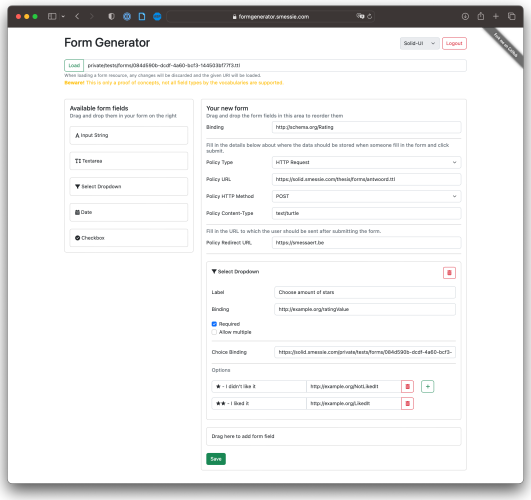
By declaratively describing the form in RDF, it should be possible to render the form in any environment. Web forms are typically HTML, while RDF represents the semantics of the form, not how you represent it in HTML. To prove that the display part is unrelated to the viewing environment, two proof-of-concept applications are implemented that can render the same form description in multiple viewing environments. The first app is the *FormRenderer* which renders the form description in a web browser using HTML. A screenshot of this app is shown in Figure 3. The second app is the *FormCli* which renders the form description in a text-based command-line interface. The architecture and implementation of these apps are very similar to each other. The main difference is that the FormCli app does not have a graphical user interface and uses a text-based terminal instead.

The implementation of these apps provides us with proof that the display part is unrelated to the viewing environment as the same form description can be rendered with the two apps. Everything about how to render the form can be derived from the RDF form description, making it declarative. By making form descriptions portable and not tight to one rendering environment or one rendering logic, machines can be controlled to create forms for producing RDF in multiple viewing environments.

4. SCHEMA ALIGNMENT AND FOOTPRINT TASKS

Unfortunately, the move to decentralization and decoupling comes with its own challenges. Two main challenges need to be tackled before this can be achieved. First, decoupling also means that another app should be able to use or generate the form description. The assumption can however not be made that all apps will use the same ontology to describe similar concepts. To achieve a real decoupled solution, we need to be able to translate from one ontology to another. Therefore, *schema alignment tasks* are introduced, functioning as a mapping to translate from one ontology to another ontology understood by the app. This is implemented using reasoning by using Notation3 (N3) rules [8] that define how to translate one piece of data to another. These rules are collected in a *N3 conversion rules resource*. This way, the form renderer can understand any vocabulary that is passed to it as long as there is a dictionary that maps it to the base vocabulary.

In addition to describing how the form should look, the form description should also declaratively describe what should happen in certain events such as submission. Therefore, the form description is extended with *policies*. The process of executing these policies is called the *footprint tasks* and is the second half of the reasoning part of the three-part view. To describe policies, two languages are needed: a *rule language* and a *policy language* describing what actually should happen when a policy is executed. As rule language, N3 [8] is used. This is the same language that is used to describe the conversion rules in the schema alignment tasks and their N3 rules do exactly what is needed. To describe the policy, a basic version of the FnO ontology [9] is used.



private/tests/forms/084d590b-dcdf-4a60-bcf3-144503bf77f3.ttl

When loading a form resource, any changes will be discarded and the given URI will be loaded.

Beware! This is only a proof of concepts, not all field types by the vocabularies are supported.

Drag and drop them in your form on the right

Drag and drop the form fields in this area to reorder them

Binding: http://schema.org/Rating

Fill in the details below about where the data should be stored when someone fill in the form and click submit.

Policy Type: HTTP Request

Policy URL: https://solid.smessie.com/thesis/forms/antwoord.ttl

Policy HTTP Method: POST

Policy Content-Type: text/turtle

Fill in the URL to which the user should be sent after submitting the form.

Policy Redirect URL: https://smessaert.be

Label: Choose amount of stars

Binding: http://example.org/ratingValue

[✓ Required]
Allow multiple

Choice Binding: https://solid.smessie.com/private/tests/forms/084d590b-dcdf-4a60-bcf3-

★ - I didn't like it	http://example.org/NotLikedIt
★★ - I liked it	http://example.org/LikedIt

Drag here to add form field

Figure 4: Screenshot of the implemented FormGenerator.

The *FormGenerator* app is implemented allowing one to declaratively define a form description by using drag-and-drop to build a form and letting them input the policy properties such as the redirect URL or the details for the HTTP request. A screenshot of this FormGenerator app is shown in Figure 4. A to-do app is implemented providing a first use case for the schema alignment tasks and allowing a simple introduction to the concept. This app also demonstrates the need for policies, as schema alignment tasks fall short in the event of a to-do status toggle. If the app vocabulary should only insert a triple on the occurrence of an event, but the dataset vocabulary requires both an insert and a delete, schema alignment tasks cannot properly satisfy this requirement because according to the app vocabulary, there are no triples to delete, i.e., there are no triples to use in the rule premise. This is also fixed by using policies, where the policy defines which triples should be inserted and deleted in case of a toggle to-do status event. Finally, the form renderer apps are extended with these new schema alignment and footprint tasks.

5. UNIFORM REASONER INTERFACE

The second challenge that was mentioned in Section 4 is that no single use case is the same. Some reasoning steps may be computationally intensive, while others are not, but need to be done fast without many dependencies. This leads to the idea that we should be able to dynamically change how we execute the reasoning without a lot of work, based on the exact use case at that moment. A *uniform reasoner interface* is designed to abstract away the differences between the different reasoners, allowing one to easily switch between them. Switching between reasoners can mean switching between reasoning in the browser or remotely, or it can mean switching between reasoner implementations to improve performance.

First, the `data` and `query` parameters are needed. The `data` parameter is used to pass the data to the reasoner together with any inference rules that should be applied. The `query` parameter is optional and defines the pattern of the data that should be returned by the reasoner. When passed as a string, the data should be formatted in the Notation3 syntax. Furthermore, the interface is designed with extensibility in mind by using a single object that contains all the additional options. This object can be extended by other reasoners, allowing them to add their options. By default, the output type should be the same as the input type. However, by passing the `outputType` option, the user can specify the output type. This option must support at least the `string` value, it should also support the `quads` value, which will return the output as an array of RDF/JS Quads. When the query parameter is left undefined, the user should have the option to execute implicit queries. This is expressed in the options object by the `output` option by defining what to output with implicit queries. Last, the option `blogic` can be defined to use blogic [10], used to support RDF Surfaces [11].

The proposed interface is implemented in the client-side EYE-JS [12] reasoner package. Furthermore, an `eye-mock` library [13] is implemented with the same interface allowing one to execute the reasoning on a remote server. Finally, a Reasoner app is implemented to demonstrate the use of the interface and to allow one to easily switch between the different reasoners. The user can use the toggle to switch easily between the two implementations to fit their needs. Under the hood, this is done by just changing the import statement of the reasoner package.

6. EVALUATION

The proposed architectures and implementations are evaluated by doing a user study. The user experience is evaluated by asking participants to use the FormGenerator and FormRenderer apps to create and fill out forms. Users were provided with a scenario explaining what they were supposed to do with the app.

The feedback received from these users was that the FormGenerator app was easy to use, especially because of the drag-and-drop functionality. However, the feedback also showed that bindings and other Linked Data concepts still confuse users. This is still required knowledge to use the app, which should not be the case. The users building a SHACL form noted that the "min count" and "max count" for a field were confusing to them because they did not know what they meant. Overall, the feedback on the FormGenerator app was positive and 6 out of the 8 technically proficient users were able to create the form without any issues besides the difficulties with the bindings.

The feedback that was received for the FormRenderer app from all the 11 users, with and without a technical background, was that the app is straightforward to use, easy to use, and clear. They were all able to fill out the form without any issues. Someone noticed that when filling out a form described using SHACL, they expected a multi-line text field to be used for the review field instead of a single-line text field. However, the SHACL vocabulary does not allow one to define a multi-line text field. Furthermore, one person noted that it was unclear what the Subject URI was for, and even though for people without that knowledge there is always at least one valid suggestion that can be used, it can be confusing because they do not know what to choose. Besides that, the users did not notice that the app was using Solid and Linked Data behind the scenes and this is exactly the goal of the FormRenderer app. People were also unaware that schema alignment tasks were being performed behind the scenes.

7. CONCLUSION

This paper demonstrates a three-part view on Solid web forms. Our first 2 contributions, the implementations of the FormRenderer and FormCli prove that the display part is not tight to one rendering environment. With our third contribution, the FormGenerator, we show how such declarative form descriptions can be produced, answering the first research question. The user evaluation made clear the SHACL ontology is not ideal for the display part. It is now empirically shown that the Solid-UI ontology is more natural for the display part.

Furthermore, with our fourth contribution, schema alignment and footprint tasks were successfully introduced allowing one to use different vocabularies than the app understands and allowing one to execute declaratively defined policies on the occurrence of events. This successfully answers the second research question. Nonetheless, the results of these applications show the need for further research to further improve the perceived accessibility issues regarding bindings in order to make these technologies optimally available to all people without expecting them to have prior technical knowledge. After all, the user evaluation showed that the FormGenerator and FormRenderer apps are for different types of people. Future research on how bindings could be automatically suggested to the user could be a solution to this problem. Additionally, a standardized way of defining policies would be interesting for future work.

Finally, our fifth contribution, the uniform reasoner interface, was introduced to allow one to easily switch between different reasoning implementations. With our sixth contribution, the implementation of the Reasoner app, we show how this abstraction can be used to run reasoning in the browser or remotely, answering the third and final research question. As future work, it would be interesting to see this interface implemented in other reasoning libraries, especially in a library that implements a different algorithm than the EYE reasoner. An HTTP server version of this interface with the same parameters would also be interesting for future work.

BIBLIOGRAPHY

[1] T. Berners-Lee and others, “Solid.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://solidproject.org

[2] T. Berners-Lee, “Linked Data Shapes, Forms and Footprints - Design Issues.” Apr. 2019. Accessed: Apr. 15, 2023. [Online]. Available: https://www.w3.org/DesignIssues/Footprints.html

[3] R. Verborgh, “Shaping Linked Data apps.” 2022. Accessed: Dec. 01, 2022. [Online]. Available: https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/

[4] S. Capadisli, T. Berners-Lee, R. Verborgh, and K. Kjernsmo, “Solid Protocol.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://solidproject.org/TR/protocol

[5] H. Knublauch and D. Kontokostas, “Shapes Constraint Language (SHACL).” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/TR/shacl/

[6] SolidOS, “Solid-UI.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/ns/ui#

[7] D. Beeke, “RDF Form.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://rdf-form.danielbeeke.nl

[8] D. Arndt, W. Van Woensel, D. Tomaszuk, and G. Kellogg, “Notation3.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://w3c.github.io/N3/spec/

[9] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, “Implementation-independent function reuse,” *Future Generation Computer Systems*, vol. 110, pp. 946–959, 2020.

[10] P. Hayes, “BLOGIC. (ISWC 2009 Invited Talk).” Oct. 2009. Accessed: May 12, 2023. [Online]. Available: https://www.slideshare.net/PatHayes/blogic-iswc-2009-invited-talk

[11] P. Hochstenbach and J. De Roo, “RDF Surfaces Primer.” 2023. Accessed: Apr. 06, 2023. [Online]. Available: https://w3c-cg.github.io/rdfsurfaces/

[12] W. Jesse and J. De Roo, “EYE JS.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/eyereasoner/eye-js

[13] I. Smessaert, “eye-mock.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://github.com/smessie/eye-mock

Een driedelige kijk op Solid Web Forms

Ieben Smessaert , Prof. dr. ir. Ruben Verborgh , Dr. ir. Ruben Taelman ,
Patrick Hochstenbach , Prof. dr. Pieter Colpaert

ABSTRACT

Webforms worden voortdurend gebruikt, toch missen ze elementaire, maar belangrijke functies zoals controleerbaarheid, herbruikbaarheid en decentralisatie. Met traditionele gecentraliseerde forms kunnen we niet beslissen waar we de ingevoerde gegevens opslaan, en kunnen we bestaande forms niet hergebruiken. Om dit op te lossen moeten we de form en zijn acties, dat wil zeggen wat er met de input moet gebeuren, onafhankelijk van de app beschrijven, zodat we deze kunnen bewerken en kopiëren. In deze paper geven we een driedelige kijk op Solid Web Forms door de formbeschrijving te ontkoppelen in een weergave-, validatie- en redeneergedeelte. In deze paper demonstreren we hoe zo'n declaratieve beschrijving kan worden gemaakt en gebruikt zonder aannames te maken over de weergaveomgeving of gegevensopslag. Met behulp van een declaratieve formbeschrijving kunnen we een form renderen met onze favoriete renderapp in elke weergaveomgeving en de beschreven acties uitvoeren met behulp van redenering (reasoning) via een uniforme reasoner interface. Door deze ontkoppeling kunnen we forms hergebruiken en gegevens decentraal opslaan. De gebruiker krijgt de mogelijkheid beschrijvingen te bewerken en krijgt zo terug controle over zijn gegevens. De eerste resultaten van deze paper zijn veelbelovend. Verder onderzoek zal moeten uitwijzen hoe verdere abstracties de behoefte aan Linked Data-kennis nog verder kunnen doen afnemen.

1. INTRODUCTIE

De huidige webforms zijn bedoeld om gebruikt te worden tegen één endpoint (1), typisch voor één (web)weergave (2), met één bepaalde workflow in gedachten (3), zonder een middel om de gegevens op een andere manier te verzenden en te ontvangen (4). We gebruiken allemaal wel eens een soort webform, dat is omdat je deelneemt aan een enquête, of misschien omdat je een of andere aanvraag moet invullen. Wanneer je een soortgelijke form nodig hebt als een reeds bestaande form, zal je een nieuwe app vanaf nul moeten ontwikkelen, zonder de mogelijkheid om van die bestaande form te beginnen en het aan jouw behoeften aan te passen. Bovendien zal de input bijna altijd worden opgeslagen op de server van de serviceprovider, en de gebruiker zal er niet meer bij kunnen of ze ergens anders kunnen opslaan (footprint).

Om dit probleem aan te pakken, wordt in deze paper een oplossing voorgesteld door Solid web forms te beschouwen als een geheel van 3 afzonderlijke onderdelen: *display*, *validation*, en *reasoning*. Met het gebruik van Solid [1] en Linked Data zijn al verschillende oplossingen voorgesteld, maar geen daarvan beschouwt webforms als een geheel van 3 afzonderlijke delen, behalve de Design Issue van Berners-Lee [2] en de blogpost over *Shaping Linked Data apps* van Verborgh [3]. Om de beschrijving volledig los te koppelen van de app, is bovendien schema alignment nodig om de beschrijving te mappen naar het vocabulaire van de app. Dit maakt het mogelijk om dezelfde beschrijving te gebruiken voor verschillende apps, zelfs als deze verschillende vocabulaires gebruiken. Dit gebeurt, samen met de uitvoering van de footprint-taken, met behulp van redenering.

Wij stellen een architectuur voor de opsplitsing van web forms voor in de drie delen en passen deze architectuur toe op proof-of-concept apps. In deze paper worden de volgende drie onderzoeksvragen onderzocht:

- Hoe kunnen machines op een declaratieve manier worden aangestuurd om forms te maken voor het produceren van RDF in

meerdere weergaveomgevingen (zoals het web en tekstgebaseerd via een commandoregel)?

- Hoe kunnen machines op declaratieve wijze worden aangestuurd om **schema alignment en footprint taken** uit te voeren door gebruik te maken van redenering?
- Hoe kan een **abstractie** worden gemaakt om **te redeneren** in de browser of op afstand?

In Section 2 wordt de high-level architectuur van de voorgestelde driedeling besproken, waarna elke onderzoeksvraag wordt beantwoord in Section 3, Section 4 en Section 5. Vervolgens wordt een evaluatie uitgevoerd met behulp van een gebruikerservaringsevaluatie in Section 6. Ten slotte wordt in Section 7 de conclusie gegeven.

2. HIGH-LEVEL ARCHITECTUUR

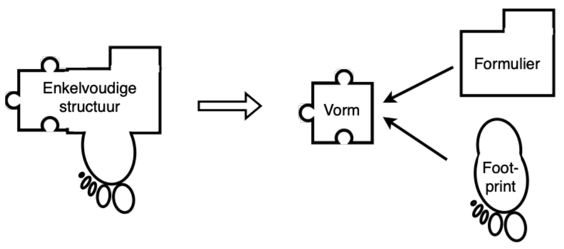


Figure 1: Overgang van de traditionele enkelvoudige structuur waarbij alle gegevens worden gedefinieerd a.d.h.v. een enkel vocabulaire, naar een driedelige kijk, bestaande uit een formulier- (voor weergave), vorm- (voor validatie) en footprintgedeelte (voor redenering).

Een gebruikelijke manier waarop webapps hun gegevens opslaan is door slechts één vaste structuur te gebruiken. Daardoor is het niet mogelijk om de gegevens te gebruiken met een andere app die een andere structuur gebruikt. Dit is zelfs het geval voor veel Solid apps die ervan uitgaan dat de gegevens worden opgeslagen op een vaste plaats in de pod met slechts één vocabulaire. In deze paper wordt deze enkelvoudige structuur vervangen door een driedeling zoals aangegeven in Figure 1. Het linker deel stelt de huidige situatie voor waarbij de gegevens worden opgeslagen in een enkele structuur en de app bovenop deze structuur wordt gebouwd. Het rechterdeel stelt het doel van dit onderzoek voor waarbij de gegevens zijn opgedeeld in drie delen: formulier- (voor weergave), vorm- (voor validatie) en footprintgedeelte (voor redenering).

De high-level architectuur kan ook vanuit een andere hoek worden bekeken. In traditionele gecentraliseerde webapps communiceren verschillende gebruikers met dezelfde gecentraliseerde webserver via verschillende interfaces. Deze webinterfaces zijn geschreven voor die server en werken alleen voor die ene webserver. Bovendien worden de gegevens opgeslagen op de server van de app, buiten de controle van de gebruiker. Het Solid-protocol [4] biedt een gestandaardiseerde interface, maar toch worden veel apps gebouwd met aannames over de gegevens die in de pod worden opgeslagen. De app is ontworpen voor één specifieke use case en de gegevens worden meestal op een specifieke manier opgeslagen.

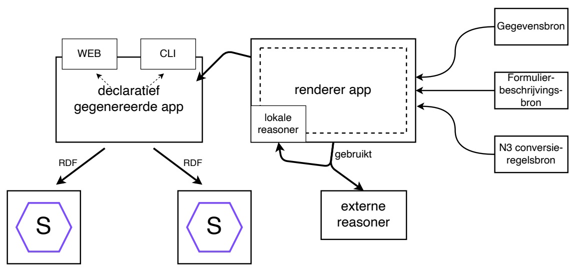
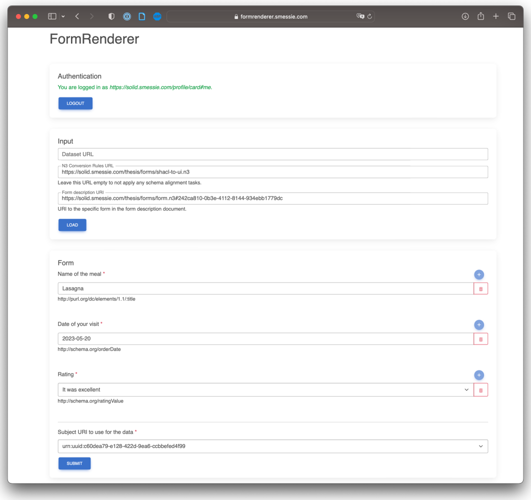


Figure 2: Men interageert met een dynamisch gegenereerde app, gebouwd door een form renderer, met behulp van de 3 inputs aan de rechterkant. Deze generieke renderer app kan voor meerdere weergaveomgevingen bouwen zonder aannames te doen over de interface en de app zelf. Het gebruikt een reasoner om de schema alignment en footprint taken toe te passen. De gebruiker kan de gegenereerde app gebruiken voor interactie met een of meer Solid pods.

Deze paper duwt deze gedecentraliseerde architectuur een stap verder met de introductie van een declaratieve Solid app die geen aannames maakt over de interface en de app zelf. Het eerdere probleem dat voor elke use case een aparte app nodig was, wordt opgelost door de gebruikersinterface op een declaratieve manier te beschrijven: de *formulier-beschrijvingsbron*. Een schematisch overzicht van de architectuur wordt getoond in Figure 2. De app moet nog steeds de ontologie van de beschrijving begrijpen. Dit probleem wordt overwonnen met behulp van *schema alignment taken* die het vertalen naar een ontologie die de app begrijpt. De derde input zoals getoond in Figure 2, de *N3 conversieregelsbron*, wordt gebruikt door de renderer app om deze mapping uit te voeren. Gegevens opgeslagen in de opgegeven *gegevensbron* kunnen worden gebruikt om het formulier vooraf al in te vullen. Vervolgens wordt redenering ook gebruikt om *footprint taken* toe te passen: het uitvoeren van policies wanneer een bepaalde actie plaatsvindt, zoals indienen. Een externe of lokale reasoner kan worden gebruikt om deze taken uit te voeren. Ten slotte mogen geen aannames worden gedaan over de app zelf of de interface die wordt gebruikt voor interactie met de app. Deze declaratief gegenereerde app, gebouwd door de renderer app, kan dan worden gebruikt voor interactie met een of meer Solid pods. Dit concept van een weergavegedeelte dat los staat van de kijkomgeving wordt besproken in Section 3.

3. MEERDERE KIJKOMGEVINGEN


formrenderer.smessie.com

FormRenderer

Authentication

You are logged in as *https://solid.smessie.com/profile/card#me.*

LOGOUT

Input

Dataset URL

N3 Conversion Rules URL: https://solid.smessie.com/thesis/forms/shacl-to-ui.n3

Leave this URL empty to not apply any schema alignment tasks.

Form description URI: https://solid.smessie.com/thesis/forms/form.n3#242ca810-0b3e-4112-8144-934ebb1779dc

URI to the specific form in the form description document.

LOAD

Form

Name of the meal *

Lasagna

http://purl.org/dc/elements/1.1/:title

Date of your visit *

2023-05-20

http://schema.org/orderDate

Rating *

It was excellent

http://schema.org/ratingValue

Subject URI to use for the data *

urn:uuid:c60dea79-e128-422d-9ea6-ccbbefed4f99

SUBMIT

Figure 3: Screenshot van de geïmplementeerde FormRenderer.

De formulierbeschrijving zorgt voor de loskoppeling van de drie delen: weergave, validatie en redenering. Het weergavegedeelte is het gedeelte dat instaat voor het weergeven van de form. Hiervoor bestaan al ontologieën die gebruikt kunnen worden, zoals SHACL [5], Solid-UI [6], en RDF-Form [7].

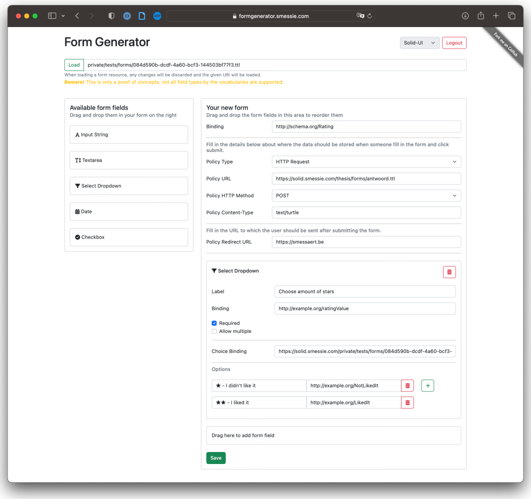
Door de form declaratief te beschrijven in RDF, moet het mogelijk zijn de form in elke omgeving te renderen. Webforms zijn typisch HTML, terwijl RDF de semantiek van de form weergeeft, niet hoe je het in HTML weergeeft. Om te bewijzen dat het weergave-gedeelte los staat van de weergave-omgeving, zijn twee proof-of-concept apps geïmplementeerd die eenzelfde beschrijving in meerdere weergave-omgevingen weergeven. De eerste app is de *FormRenderer* die de form in een webbrowser weergeeft met behulp van HTML. Een screenshot van deze app staat in Figure 3. De tweede app is de *FormCli* die de form weergeeft in een tekstgebaseerde commandoregel-interface. De architectuur en implementatie van deze apps lijken erg op elkaar. Het belangrijkste verschil is dat de FormCli app geen GUI heeft, maar een tekstgebaseerde terminal gebruikt.

De implementatie van deze apps levert het bewijs dat het weergavegedeelte losstaat van de weergaveomgeving, aangezien dezelfde beschrijving kan worden weergegeven met beide apps. De RDF-beschrijving bevat alles over de weergave van de form, wat het declaratief maakt. Door beschrijvingen overdraagbaar te maken en niet gebonden aan één renderomgeving of één renderlogica, kunnen machines worden aangestuurd om forms te maken die RDF produceren in meerdere weergaveomgevingen.

4. SCHEMA ALIGNMENT EN FOOTPRINT TAKEN

Helaas brengt de overgang naar decentralisatie en ontkoppeling zijn eigen uitdagingen met zich mee. Twee belangrijke uitdagingen moeten worden aangepakt voordat dit kan worden bereikt. Ten eerste betekent loskoppeling ook dat een andere app de beschrijving moet kunnen gebruiken of genereren. Er kan echter niet van worden uitgegaan dat alle apps dezelfde ontologie zullen gebruiken om soortgelijke concepten te beschrijven. Om tot een echte ontkoppeling te komen, moeten we kunnen vertalen van de ene ontologie naar de andere. Daarom worden *schema alignment taken* geïntroduceerd, die functioneren als een mapping om van de ene ontologie te vertalen naar een andere ontologie die de app begrijpt. Dit wordt geïmplementeerd door te redeneren met behulp van Notation3 (N3) rules [8] die bepalen hoe het ene gegeven moet worden vertaald naar het andere. Deze regels worden verzameld in een *N3 conversieregelsbron*. Op deze manier kan de formrenderer elke taal begrijpen dat hem wordt doorgegeven, zolang er een mapping bestaat naar de basistaal.

Naast het beschrijven van hoe de form eruit moet zien, moet de beschrijving ook declaratief beschrijven wat er moet gebeuren bij bepaalde gebeurtenissen zoals indiening. Daarom wordt de beschrijving uitgebreid met *policies*. Het proces van het uitvoeren van deze policies wordt de *footprint taken* genoemd en vormt de tweede helft van het redeneergedeelte van de driedelige kijk. Om policies te beschrijven zijn twee talen nodig: een *regeltaal* en een *policy taal* die beschrijft wat er eigenlijk moet gebeuren als een policy wordt uitgevoerd. Als regeltaal wordt N3 [8] gebruikt. Dit is dezelfde taal die wordt gebruikt om de conversieregels bij schema alignment te beschrijven en hun N3 regels doen precies wat nodig is. Om de policy te beschrijven wordt een basisversie van de FnO ontologie [9] gebruikt.



Form Generator

private/tests/forms/084d590b-dcdf-4a60-bcf3-144503bf77f3.ttl

When loading a form resource, any changes will be discarded and the given URI will be loaded.

Beware! This is only a proof of concepts, not all field types by the vocabularies are supported.

Drag and drop them in your form on the right

Your new form

Drag and drop the form fields in this area to reorder them

Binding: http://schema.org/Rating

Fill in the details below about where the data should be stored when someone fill in the form and click submit.

Policy Type: HTTP Request

Policy URL: https://solid.smessie.com/thesis/forms/antwoord.ttl

Policy HTTP Method: POST

Policy Content-Type: text/turtle

Fill in the URL to which the user should be sent after submitting the form.

Policy Redirect URL: https://smessaert.be

Select Dropdown

Label: Choose amount of stars

Binding: http://example.org/ratingValue

[✓] Required

[ ] Allow multiple

Choice Binding: https://solid.smessie.com/private/tests/forms/084d590b-dcdf-4a60-bcf3-

Options

★ - I didn't like it	http://example.org/NotLikedIt
★★ - I liked it	http://example.org/LikedIt

Drag here to add form field

Figure 4: Screenshot van de geïmplementeerde FormGenerator.

De *FormGenerator* app is geïmplementeerd waarmee men declaratief een beschrijving kan definiëren door met drag-and-drop een form te bouwen en de policy-eigenschappen zoals de redirect URL of de details voor het HTTP-verzoek in te voeren. Een screenshot van deze FormGenerator app wordt getoond in Figure 4. Er is een to-do app geïmplementeerd die een eerste use case biedt voor de schema alignment taken en een eenvoudige introductie tot het concept mogelijk maakt. Deze app demonstreert ook de noodzaak van policies, aangezien schema alignment taken tekort schieten bij een to-do status toggle. Als de app-vocabulaire alleen een triple moet invoegen bij een gebeurtenis, maar de dataset-vocabulaire zowel een invoeging als een verwijdering vereist, kunnen schema alignment-taken niet goed aan deze eis voldoen omdat er volgens de app-vocabulaire geen triples zijn om te verwijderen, dat wil zeggen er zijn geen triples om te gebruiken in de regelpremisse. Dit wordt ook opgelost door het gebruik van policies, waarbij de policy bepaalt welke triples moeten worden ingevoegd en verwijderd in geval van het event. Ten slotte worden de formrenderer apps uitgebreid met deze nieuwe schema alignment en footprint taken.

5. UNIFORME REASONER INTERFACE

De tweede uitdaging vermeld in Section 4 is dat geen enkele use case hetzelfde is. Sommige redeneerstappen kunnen computationeel intensief zijn, terwijl andere dat niet zijn, maar snel moeten worden uitgevoerd zonder veel afhankelijkheden. Dit leidt tot het idee dat we zonder veel werk dynamisch de uitvoering van de redenering willen aanpassen op basis van de precieze use case op dat moment. Een *uniforme reasoner interface* is ontworpen om de verschillen tussen de reasoners weg te abstraheren, wat gemakkelijk switchen toelaat. Dit kan betekenen wisselen tussen redeneren in de browser of op afstand, of wisselen tussen implementaties van reasoners om de prestaties te verbeteren.

Eerst zijn de parameters `data` en `query` nodig. De `data` parameter wordt gebruikt om gegevens door te geven aan de reasoner samen met eventuele toe te passen inferentieregels.

De `query` parameter is optioneel en definieert het patroon van de door de reasoner terug te geven gegevens. Wanneer de gegevens worden doorgegeven als een string, moet dit in de Notation3-syntax gebeuren. Verder is de interface ontworpen met het oog op uitbreidbaarheid, met behulp van een enkel object dat alle extra opties bevat. Dit object kan worden uitgebreid door andere reasoners, zodat zij hun opties kunnen toevoegen. Standaard is het output type hetzelfde als het input type. De gebruiker kan echter het output type specificeren met behulp van de `outputType` optie. Deze optie moet minstens de waarde `string` ondersteunen, maar kan ook de waarde `quads` ondersteunen, die de output teruggeeft als array van RDF/JS Quads. Bij een ongedefinieerde query parameter moet de gebruiker de optie hebben om impliciete queries uit te voeren. Welke impliciete query wordt gedefinieerd met de optie `output`. Tot slot kan de optie `blogic` worden gedefinieerd om blogic [10] te gebruiken, om RDF Surfaces [11] te ondersteunen.

De voorgestelde interface is geïmplementeerd in de client-side EYE-JS [12] package. Verder wordt een `eye-mock` bibliotheek [13] geïmplementeerd met dezelfde interface die de redenering uitvoert op een externe server. Tenslotte wordt een Reasoner app geïmplementeerd om het gebruik van de interface te demonstreren en om gemakkelijk te kunnen wisselen tussen de verschillende reasoners. Met behulp van een toggle kan de gebruiker naargelang zijn behoeften schakelen tussen de twee implementaties. Achter de schermen gebeurt dit door het import statement van de reasoner package aan te passen.

6. EVALUATIE

De voorgestelde architecturen en implementaties worden geëvalueerd door middel van een gebruikersonderzoek. De gebruikerservaring wordt geëvalueerd door deelnemers te vragen de FormGenerator en FormRenderer apps te gebruiken om forms te maken en in te vullen. Ze kregen een scenario voorgelegd waarin werd uitgelegd wat ze met de app moesten doen.

De feedback van deze gebruikers was dat de FormGenerator app gemakkelijk te gebruiken was, vooral door de drag-and-drop functionaliteit. Uit de feedback bleek echter ook dat bindings en andere Linked Data concepten gebruikers nog steeds in verwarring brengen. Dit is nog steeds vereiste kennis om de app te gebruiken, wat niet het geval zou moeten zijn. Zij die een SHACL-formulier maakten, merkten op dat de “min count” en “max count” voor een veld verwarrend waren omdat zij niet wisten wat die betekenden. In het algemeen was de feedback over de FormGenerator app positief en konden 6 van de 8 technisch onderlegde gebruikers het formulier maken zonder problemen, naast de problemen met de bindings.

De feedback die voor de FormRenderer app werd ontvangen van alle 11 gebruikers, met en zonder technische achtergrond, was dat de app rechttoe rechtaan, gemakkelijk te gebruiken en duidelijk is. Ze konden allemaal zonder problemen de form invullen. Iemand merkte op dat hij bij het invullen van een SHACL-gebaseerde form verwachtte dat voor het beoordelingsveld een meerregelig tekstveld zou moeten worden gebruikt in plaats van een eenregelig tekstveld. De SHACL ontologie voorziet deze mogelijkheid echter niet. Verder merkte één persoon op dat het onduidelijk was waarvoor de Subject URI diende, en hoewel er altijd minstens één geldige suggestie is die kan worden gebruikt, kan het verwarrend zijn omdat ze niet weten wat ze moeten kiezen. Daarnaast merkten de gebruikers niet dat de app achter de schermen Solid en Linked Data gebruikte, en dat is precies het doel van de FormRenderer app. Mensen hadden ook niet door dat schema alignment taken werden uitgevoerd.

7. CONCLUSIE

Deze paper demonstreert een driedelige kijk op Solid web forms. Onze eerste 2 contributies, de FormRenderer en FormCli implementaties, bewijzen dat het weergavegedeelte niet gebonden is aan één renderomgeving. Met de derde contributie, de FormGenerator, laten we zien hoe dergelijke declaratieve beschrijvingen kunnen worden geproduceerd, wat de eerste onderzoeksvraag beantwoordt. De gebruikersevaluatie maakte duidelijk dat de SHACL-ontologie niet ideaal is voor het weergavegedeelte. Empirisch blijkt nu dat de Solid-UI ontologie natuurlijker is voor dit deel.

Verder werden met onze vierde contributie schema alignment en footprint taken met succes geïntroduceerd waardoor men andere vocabulaires kan gebruiken dan de app begrijpt en waardoor men declaratief gedefinieerde policies kan uitvoeren bij het voorkomen van events. Hiermee is de tweede onderzoeksvraag met succes beantwoord. Niettemin tonen de resultaten van deze apps de noodzaak naar verder onderzoek om de waargenomen problemen met betrekking tot bindings verder te verbeteren, teneinde deze technologieën optimaal beschikbaar te maken voor alle mensen zonder te verwachten dat zij over voorafgaande technische kennis beschikken. Uit de gebruikersevaluatie bleek immers dat de FormGenerator en FormRenderer voor verschillende soorten mensen bestemd zijn. Toekomstig onderzoek naar hoe bindings automatisch kunnen worden voorgesteld, zou een oplossing voor dit probleem kunnen zijn. Daarnaast zou een gestandaardiseerde definitie van policies interessant zijn voor toekomstig werk.

Tot slot werd onze vijfde contributie, de uniforme reasoner interface, geïntroduceerd om gemakkelijk te kunnen wisselen tussen reasoning implementaties. Met onze zesde contributie, de Reasoner app, laten we zien hoe deze abstractie kan worden gebruikt om redeneringen in de browser of op afstand uit te voeren, waarmee de laatste onderzoeksvraag wordt beantwoord. Als toekomstig werk zou het interessant zijn deze interface te implementeren in andere reasoning bibliotheken, vooral in een bibliotheek die een ander algoritme implementeert dan EYE. Een HTTP-serverversie van deze interface met dezelfde parameters zou ook interessant zijn voor toekomstig werk.

BIBLIOGRAFIE

[1] T. Berners-Lee and others, “Solid.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://solidproject.org

[2] T. Berners-Lee, “Linked Data Shapes, Forms and Footprints - Design Issues.” Apr. 2019. Accessed: Apr. 15, 2023. [Online]. Available: https://www.w3.org/DesignIssues/Footprints.html

[3] R. Verborgh, “Shaping Linked Data apps.” 2022. Accessed: Dec. 01, 2022. [Online]. Available: https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/

[4] S. Capadisli, T. Berners-Lee, R. Verborgh, and K. Kjernsmo, “Solid Protocol.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://solidproject.org/TR/protocol

[5] H. Knublauch and D. Kontokostas, “Shapes Constraint Language (SHACL).” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/TR/shacl/

[6] SolidOS, “Solid-UI.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/ns/ui#

[7] D. Beeke, “RDF Form.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://rdf-form.danielbeeke.nl

[8] D. Arndt, W. Van Woensel, D. Tomaszuk, and G. Kellogg, “Notation3.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://w3c.github.io/N3/spec/

[9] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, “Implementation-independent function reuse,” *Future Generation Computer Systems*, vol. 110, pp. 946–959, 2020.

[10] P. Hayes, “BLOGIC. (ISWC 2009 Invited Talk).” Oct. 2009. Accessed: May 12, 2023. [Online]. Available: https://www.slideshare.net/PatHayes/blogic-iswc-2009-invited-talk

[11] P. Hochstenbach and J. De Roo, “RDF Surfaces Primer.” 2023. Accessed: Apr. 06, 2023. [Online]. Available: https://w3c-cg.github.io/rdfsurfaces/

[12] W. Jesse and J. De Roo, “EYE JS.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/eyereasoner/eye-js

[13] I. Smessaert, “eye-mock.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://github.com/smessie/eye-mock

Acknowledgments

First, I would like to thank my promotors Ruben Verborgh and Ruben Taelman and my counselors Pieter Colpaert and Patrick Hochstenbach for their guidance and support throughout this thesis. Thank you for letting me work on this very interesting topic. I especially want to thank Patrick for all his help, insights, constructive feedback, and the many meetings we had throughout the year. I also want to thank the other people in the lab for the chats in the office, and Jan-Pieter for rubber ducking and working together in the office.

Thanks to the people from Zeus WPI, other friends and my family for participating in the user experience evaluation and giving me useful feedback. Thanks for listening to all my stories about my thesis, Solid, and Linked Data.

I thank Robbe for providing the ScholarMarkdownThesis template that I used to write this thesis, and for helping me fix the problems I encountered while using it.

Next, I would like to thank my parents for all the opportunities they have given me, for the endless support, and for letting me complete this five-year journey through university. Thanks to my dad for proofreading the user experience scenarios.

Last but not least, I want to say thanks to all those who took the time to read the final part of my journey through university.

Permission for Usage

Table of Contents

List of Figures

List of Listings

List of Acronyms

HTTP	Hypertext Transfer Protocol
RDF	Resource Description Framework
SKOS	Simple Knowledge Organization System
WASM	WebAssembly
EYE	Euler Yet another proof Engine
N3	Notation3
HTML	HyperText Markup Language
OWL	Web Ontology Language
SPARQL	SPARQL Protocol And RDF Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language
SHACL	Shapes Constraint Language
API	Application Programming Interface
Turtle	Terse RDF Triple Language
JSON-LD	JavaScript Object Notation for Linked Data
FnO	Function Ontology
ShEx	Shape Expressions
IDP	Identity Provider
RegEx	Regular Expression

List of Ontologies

foaf	Friend of a Friend
schema	Schema.org
sh	Shapes Constraint Language (SHACL)
ui	User Interface
owl	OWL 2
form	RDF-Form
hydra	Hydra Core
solid	Solid Terms
ncal	Nepomuk Calendar
fno	Function Ontology
cal	RDF Calendar
log	Log
st	Shape Trees

Chapter 1: Introduction

We all use a kind of web form once in a while, that is because you take part in a survey, or maybe because you have to fill in some kind of request. However, all of today's forms have the same problem: they are application specific. Current web forms are meant to be used against one endpoint (1), often used for one (web) display (2), with one particular workflow in mind (3), without a means to send and receive the data in another way (4). The first two points show that current web forms are centralized. Additionally, the second point not only shows that they are centralized, but together with the third and fourth points it also shows that they are coupled in the sense that the display and the data are not separated.

A concrete example where we can see this problem in action is as follows. Alice goes to a Google Forms form, she can only use that form to save data to the Google Drive and nowhere else. However, we want Alice to be able to choose for herself where to store the data, e.g. she might want to store it in her Solid pod [1] so she keeps ownership of the data she enters. Forms worldwide have display logic for humans, but not for machines. To fill in a form, you need a human being like Alice to interpret the fields in the form and to know which and how to fill in the fields. Alice can thus not rely on a machine to help her fill in the form with data she for example already entered once before in her Solid pod. However, it could make Alice's life way easier if it can tick the checkbox automatically saying she eats vegetarian while filling in a form to register for an event, based on what she specified once earlier in her own pod. In addition, Bob may have created a form that Alice is very interested in. She wants to reuse it for another workflow, but she needs to tweak the form a bit to fit her needs. Now she is lost because she cannot get the form description to do this. Any information about the data model, the display model, and the reasoning is now all centralized at Google.

To tackle this problem, this thesis introduces one solution by looking at Solid web forms as a whole of 3 separate parts: *display*, *validation*, and *reasoning*. This will address the fact that current web forms are centralized and coupled as described earlier. A solution will be built that is decentralized and decoupled so that the display and the data are separated from each other. A web form definition should consist of a display part defining out of which components the form consists, so how your form should look in terms of elements. As an example, it could say that there should be a text field where the user can fill in a name of a book. Next to that, there should also be a validation part where the shape of the form should be defined. More specifically, it should say e.g. if a certain part is required or not, or if multiple values are allowed and if so, how many. It could define that the password field in the form requires at least 3 capital characters and 2 symbols. Lastly, we have the third part, reasoning. For this, the assertion and logic language Notation3 [2] could be used. By use of reasoning, the machine knows what to do with the data when the submit button is pressed. As an example and to link the three parts together, one can agree that specifying an element as required could happen in both the display and the validation part. We might want to *display* a red asterisk next to the name of our element in the form indicating that this field is required. But at the same time, we also want this required property to be defined in our *validation*, because once a form is being validated, we want to be able to check if the required fields are filled in, but to do so, we need to know which fields are filled in. However, this might lead to contradictions where a field is marked as required in the display part but is not in the validation part. Here is where reasoning comes into play. By introducing an extra reasoning layer, extra logic checks should happen on the data that is submitted. To stick to the same example, that could mean that when a required property is defined in the display part, it should error if that required property is not also defined in the validation part. Different types of relations could exist, e.g. it could be that if it is a property in the display part, then it should be a property in the validation part as well, but it could also be only a property in the validation part.

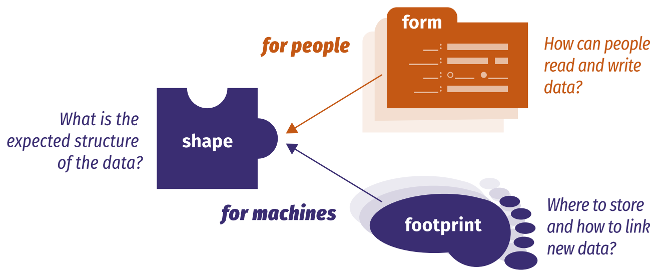

form
for people
How can people read and write data?
What is the expected structure of the data?
shape
for machines
footprint
Where to store and how to link new data?

Figure 1: A shape can have associated forms so people can easily view and edit data, and footprints for determining how new data should be stored [3].

Ideally, the goal is to have this reasoning happening under the hood without extra work required from the app developer. That could enable the app developer to submit only one definition e.g. the display part, after which the validation part is generated using the reasoning part. Unfortunately, the move to decentralization and decoupling comes with its own challenges. Two main challenges need to be tackled before this can be achieved. First, decoupling also means that another app should be able to use the form definition to display the form. The assumption can however not be made that all apps will use the same ontology to describe similar concepts. For example, Alice might have created a form using ontology A, but if Bob wants to render that form using his favorite app that only supports ontology B, he cannot do so. To achieve a real decoupled solution, we need to be able to translate from one ontology to another. The concept of *schema alignment tasks* will be introduced, functioning as a mapping to translate from one ontology to another ontology understood by the app. This could be implemented using reasoning by using rules that define how to translate one piece of data to another, just like how you translate one language to another using a dictionary. Second, no single use case is the same. Some reasoning steps, as were described earlier, might be computationally heavy, while others are not but are required to happen fast without a lot of dependencies. This leads to the idea that we should be able to dynamically change how we execute the reasoning without a lot of work, based on the exact use case at that moment. Therefore, we want to switch between browser-based and remote server-based solutions for reasoning. We even might want to choose another kind of implementation that happens to be more performant for a specific use case. This shows the need for a *uniform reasoner interface* that is the same for all reasoning implementations, such that we can easily switch between them. Verborgh showed in *Shaping Linked Data apps* [3] a similar approach where he split the whole into three parts: shapes, forms, and footprints, and how they relate to each other. This is shown in Figure 1. As the footprint defines where to store the new data corresponding to a shape, this corresponds to the reasoning part as explained above. Although, they do not fully map onto each other as the reasoning part as was earlier proposed consists of both schema alignment and footprint tasks. The footprint part in Verborgh's view is only one of the two parts of the reasoning part of this thesis. The shape defines the validation part, and the form defines the display part.

However, as this is the ultimate goal, it would already be interesting to see if such a split into these three parts is feasible. Because of this, this thesis attempts to 1) be able to generate semantic forms, allowing to make multiple interfaces based on that generated form definition and allowing them to be interpreted by both humans and machines, 2) be able to publish the data model behind the form in a machine-readable form, such that machines can validate these data, and 3) be able to include a reasoning step in the whole, such that a machine knows what to do with the data as soon as the submit button is pressed.

The goal of this thesis is therefore defined as follows: towards the first steps of a **three-part view** on Solid Web Forms by looking into the following three questions:

- How can machines be controlled in a declarative way to create forms for producing RDF in **multiple viewing environments** (such as the web and text-based via a command line)?
- How can machines be controlled in a declarative way to perform **schema alignment and footprint tasks** by the use of reasoning?
- How can an **abstraction** be made to **run reasoning** in the browser or remotely?

To elaborate these goals, appropriate architectures are proposed to tackle each of these questions and then an implementation is provided showing how these architectures can be implemented in practice. This will provide a proof of concept for the proposed architectures. Several proofs of concept will be built, each one becoming more and more complex and building on top of the challenges that have been overcome in the previous ones.

After this introduction chapter, Chapter 2 will first discuss existing approaches and related work. Chapter 3 will give a technical introduction to the technologies that will be used throughout this thesis. The high-level architecture of the proposed idea of a three-part view and how these parts relate to each other will be discussed in Chapter 4. Next, the 3 predetermined research questions will be addressed, each in a separate chapter. Chapter 5 will go into more detail on how multiple viewing environments can be supported with the same produced RDF. Chapter 6 will discuss how reasoning can be used to perform schema alignment and footprint tasks. An abstraction will be made to run reasoning in the browser or remotely, which will be discussed in Chapter 7. Finally, Chapter 8 will conclude this thesis and discuss future work.

Chapter 2: Related Work

The idea of describing a form in RDF is not new. In the literature, there are already some existing approaches that describe a form in RDF. As was already mentioned in the introduction, the idea of this thesis is to split up the form in three separate parts. The first part is the display part and consists of the form definition in Linked Data. To be able to define a form, a vocabulary or ontology is needed that functions as the language. Most of the existing approaches to describe a form in RDF are based on one certain ontology to express what is called the display part in this thesis. In the following, there will be a discussion of some notable ontologies which could be used for the definition of a form.

1. Ontologies

1.1. XForms 2.0

For the reasoning part, a certain language is needed to be able to express what actions should happen at what events. You should e.g. be able to define what should happen with the filled-in data when the user clicks on the submit button. Do we want to perform some checks on it? Do we want to alter the data? Do we want to store it somewhere? All of these actions should be possible to define.

One existing specification found in the literature that seemed to be useful for this problem is XForms 2.0 [4]. This language to define forms is a W3C specification. It is based on XML and also has a split architecture that separates presentation, content, and purpose, this closely relates to the three parts: display, validation, and reasoning which are needed to tackle the problem in this thesis. In XForms 2.0, a model defines what data is needed, then one can have XML data defined concerning the model to become submitted data. Values can be constrained by defining bindings. A bind element can be defined by referring to an XML element which is part of a model. By adding attributes to that bind element, the type can be specified and other constraints can be set. A submission XML element can be added to define the destination of the data. Next to that, the HTTP method can also be specified. However, adding statements allowing to perform actual reasoning is something that does not exist in XForms 2.0. Next to this limited possibility to reason over the data, it is also an XML-based language, and as defining the forms in RDF is desired because that is what is nowadays being used in this domain, it is not a perfect fit for this research. It could however still be useful as a form of inspiration.

1.2. SHACL

The first ontology is SHACL [5], short for Shapes Constraint Language, which is a W3C recommendation. It is a vocabulary to describe and validate RDF graphs against a set of conditions. As the name suggests, SHACL is used to declare shapes, the target of the shape is specified with the `sh:targetClass` property. SHACL can be used for validation, during this validation, the target nodes become focus nodes for the shape. This is a shape-validated RDF term using the triples from a data graph. In other words, the values of the properties and other characteristics of the focus node are validated against the constraints defined in the shape. Node shapes are used to declare constraints on the focus nodes. These constraints can be various kinds of things but are about the focus node itself. Other constraints can be declared as well by the node shape via the `sh:property` property to a property shape. Constraints like `sh:datatype` and `sh:minCount` are being declared by these property shapes. In contrast to node shapes, property shapes are used to declare constraints on the values of the properties of the focus nodes. These constraints can be about the value of the property itself or the value of the property concerning the focus node.

SHACL also provides some non-validating properties that are ignored by the SHACL processors. These properties can be used to provide additional information, e.g. for building a form. Some of those properties that speak for themselves are `sh:name`, `sh:description` and `sh:order`, but also `sh:defaultValue` and `sh:group` can be used to provide additional information. These non-validating properties could be used in the context of a form to define what the form should look like. For example, the `sh:defaultValue` property could be used to define what the default value of a field should be. The `sh:group` property could be used to group fields together in a form. The `sh:order` property could be used to define the order in which the fields should be displayed in the form.

1.3. Solid-UI

Solid-UI is the name for the User Interface widgets and utilities for Solid developed by the SolidOS team. Next to the building blocks for Solid-based apps, there also exists a Solid-UI vocabulary [6]. This vocabulary can be used to define the user interface of a Solid application. It is also called Solid-UI Forms, reflecting the fact that it is used to define forms or to render WebApp front-end elements. In this section, the Solid-UI vocabulary is discussed.

The Solid-UI vocabulary is mainly focused on the display and rendering part of the front end of forms. Next to that, a crucial part is the fact that it provides a binding to the underlying Linked Data. To do so, there is a `ui:property` predicate that is used to bind a form field to a property of the underlying data. There exist many different kinds of form fields like `ui:SingleLineTextField`, `ui:MultiLineTextField`, `ui:DateField`, `ui:BooleanField`, `ui:Choice`, etc. To define a field using one of these types, a subject must receive the `rdf:type` predicate with the type as the object. Next to that, the `ui:property` predicate must be used to bind the field to a property of the underlying data. Additionally, extra properties can be defined with predicates like `ui:label`, `ui:sequence`, `ui:required`, `ui:multiple`, `ui:maxLength`, `ui:pattern`, etc. to define the behavior of the field. To define a choice or select field, the subject of type `ui:Choice` must receive a `ui:from` predicate with as object an instance of `owl:Class`. Then all instances of that class will be used as options for the select field.

The blog post by Hochstenbach, Wright, and Turdean on *RDF forms for Solid* [7] discusses how the Solid-UI ontology can be used to define forms in RDF. These forms allow users to edit their data in their Solid Pod in a user-friendly way, providing a use case for the Solid-UI ontology. In addition, Smessaert's blog post on *Google Forms but the Solid way* [8] demonstrates how such an RDF form description can be created in a user-friendly way using a drag-and-drop interface. A form description can be created not only in the Solid-UI ontology, but also in the SHACL ontology as well as in the RDF-Form ontology.

1.4. RDF-Form

RDF-Form [9] is another vocabulary that can be used to define forms in RDF. Different from SHACL and Solid-UI, RDF-Form has no way defined to link the fields to the form object. This means that all the fields that are defined in the same resource file as the form are being used for that form. This is a limitation of RDF-Form, but it is also a simplification as it is not needed to define a link between the form and the fields. However, this does not conform to the idea of Linked Data, where everything should be linked to everything else and multiple form definitions can be defined in the same resource file.

RDF-Form has multiple properties defined to function as predicates on form fields. One important predicate is `form:widget` which is used to define what kind of field it is. The value of this predicate is a string which is the name of the field type. Examples are `"string"`, `"group"`, `"textarea"`, `"number"`, `"dropdown"`, `"duration"`, etc. Next to that, there are also predicates like `form:label`, `form:required`, `form:order`, `form:placeholder`, `form:option`, etc. This `form:required` is only one of the few predicates that can be used to add some basic validation to the form. However, more advanced validation is not possible with RDF-Form. Again, a `form:binding` predicate exists to bind the field to a property of the underlying data. This predicate has as its value a URI which is the property that the field is bound to.

1.5. Hydra

Hydra [10] is a vocabulary to describe Web APIs in Linked Data. Its intended use is to describe the server side of the API in a machine-readable way. There already exist some technologies to describe a data model, but with Hydra, the focus is on describing Web APIs to, as a server being, advertise to a client what possible actions are allowed to make changes to the data. By doing this, a client can use this description to talk to the API without the need to hardcode how to talk to the API. The Web API is being defined as an ApiDocumentation class. In this class, the main entry point of the API can be defined. Next to that, it can also hold the supported operations, classes, and properties. As sometimes HTTP status codes are not expressive enough on their own to describe the actual problem, extra information can be given to status codes as well. Something that was not yet possible with RDF, RDF Schema, and OWL was the ability to describe whether an IRI is dereferenceable or if it can only be used as an identifier. With the Resource class, Hydra allows us to describe this. Hydra also has the functionality to mark properties as read-only, write-only, and required. Lastly, it also has numerous other concepts that lean towards the backend description of a Web API, like the PagedCollection which gives extra info about paginated requests.

2. RDForms

RDForms [11] is a technology that allows form-based editing and presentation of RDF. It is an editing framework for RDF that focuses on read-write Linked Data while keeping it as simple as possible for the developer. It does so by providing 2 main components: the *RDForms library* and the *RDForms templates*. The first one is responsible for the parsing, serializing, and manipulation of RDF graphs, while the latter makes sure the right RDF expression is produced and manipulated. These templates can be used for presentation, editing, and validation of RDF data. Although this does the job of providing form-based editing and presentation of RDF, it does not include any reasoning possibilities as was introduced for the idea of this thesis. There is no possibility to apply schema alignment tasks or anything similar to that, nor is it possible to define actions to be performed on the occurrence of certain events. It is just a tool to construct form-based RDF editors with the requirement that RDF must be non-cyclic with a single root node.

3. Validation

3.1. Shape-Validator-Component

There has already been done quite some research on how to validate RDF data. The most common way is to use SHACL. SHACL is a W3C recommendation that allows for describing constraints on RDF data and has been discussed earlier in Subsection 2.1.2. In Slabbinck's thesis about cross-application interoperability [12], data validation using Shape Trees is discussed. Shape Trees [13] are a way to describe the shape of data in a way that is independent of the actual data. This allows one to validate data without having to know the actual data. This is done by using a Shape Tree to describe the shape of the data and then validating the data against that Shape Tree. With the `st:shape` property, a Shape Tree can be linked to a SHACL or ShEx shape. This allows using SHACL or ShEx to validate data against a Shape Tree. Slabbinck also made a shape validator component [14] which is the implementation of such a Shape Tree validator using SHACL shapes. Using this component made for the Community Solid Server, it is possible to validate data against a Shape Tree using SHACL shapes to make sure that all the resources in the containers conform to the SHACL shape. Having this guarantee lets applications assume that this structure in the resources is correct and can be used to build on top of it. Only resources that conform to the constraints of the SHACL shape will be able to be added to the constrained container.

3.2. Rdf-Validate-Shacl

The previously mentioned implementation is a validator that functions on the server side. Next to that, client-side validation is also something that is possible and has been researched before. One such example is the rdf-validate-shacl package by Zazuko [15]. This is a JavaScript package that implements the SHACL specification on top of the RDFJS stack. It works by first initializing the validator with a given shape and then validating a given dataset against that shape. The validator will then return a list of validation results that can be used to determine if the dataset conforms to the shape. This is done in the form of a `ValidationReport` object containing on one hand these results and on the other hand a boolean `conforms` indicating if the dataset conforms to the shape. Everything is done in JavaScript and can thus be used on the client side.

3.3. Shacl-Engine

Another noteworthy implementation of a client-side validator in JavaScript is the *shacl-engine* package by Thomas Bergwinkl [16]. At the time of writing, it is a new implementation of the SHACL specification that focuses on performance. The author discovered a bottleneck in the rdf-validate-shacl package which was discussed in Subsection 2.3.2 and decided to create a new implementation that would be faster. The bottleneck lies in the fact that all properties and values were fetched from the `Dataset` object in the `rdf-validate-shacl` package each time a shape was processed [17]. This implementation eliminates this cost by introducing a compile step. This compile step finds a matching compile function while going through all properties of a shape and then adds the result of that function to the shape. By doing so, validating a dataset can now be done by using the compiled validation functions. A comparison of the performance of the two implementations shows that the `shacl-engine` package is 15 times faster than the `rdf-validate-shacl` package [17].

Chapter 3: Technical Introduction

1. Linked Data

Today's web contains a lot of data. However, all of this data is useless if no one is able to find it. To make data discoverable, we should create a web of data. In 2001, Berners-Lee, Hendler, and Lassila proposed the Semantic Web [18] as the idea to make the web machine-readable. This is done by using RDF to describe the data and SPARQL to query it. Linked Data [19] is a set of best practices to publish and interlink data on the web. The idea was to use RDF to publish data as machine-readable and to link it to other related data. Berners-Lee proposed four rules to publish data as Linked Data. Although he called them rules, they are more like best practices that turn your data into Linked Data. First, URIs should be used as names for things. Second, these URIs should be dereferenceable, meaning that when you visit the URI, you should get back data about the thing. Third, the data should be published in a standard format, such as RDF. Finally, the data should be linked to other data, so that you can discover more things.

2. RDF

RDF [20], short for Resource Description Framework, is a framework for describing resources on the web. Where information is often only available for humans to read, RDF makes it possible to describe information in a way that is also machine-readable. This is done by describing resources as a set of triples, where each triple consists of a subject, a predicate, and an object. This is also called an RDF statement and is written as `<subject> <predicate> <object>`. The subject is the resource that is described, the predicate is the property of the resource, and the object is the value of the property. For example, the sentence "Alice is 1.65 meters tall" can be described in RDF as `<Alice> <height> "1.65"^^<http://www.w3.org/2001/XMLSchema#double>`. In this example, the subject is `Alice`, the predicate is `height`, and the object is `"1.65"^^<http://www.w3.org/2001/XMLSchema#double>`. The object is a literal, meaning that it is a value instead of a resource. The literal is a string with the value `1.65` and the datatype `http://www.w3.org/2001/XMLSchema#double`. The datatype is used to indicate the type of the value, in this case, a double. However, the object can also be another resource, for example, the sentence "Alice is the daughter of Bob" can be described in RDF as `<Alice> <daughter> <Bob>`. This creates links between resources, which is the basis of the Linked Data idea. To keep the example simple, no full IRIs are used, but in reality, IRIs are used to identify resources and predicates. One can look up the predicate to find out what it means, for example, the predicate `height` can be looked up to find out that it is a property of a person that describes the height of that person. Vocabularies are often used in RDF, they are a set of predicates and classes that can be used for resource description. The advantage of such vocabularies is that the description of certain concepts can be easily reused. This opens the door to more qualitative definitions of concepts and improves data interoperability.

There exist multiple serializations of RDF, the most common ones are Turtle [21] and

JSON-LD [22]. Turtle is a textual syntax for RDF allowing us to write RDF graphs more compactly and in natural text form. Turtle allows the use of prefixes to shorten IRIs, for example, the predicate `http://example.org/height` can be written as `ex:height` if the prefix `ex` is defined as `http://example.org/`. One can define the prefix by writing `@prefix ex: <http://example.org/> .` at the top of the document. JSON-LD is a JSON-based serialization of RDF that allows embedding RDF in JSON documents. There are also other serializations, such as RDF/XML [23], RDFa [24], and N-Triples [25]. These are all different ways of writing the same triple, so they are logically equivalent.

3. SPARQL

SPARQL [26], short for SPARQL Protocol and RDF Query Language, is a query and manipulation language for RDF. Only SPARQL Query will be used in this thesis, as SPARQL Update is no longer part of the Solid specification [27] as a way to modify resources. SPARQL Query is a language that is used to query RDF graphs. It is a declarative language, meaning that you describe what you want to get back, not how it should be done. Different types of queries can be written in SPARQL, but the most common one is the `SELECT` query. This query is used to get back a set of bindings that match the query. Other types of queries are `CONSTRUCT` and `ASK`. `CONSTRUCT` queries are used to get back a new RDF graph that matches the query. `ASK` queries are used to check if the query matches the RDF graph and will thus return a boolean. There are also `DESCRIBE` queries that return a single result RDF graph containing RDF data about resources.

A simple `SELECT` query is shown in Listing 1. It is a query that selects the height of Alice. The query starts with the `SELECT` keyword, followed by the variables that should be returned. In this case, the variable `?height` is returned. The `WHERE` keyword is used to define the pattern that should be matched. Each variable is always preceded by a question mark.

```
@prefix ex: <http://example.org/>.

SELECT ?height
WHERE {
    ex:Alice ex:height ?height .
}
```

Listing 1: Simple example of SELECT SPARQL query that selects the height of Alice.

4. Solid

Solid [1] is a web decentralization initiative that aims to give people back control over their data. It is a set of specifications [27] that is used to build decentralized applications. The Solid ecosystem is built on top of the Linked Data principles and uses RDF as the data model. Solid introduces the concept of a *Pod*, which is a personal decentralized online data store. People can decide for themselves who has access to what resources in their Pod, both people and applications. These resources can be anything, from simple text files to (complex) structured data or any other regular file that you can find on the web. The aforementioned specifications define interoperable data formats and protocols that are used to interact with Pods. It builds on top of existing web standards, such as HTTP, CORS, and RDF, and uses them to create a decentralized web that focuses on privacy and data ownership. Applications can then read and write data to Pods using the Solid specifications. This allows people to use different applications to interact with their data, without having to worry about data silos [28].

5. Notation3

Another language that could be useful is Notation3 [2]. This language is a logic language that is used to define assertions and rules. It is a language that is used in the Semantic Web and is also used in the Solid ecosystem. As it is a language that is used to define rules and assertions, it is exactly what we want to use for the reasoning part. It looks very promising to use it to define the reasoning part of a form, as it is a logic language, but it is not a language that is designed for this purpose. It builds on top of Turtle meaning that every valid Turtle document is also a valid N3 document, it just adds even more aspects to the language. One of those additional elements is also really important for this research, namely logical implications and variables. This type of extension implements the If-Then style via modus ponens which makes it possible to define the kind of statements as discussed earlier. Next to these logical implications, N3 also allows one to write statements about other statements by quoting them. You could for example easily write the following: `:Alice :says { :Bob :eats :chicken. :Trudy :likes :Bob. }.`. Another big addition to N3 is the set of built-ins that allows via a set of vocabularies to query and manipulate N3 documents. How built-ins work can be compared with the logical programming language Prolog, the built-in `list:first` can be used to get the first element of an RDF list, but on the other hand it can also be used to check if the first element is equal to a given value.

As stated earlier, N3 is a language built on top of Turtle, but at the same time, Turtle is a textual syntax for RDF allowing to write RDF graphs more compact and in natural text form [21]. In contrast to XForms 2.0, this makes Notation3 a perfect fit for this problem.

5.1. N3 Patch

As N3 is a language that is used to define rules and assertions, it is also a language that is used to define changes to a document. A Notation3 document is passed as a body of an HTTP `PATCH` request, identified by defining the `Accept-Patch` header equal to `text/n3`. This concept is described in the *Modifying Resources Using N3 Patches* subsection of the Solid specification [27] as the way to modify resources in a Solid pod. The request is targeted to the resource that should be modified. As said earlier, the body should be a N3 document and the content of this document should be a `solid:InsertDeletePatch` subject having a `solid:inserts` and `solid:deletes` predicate. The `solid:inserts` predicate should contain a list of triples that should be added to the resource and the `solid:deletes` predicate should contain a list of triples that should be removed from the resource. The `solid:InsertDeletePatch` subject can also have a `solid:where` predicate which should contain a list of triples that should be matched in the resource. If the `solid:where` predicate is not present, the `solid:inserts` and `solid:deletes` predicates should be applied to the whole resource. An example of such a N3 Patch body is given in Listing 2.

```
@prefix solid: <http://www.w3.org/ns/solid/terms#>.
@prefix ex: <http://www.example.org/terms#>.
@prefix ncal: <https://www.semanticdesktop.org/ontologies/2007/04/02/ncal/>.

_:executePolicy a solid:InsertDeletePatch;
  solid:inserts { ex:Todo ncal:todoStatus ncal:completedStatus. };
  solid:deletes { ex:Todo ncal:todoStatus ncal:inProcessStatus }.
```

Listing 2: Example of a N3 Patch body.

6. Reasoner Implementations

Next to a language to define the reasoning part, an implementation to execute the reasoning part is also needed. There already exist many implementations of various kinds. They differ based on the programming language they are implemented in and on the logic language they work on. As the focus of this thesis is on the logic language N3, the implementations working on N3 are the most relevant to discuss here. Existing implementations working on N3 include but are not limited to EYE [29], EYE-JS [30], jen3 [31] and N3.js [32]. Next to those N3 reasoner implementations, also RoXi [33] exists which is an implementation in Rust working on the Datalog [34] logic language with N3 syntax. Datalog is situated in the Prolog family using a bottom-up rather than a top-down evaluation model.

6.1. EYE

As EYE will be used during this thesis, a short description of it is given here. EYE [29] – short for Euler Yet another proof Engine – is a reasoning engine accepting N3 P-code which is being interpreted with the use of a Prolog virtual machine. N3 P-code is a Prolog interpretation of N3 data by parsing RDF triples and N3 rules. EYE avoids infinite rules by interpreting a N3 rule `P => C` as `P AND NOT(C) => C`. As discussed in *Drawing Conclusions from Linked Data on the Web: The EYE Reasoner* [29], because EYE tries to reach the goal set by a user by applying logical rules, EYE is in terms of algorithms a theorem prover. In EYE, independent proof validation is possible thanks to the possibility to follow the steps via which the proof came to its goal allowing one to understand the reasoning process and validity. It is being developed by De Roo, a researcher at Ghent University. It is one of the most complete N3 reasoners and is therefore used in this thesis.

6.2. EYE-JS

In addition to EYE, EYE-JS [30] is a distribution of EYE to deliver reasoning for the browser and node by the use of WebAssembly. Under the hood, it uses EYE by using the new technology *SWI-Prolog in the browser using WASM - Wiki* [35] allowing to run Prolog code in the browser. EYE-JS makes use of this new technology by running EYE in the browser or Node via this SWI-Prolog implementation and performing reasoning on the data. This removes the need for a separate EYE instance running on a server and allows for direct reasoning execution client side.

7. Comunica

Comunica [36] is a knowledge graph querying framework initially developed in 2018 as an open-source framework by IDLab at Ghent University - imec. Since September 2022, the *Comunica Association* is launched to make Comunica more sustainable in the long term. The primary goal of Comunica is to have one or more interfaces like SPARQL endpoints and Triple Pattern Fragments (TPF) interfaces over which SPARQL queries can be executed. Comunica querying works on Linked Data which can be published on the Web in many different shapes and forms using plain RDF files in various syntaxes. Such syntaxes include but are not limited to JSON-LD, Turtle, and HTML+RDFa. Comunica is being built as a set of modules allowing easy plugging in of new components to use different algorithms or experimental features. This is especially useful because of the increasing heterogeneity of Linked Data on the Web, which makes it hard to build a single query engine that can handle all of it. Components.js [37] is used to put the concept of dependency injection [38] to use in Comunica. This allows for easily swapping out components and easily adding new components, as well as configuring and combining them by just using a configuration file. As Comunica is written in JavaScript, it can be used in the browser, via the command line, or any Web or JavaScript application. In this thesis, Comunica is used in the browser to query resources with SPARQL. Next to SPARQL query evaluation, modularity, being Web-based and supporting heterogeneous interfaces, the engine also supports federated querying over different interfaces. This allows to query multiple sources at once and combine the results into a single result set.

8. Ember-Solid

Ember-solid [39] is an add-on for the Ember.js [40] front-end framework developed by redpencil.io. It allows easy integration with a user's Solid pod and conforms to the specifications in a way that is maximally abstracted for the developer. A developer can by working in the same way as with the frequently used ember-data [41] add-on, work with Solid resources. This allows for easy integration of Solid into existing Ember applications. This add-on can thus be seen as a (more limited) drop-in replacement for ember-data which instead of storing data in a certain kind of database, stores it in a Solid pod. Ember-solid itself makes use of the rdflib.js [42] library to work with RDF data. Rdflib.js is a Linked Data API for JavaScript that can be used in the browser and Node.js. It is being developed by Berners-Lee, the LinkedData team, and many contributors.

9. The Function Ontology

The Function Ontology (FnO) is a way to, semantically, define and describe implementation-independent functions, just like their relations to related concepts such as parameters, and mappings to specific implementations and executions. It is first introduced in *Implementation-independent function reuse* [43] by Ben De Meester et al. The full specification is available at https://w3id.org/function/spec [44].

A link can be made with the Hydra vocabulary as the `hydra:ApiDocumentation` can be seen as a `fno:Implementation`. To be able to describe what a `fno:Implementation` is, some other parts of the Function Ontology need to be discussed first. The ontology consists of some concepts. First, a `fno:Function` is a process that performs a specific task by associating one or more inputs to an output. It expects a list of ordered *Parameters* and returns a list of *Outputs*. The Parameters define the relation being used for the execution in the form of a predicate and can also hold a specific type or other metadata. *Functions* can be linked to *Problems*, which are a more general description in comparison to Functions. Problems themselves can be linked to other Problems by using the SKOS standard [45]. A `fno:Execution` links the input data and the resulting output data to the parameters and outputs of Functions. It allows one to describe independently of the implementation how input data is transformed into output data. A `fno:Implementation` is a set of function units. The description of the implementation itself is decoupled from the Function Ontology (FnO). The model is not limited to a specific set of supported development contexts by allowing any development context to be specified. Lastly, a `fno:Mapping` connects an abstract function with a specific part of a concrete implementation existing of a link between the function and the implemented method and a link between the inputs and outputs of a function and the parameters and returns values of the methods.

9.1. Orchestrator for a Decentralized Web Network

One application of the Function Ontology is the Orchestrator. The Orchestrator [46] is a specification that describes the implementation requirements for the Orchestrator component. This specification uses the Function Ontology as described earlier to describe what should happen when a trigger takes place. In the then-part of this policy, this FnO description is expressed. These policies are written in a *policy language* understandable by the Orchestrator. In practice, this means that policies are written using a *rule language* like SHACL, SPARQL, or Notation3. An example of a policy is taken from the Orchestrator specification [46] and is shown below in Listing 3. When the orchestrator now detects that a new triple `?notification a as:Create` is added to one of the watched resources, it will execute the then-part of the policy. In this case, it will send a notification to Bob. This is done by executing the function `ex:sendNotification` with the parameter `ex:notification` set to the value of `?notification`. The event to which the Orchestrator responds is called a trigger.

```
rule "Notify Bob about newly created artifacts"

when

   ?notification a as:Create .

then

   ?notification as:target <http://bob.institution.org/profile/card#me> .

   [ a fno:Execution ;
     fno:executed ex:sendNotification
        ex:notification ?notification
   ] .
```

Listing 3: Example of a policy.

Koreographeye [47] is a miniature implementation of this specification. It only uses a top layer of what FnO is capable of, FnO goes much further than this but for the use case of Koreographeye, this is enough. It only needs a vocabulary to be able to describe what JavaScript implementations need to be executed when a certain event takes place.

Chapter 4: High-Level Architecture

As was already made clear in the introduction, the goal of this thesis exists out of three parts, each having its own research question. The methods and applications used to answer these questions will overlap and build upon each other. In the following three chapters, the three research questions will be examined and discussed. Each chapter will contain a more in-depth architectural description and, per chapter, the relevant parts of these methods and applications will be explained and the subsequent chapters will further expand on these. This chapter will present the high-level architecture to present an overview of how the different parts of this thesis relate to each other.

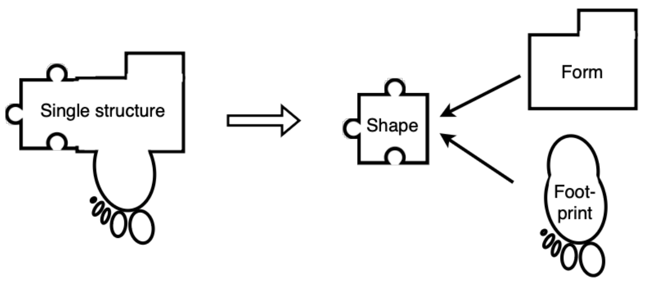


Figure 2: Transition from the traditional single structure where all the data is defined using a single vocabulary, to a three-part view of data, consisting of a form (for display), shape (for validation), and footprint (for reasoning) part.

A common way how web applications store their data is by using only one fixed structure or vocabulary. This means that the data is stored in a fixed structure and the application is built on top of this structure, leaving the app developer to decide on how the UI of the app will look like. Because of that, it is not possible to use the data with another application that uses a different structure, which will quickly be the case when applications are built without taking into account the use of other applications. This is even the case for many Solid apps that assume the data is stored in a fixed location in the pod with only one vocabulary and is also available in the pod. This means that hard-coded assumptions about data access patterns and structure are made about the data that is stored in the pod [3]. In 2019, Verborgh and Berners-Lee proposed a new vision [48] on how to build web applications that are more flexible and allow for more use cases. The idea of having a three-part view already originates from their view on *Linked Data Shapes*, *Forms*, and *Footprints*. This shift from a single structure to a three-part view is shown schematically in Figure 2. The left part is the current situation where the data is stored in a single structure and the application is built on top of this structure. The right part is the goal of this thesis where the data is divided into three parts, each with its own purpose, namely the *form* for the *display* part, the *shape* for the *validation* part, and the *footprint* for the *reasoning* part. As there has been a lot of research done on the topic of validation using shapes, as discussed in Section 2.3, this will not be a core part of this thesis.

The high-level architecture can also be viewed from a different angle. In traditional centralized web applications, different users interact with the same centralized web server using different interfaces. These web interfaces are written for that server and only work for that single web server. Additionally, not only is the data stored in a fixed structure, but the data is also stored on the servers of the application. This makes it impossible to use the data with another application. This is shown in Figure 3. Different users are shown interacting with the same traditional centralized web server using different interfaces. The interfaces used belong to that specific web server and work only for that web server. There is no interaction with any other data storage system, meaning that the data will stay on the application's server, being unavailable to other applications and creating the by David Simonds called *Walled Gardens* [49] where people are stuck to certain web applications because their data is stored on the servers of these applications and is not available to other applications.

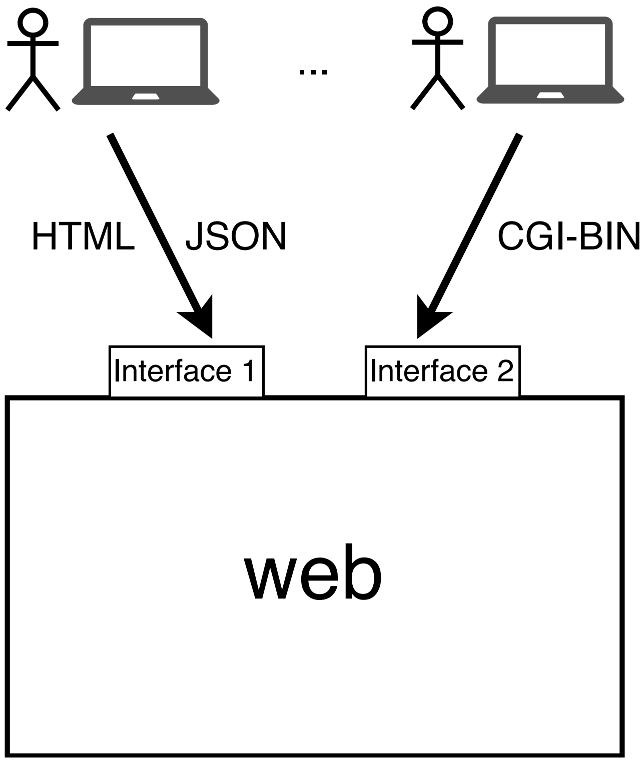


Figure 3: Users interact with the same traditional centralized Web server through different interfaces that are written for that server and work only for that Web server.

Solid has a decentralized approach to how data is stored and accessed by web applications. The user interacts with Solid apps using data from one or more pods. However, the apps still make assumptions about the data that is stored in the pod. The app is designed for one specific use case and the data is most of the time stored in a specific way.

A typical example is a Solid app that manages recipes. It makes some assumptions about the data, vocabulary, and storage location. These assumptions are different for a to-do app, a book review app, or any other app. This means that if there exists an app that let users fill in a form to enter a recipe, a separate app would be needed in the case that the user wants to enter a to-do list or a form to enter a book review. Figure 4 shows this high-level architecture of Solid apps where the user interacts with a Solid application designed for a specific use case using data from one or more pods by making assumptions about the stored data. This differs from the traditional centralized web applications where the data is stored on the servers of the application. This works because the Solid pod uses a standardized protocol and RDF format to communicate with the app. By using this standardized protocol for communication between the app and the pod, the problem of having all different web servers using different protocols from the first stage is solved.

This thesis tries to push this decentralized architecture a step further. A declarative Solid app is introduced that makes no assumptions about the interface and app itself. The idea is to tackle the problem of the previous stage where a separate app was still needed for each use case, no matter how similar they were. After all, it is much more efficient to have one app that can handle multiple use cases than to have multiple apps that can only handle one use case. To still be able to present a different user interface to the user for each use case, this user interface should be described in a declarative way that is then passed to the app. This declarative description of the user interface is called the *form description resource* in Figure 5 and as its name suggests, it describes the form that the user will see. The application still needs to be able to understand this declarative description of the user interface. This only holds if the form description resource is written using a certain ontology that the app can understand.

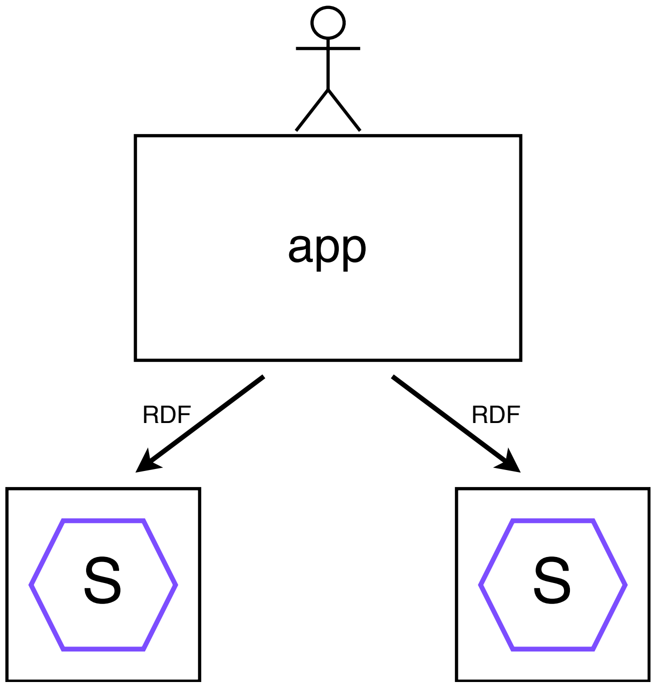


Figure 4: Users interact with a Solid application designed for a specific use case using data from one or more pods by making assumptions about the stored data.

Imagine a world where both Alice and Bob store their to-do lists in their Solid pods. Charly wants to create an app that allows to render these to-do's in a declarative app. As there is no standard ontology for to-do's, the case that Alice uses ontology A and Bob uses ontology B is likely to occur. This means that Charly has to write an app that can understand both ontology A and ontology B. However, without a standard, no assumptions can be made about the ontology used, it could be that only ontology A and B are used, but it could also be that Alice uses ontology A, Bob uses ontology B, and Dave uses ontology C. It becomes clear that Charly has to write an app that can understand all ontologies that are used to describe a to-do. To weaken this assumption, a so-called *N3 conversion rules resource* is provided to the app. The purpose of this resource is, in the case of forms, to convert the *form description resource* into a language that the application can understand. It does so by functioning as a dictionary between the ontology used in the form description resource and the ontology that the app can understand. This way, the app should only be able to understand one ontology, the ontology that the *N3 conversion rules resource* converts to. The translation from one ontology to another is called the schema alignment task, the execution of the policies when an action happens like pressing a button is called the footprint task. These tasks should be executed by the renderer app with the use of reasoning. Therefore, the renderer app uses a reasoner, local or remote, depending on the use case. How this works is covered further in Chapter 6.

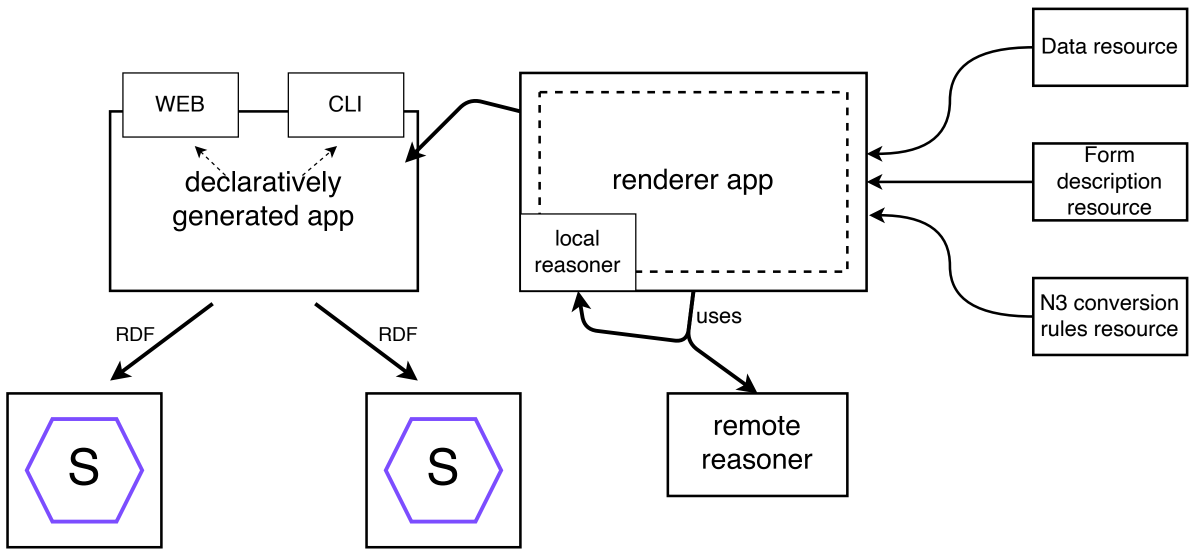

WEB
CLI
declaratively generated app
renderer app
local reasoner
Data resource
Form description resource
N3 conversion rules resource
uses
remote reasoner
RDF
RDF
S
S

Figure 5: Users interact with a dynamically generated application built by a form renderer application using the declarative form description, an optional data resource, and an optional resource with N3 conversion rules as input. This generic renderer application can build for multiple viewing environments without making assumptions about the interface and app itself. The renderer uses a reasoner to apply the schema alignment and footprint tasks. The user can use the generated application to interact with one or more Solid pods.

Lastly, a *data resource* can be provided to the app. This resource contains any pre-existing data that is used to fill in the form after rendering it. Depending on the policies, the data resource URI could also be used as the location to store the data that is entered in the form. These input resources are then used by the renderer app in Figure 5 to output the actual app that will be presented to the user. No assumptions should be made about the app itself or the interface that is used to interact with the app. The app could be a web app, a command line app, a mobile app, or any other kind of app. This concept of having a display part that is unrelated to the viewing environment is discussed in Chapter 5. This declaratively generated app then interacts with one or more Solid pods as was already the case in the previous stage to store the data that is entered in the form. This last step where the app interacts with the pod can be any kind of interaction with the pod or any other web server. Again, no assumptions should be made and other types of interactions such as HTTP requests to other services should be possible as well. The strength of this architecture is that it allows for a declarative way of describing these tasks that should happen when an action is performed by the user.

Chapter 5: Multiple Viewing Environments

This chapter will try to answer the first research question: “How can machines be controlled in a declarative way to create forms for producing RDF in **multiple viewing environments** (such as the web and text-based via a command line)?”. First, an explanation of the architecture and design choices will be given. After that, an evaluation part will be given based on a user experience study. To conclude the chapter, the implementation is discussed in more detail, followed by testing the solution against the research question and discussing the results.

1. Architecture

As was discussed earlier in Chapter 4, the architecture of the solution is split into three parts: display, validation, and reasoning. In this chapter, the display part will be discussed in more detail. The display part is the part that is responsible for rendering the form to the user. It describes how the form should look like, i.e. what fields it should contain.

Let’s first take a look at the flow of all the data that is involved in the process of creating and filling in a form. The flow of forms data is shown in Figure 6. First, some definition is needed such that a renderer application knows what to render. This definition is called a *form description* and is an RDF resource that describes the form. This form description can be defined in any ontology that is capable of describing a form. There already exist ontologies that can be used for this purpose, in this thesis three ontologies will be used: the SHACL ontology [5], the Solid-UI ontology [6], and the RDF-Form ontology [9]. The form description will do the decoupling of the three parts: display, validation, and reasoning. The advantage of such a definition is that there is a standard for how to describe a form and what actions should be taken when the form is submitted. The disadvantage of such a definition is that the display and validation parts are mixed in the same form description resource, while possibly being defined in different languages. Because both can be used to describe certain properties, e.g. required, it can create a form of ambiguity that conflicts with each other.

Let’s, for now, assume that such a form description exists. (How to create such a form description is discussed later in Chapter 6, along with an implementation of such a FormGenerator application.) The form description is saved in the user’s Solid pod in RDF. This RDF form description can then be used to render the form for any user who wants to fill it out. The *FormRenderer* is an application that allows the user to fill in the form via a Web browser. However, the strength of this split architecture lies in the fact that any application could be used to render the form. An example of such an application is the *FormCli* which is a command line interface that allows the user to fill in the form via the terminal. This application provides a text-based interface to fill in the form. As said before, the form description contains all the information needed to render the form, but it also contains the policies that describe what should happen with the data that is filled in the form after it is submitted. How these policies are defined and how they work in detail relates to the footprint tasks and is discussed in Chapter 6.

To apply the schema alignment tasks to the form description in Chapter 6, a *N3 conversion rules resource* is required in addition to the form description resource. The third input resource that is displayed in Figure 6 is the resource containing the pre-existing data triples. This resource is optional and can be used to fill in the form with pre-existing data. The decision was made to make this a separate resource as this separates the filled-in data from the form description. This way, any resource can be provided by the user potentially containing pre-existing data. This also means that a user who fills out the form does not need to have write access to the resource that contains the form description.

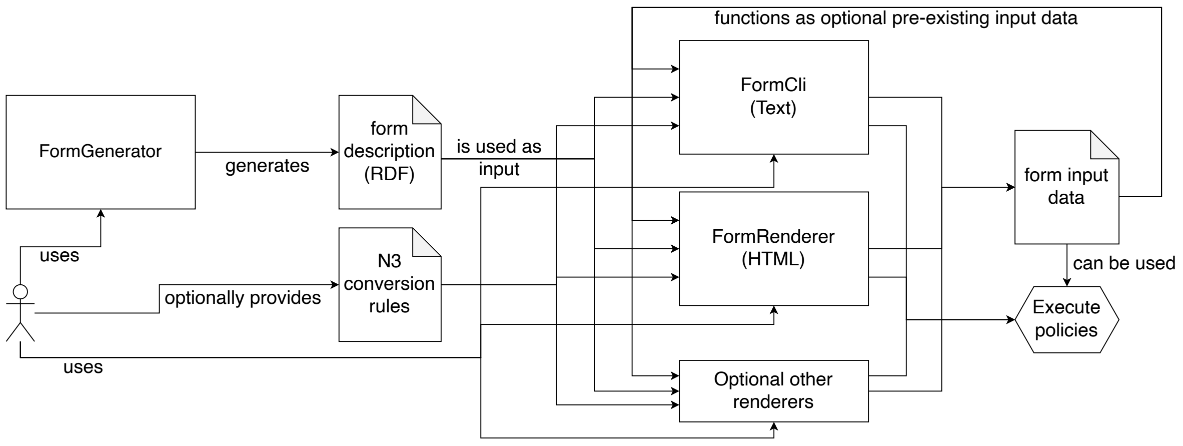
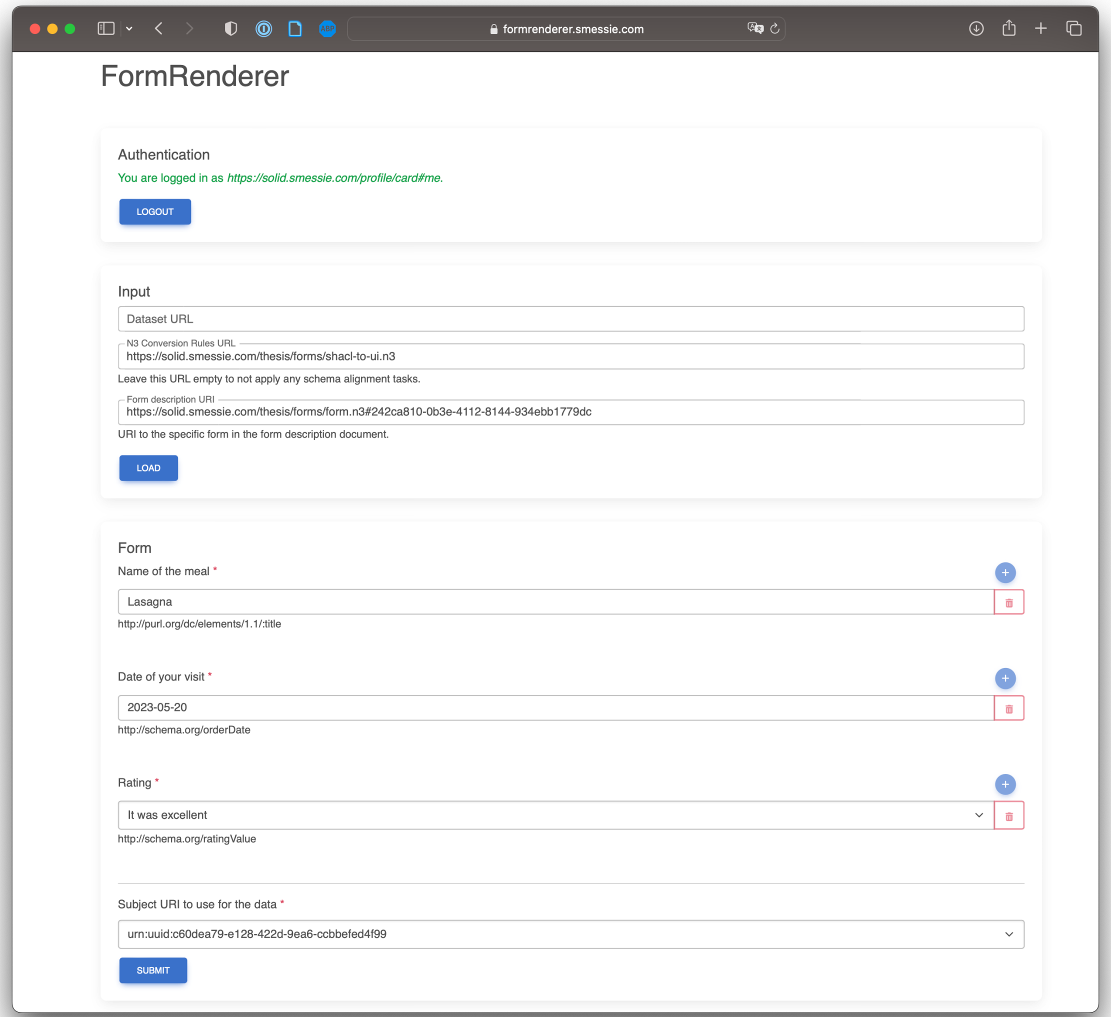


Figure 6: The user first creates a form description using the FormGenerator. Any renderer application, such as the FormRenderer or FormCli, can then be used to render the form for the user to fill out. The user can provide a resource with pre-existing data to fill in the form, and a resource with conversion rules to apply the schema alignment tasks to the form description in case the form description is not written in the same ontology as the renderer application. The completed form can then be saved to the user's Solid pod by executing the policies on submission.

By declaratively describing the form in RDF in the display part, it should be possible to render the form in any environment. Any existing or new ontology can be used, as long as it is capable of describing a form. The idea here is to prove that these descriptions are not dependent on a specific environment like the web with HTML but that they can be used in any environment like the command line as text. The RDF description should be parsed and then interpreted by the renderer app to display the form in the environment that it is designed for.

2. Implementation

2.1. Form Renderer



formrenderer.smessie.com

FormRenderer

Authentication

You are logged in as *https://solid.smessie.com/profile/card#me.*

LOGOUT

Input

Dataset URL

N3 Conversion Rules URL
https://solid.smessie.com/thesis/forms/shacl-to-ui.n3
Leave this URL empty to not apply any schema alignment tasks.

Form description URI
https://solid.smessie.com/thesis/forms/form.n3#242ca810-0b3e-4112-8144-934ebb1779dc
URI to the specific form in the form description document.

LOAD

Form

Name of the meal *
Lasagna
http://purl.org/dc/elements/1.1/:title

Date of your visit *
2023-05-20
http://schema.org/orderDate

Rating *
It was excellent
http://schema.org/ratingValue

Subject URI to use for the data *
urn:uuid:c60dea79-e128-422d-9ea6-ccbbefed4f99

SUBMIT

Figure 7: Screenshot of the implemented FormRenderer application.

This section will explain the FormRenderer application to render forms in a Web browser. A screenshot of the FormRenderer application is shown in Figure 7. In the next section, the text-based FormCli that runs in the terminal will be explained. In Figure 6 can be seen that the FormRenderer application, just like any other renderer application such as the FormCli, takes 3 URLs to resources as input from the user: the optional URI of a resource containing any pre-existing data triples to fill into the form, the optional URI to the resource containing the N3 conversion rules to apply the schema alignment tasks onto the form description, and the URI to the resource containing the actual form description. Let's first take a look at the simple case without schema alignment tasks and focus on the other two resources by assuming that the form description is already written using the same ontology as the renderer app, i.e. Solid-UI. First, an introduction to how this app was built will be given. The FormRenderer was created in the Vue.js framework, Subsubsection 5.2.1.1 will elaborate on this. Authentication is implemented in the application so that the user does not have to make his Solid pod publicly readable and writable.

This allows a user to authenticate with their Solid pod and then the app can read and write to the pod on behalf of the user. This is discussed in Subsubsection 5.2.1.2. Parsing the form description is discussed in Subsubsection 5.2.1.3 and parsing the pre-existing data to fill in the form is discussed in Subsubsection 5.2.1.4.

2.1.1. Vue

Vue is a progressive framework for building user interfaces. The Comunica query engine is used to query the resources that are passed to the app. However, Comunica did not work out of the box with Vue, so some workarounds had to be done to make it work, which are described here. To make it work, the `vite.config.js` file was adapted to make it work with Vue and Vite. The required changes are shown in Listing 4. All Comunica queries are executed in the browser with input data that is passed as text to the query engine. The data is fetched separately from the given URLs using the authenticated session because this allows the data to be manipulated before it is passed to the query engine.

```
import { defineConfig } from 'vite';
import vue from '@vitejs/plugin-vue';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  build: {
    commonjsOptions: {
      strictRequires: true,
    },
  },
  define: {
    global: "window",
    "process.env": {},
  },
});
```

Listing 4: Configuration changes in vite.config.js to make Comunica work with Vue.

2.1.2. Authentication

The resources containing the form description and the pre-existing data to fill in the form are fetched from the given URLs. However, it is easy to come up with a scenario where the user does not want to make these resources publically readable and writable. Therefore, a form of authentication was implemented such that the resources don't need to be publicly available. With this, the user can authenticate with a Solid pod and the application will automatically load the resources from the pod using an authenticated request. To authenticate, the `@inrupt/solid-client-authn-browser` library is used. The user has to enter their Solid Identity Provider (IDP) and then click the login button. The user is then redirected to the IDP to authenticate. After authentication, the user is redirected back to the application and the application can now make authenticated requests to the Solid pod. To prevent the user from having to authenticate every time they want to use the application, silent authentication [50] is used. This is done by calling the `handleIncomingRedirect({ restorePreviousSession : true })` function.

As before, the user is redirected to the IDP for authentication. However, if the user is still logged in, the IDP will not ask the user to authenticate again, but will immediately redirect the user back to the application without requiring any interaction from the user. In other words, this happens under the hood without the user being aware of it, which improves the user experience by eliminating the need to redirect to the authentication page.

2.1.3. Parsing the Form Description

First, all content of the resources linked by their URIs is fetched. The authenticated fetch from Inrupt's library is used in the case that the user is authenticated. As mentioned before, for now, the assumption is made that the form description is already written using the same ontology as the renderer app, i.e. Solid-UI. This assumption will later on in Chapter 6 be replaced with the appropriate solution of applying the schema alignment tasks to the form description. The form description is then parsed by the Comunica engine using the SPARQL query shown in Listing 5.

```
PREFIX ui: <http://www.w3.org/ns/ui#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?type ?property ?label ?from ?required ?multiple ?sequence WHERE {
  <${this.formUrl}> ui:parts ?list .
  ?list rdf:rest*/rdf:first ?field .
  ?field a ?type;
    ui:property ?property.
  OPTIONAL { ?field ui:label ?label. }
  OPTIONAL { ?field ui:from ?from. }
  OPTIONAL { ?field ui:required ?required. }
  OPTIONAL { ?field ui:multiple ?multiple. }
  OPTIONAL { ?field ui:sequence ?sequence. }
}
```

Listing 5: SPARQL query to parse the form description.

This outputs a list of JavaScript objects, each representing a field in the form. First, this list is sorted by the defined sequence number of each field. Then, in the case of a choice field, the options are retrieved from the resource specified in the `from` property of the field by performing an additional SPARQL query on the resource. This SPARQL query is given in Listing 6. The options are then added to the field object in JavaScript.

```
PREFIX ui: <http://www.w3.org/ns/ui#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?value ?label WHERE {
  ?value a <${field.from}> ;
    skos:prefLabel ?label.
}
```

Listing 6: SPARQL query to retrieve and parse the options of a choice field.

As can already be inferred from the listings above, the base vocabulary is chosen to be Solid-UI for this implementation, as it is a vocabulary that was specifically designed for use in forms and is already being used by the SolidOS team in this context. It already exists for quite some time, and it is fairly complete in terms of elements, or at least for this use case. Another alternative would have been SHACL, but this is a more general ontology and is not specifically designed for forms. It could however be used for this, and later on, it will also be used to show how the FormRenderer application can render a form written in another ontology, i.e. SHACL.

2.1.4. Parsing the Pre-Existing Data

The FormRenderer app can also be used to edit existing data. Before going any further, let's look at how the data is structured. When saving data after submitting a form, the data is stored with a certain structure. This structure is shown in Listing 7. The `?subject` can be any URI and is used to identify the triples belonging together as an answer to a form. To specify to which class the subject belongs, the `?targetClass` variable is used. This corresponds with the binding of the form when the form description was generated. For example, when using SHACL as the language for the form description, the main form element is a `sh:NodeShape` and the binding is specified using the `sh:targetClass` property. In the case of the Solid-UI ontology, the main form element is a `ui:Form` and the binding is specified using the `ui:property` property. Next, the `?predicate` is the predicate of the triple, this corresponds with the binding or property of the form field. Again, depending on the used ontology, what this property is, is defined using a different predicate but for Solid-UI, this predicate is defined by the `ui:property` property on the field element. The `?value` is the value of the triple, this corresponds with the value that was entered in the form field. This can be a literal or a URI and fully depends on the type of the form field. Finally, many more values can be entered in the form and they all can be saved using other triples with the same subject. This is represented in the example with the `?otherPredicate` and `?otherValue` variables.

```
?subject a ?targetClass ;
        ?predicate ?value ;
        ?otherPredicate ?otherValue .
```

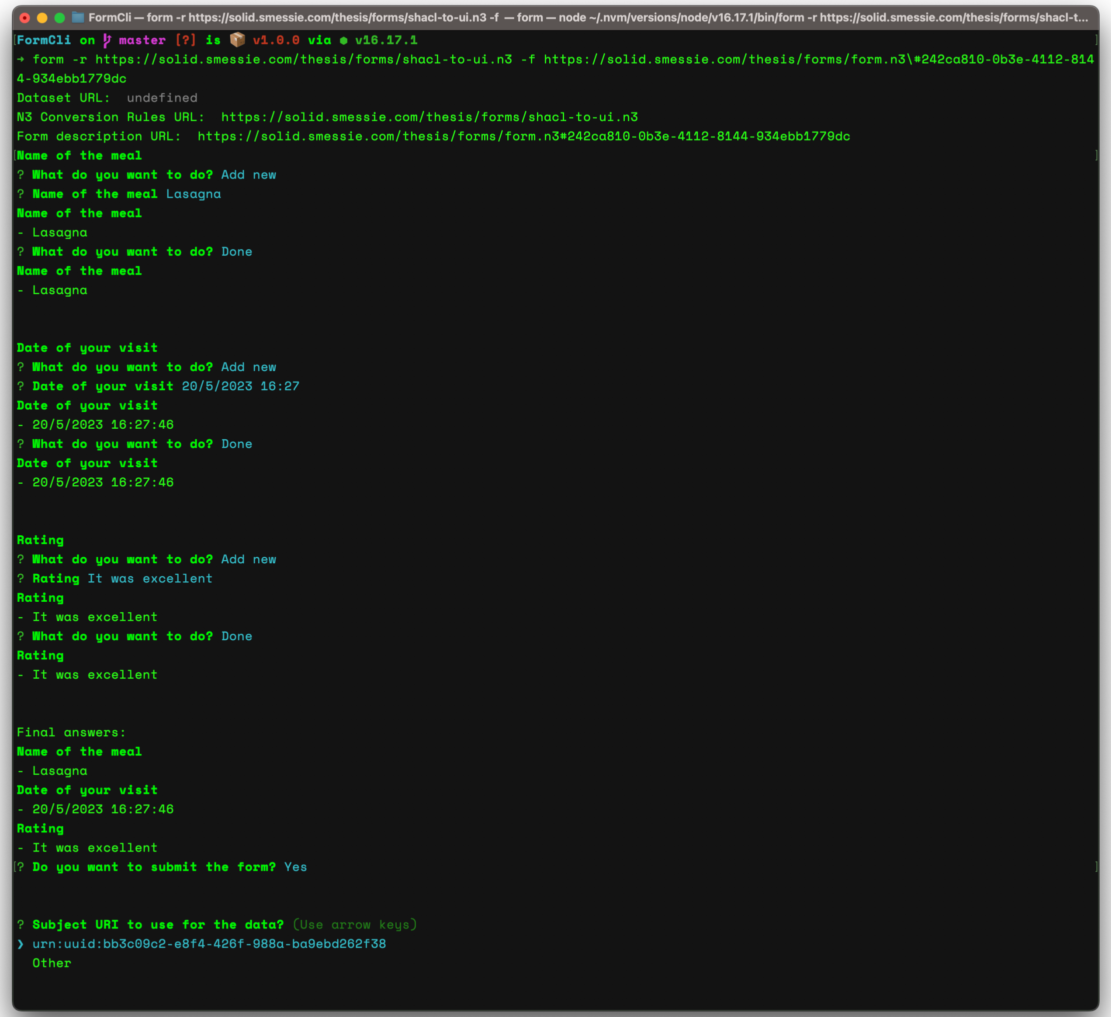
Listing 7: Structure of the data saved after submitting a form.

As every field in the form has a binding, the pre-existing data can be retrieved by performing a SPARQL query on the resource containing the data. This binding defines exactly what the triple's predicate should be with this value as object. As a result, retrieving the pre-existing data is as simple as performing a SPARQL query on the resource containing the data with the binding as the predicate. The SPARQL query to retrieve the pre-existing data is given in Listing 8.

```
SELECT ?s ?value WHERE {
  ?s a <${this.formTargetClass}> ;
    <${field.property}> ?value.
}
```

Listing 8: SPARQL query to retrieve the pre-existing data for a certain field.

2.2. Form Cli



```
FormCli — form -r https://solid.smessie.com/thesis/forms/shacl-to-ui.n3 -f — form — node ~/.nvm/versions/node/v16.17.1/bin/form -r https://solid.smessie.com/thesis/forms/shacl-t...
FormCli on master [?] is 📦 v1.0.0 via ● v16.17.1
→ form -r https://solid.smessie.com/thesis/forms/shacl-to-ui.n3 -f https://solid.smessie.com/thesis/forms/form.n3\#242ca810-0b3e-4112-814
4-934ebb1779dc
Dataset URL:  undefined
N3 Conversion Rules URL:  https://solid.smessie.com/thesis/forms/shacl-to-ui.n3
Form description URL:  https://solid.smessie.com/thesis/forms/form.n3#242ca810-0b3e-4112-8144-934ebb1779dc
Name of the meal
? What do you want to do? Add new
? Name of the meal Lasagna
Name of the meal
- Lasagna
? What do you want to do? Done
Name of the meal
- Lasagna


Date of your visit
? What do you want to do? Add new
? Date of your visit 20/5/2023 16:27
Date of your visit
- 20/5/2023 16:27:46
? What do you want to do? Done
Date of your visit
- 20/5/2023 16:27:46


Rating
? What do you want to do? Add new
? Rating It was excellent
Rating
- It was excellent
? What do you want to do? Done
Rating
- It was excellent


Final answers:
Name of the meal
- Lasagna
Date of your visit
- 20/5/2023 16:27:46
Rating
- It was excellent
? Do you want to submit the form? Yes


? Subject URI to use for the data? (Use arrow keys)
❯ urn:uuid:bb3c09c2-e8f4-426f-988a-ba9ebd262f38
  Other
```

Figure 8: Screenshot of the implemented FormCli application.

Just like the previously discussed FormRenderer app, FormCli is also a form renderer application. The difference is that this application is a command line application, meaning that it can be used without a graphical user interface. This application is written in JavaScript and uses the Node.js runtime environment. It uses the same library as the FormRenderer app to query the different resources, namely Comunica. The architecture and implementation of this application are very similar to the FormRenderer app. In particular, in terms of parsing the form description and any pre-existing data, the implementation is exactly the same, using the same SPARQL queries as shown in Listing 5, Listing 6, and Listing 8. A screenshot of the application is shown in Figure 8.

2.2.1. Command-Line Prompting of Form Questions

The main difference between the FormRenderer app and the FormCli app is that the FormCli app does not have a graphical user interface. Instead, it uses a command-line interface to prompt the user with the different questions contained in the form. This is done using the Inquirer.js [51] library. This library allows for creating a list of questions that can be interactively asked to the user on the command line. Based on the field type, a different kind of prompt is used. To support easy input of dates, the inquirer-date-prompt [52] library is used. This is a plugin for the Inquirer.js library that allows for easily asking the user for a date in an intuitive way.

2.2.2. Lack of Authentication

The FormCli app does not support authentication with a Solid identity provider. This is because the Solid protocol does not support proper authentication with a command-line application yet. Inrupt has developed a library that allows for authentication with a Solid identity provider using the command line, but this requires prerequisites like refresh tokens and client credentials to be set up manually [53]. This is beyond the scope of this thesis and therefore the FormCli application does not support authentication. The most important part of this application is showing that it is possible to create a form renderer application that can be used without a graphical user interface and that does not necessarily require authentication.

3. Discussion

Both the FormRenderer app and the FormCli app are proof of concepts that were successfully implemented and show that it is possible to create a form renderer application in multiple viewing environments. The source code of the FormRenderer app can be found at https://github.com/smessie/FormRenderer and a live version of the application can be found at https://formrenderer.smessie.com. The source code of the FormCli app can be found at https://github.com/smessie/FormCli. The same form description resource can be used for both applications resulting in the same form being prompted to the user. Only the layout will differ as the FormRenderer app uses a graphical user interface in HTML and the FormCli app uses a text-based command-line interface. In addition, the FormCli application also shows that a vocabulary such as Solid-UI does not depend on HTML to be used to represent elements.

The first research question was: "How can machines be controlled in a declarative way to create forms for producing RDF in **multiple viewing environments** (such as the web and text-based via a command line)?". The FormRenderer app and the FormCli app are apps in different viewing environments and thus demonstrate that it is possible to create a form renderer application in multiple viewing environments. The source code shows how this can be done with the use of SPARQL queries on the declarative form description executed by the Comunica query engine. The form was described in a declarative way as the display part is fully described using the already existing Solid-UI ontology. By making form descriptions portable and not tight to one rendering environment or one rendering logic, machines can be controlled to create forms for producing RDF in multiple viewing environments. The form description describes in a declarative way what should be displayed and what should happen in case of a certain action. Because this is described in a machine-readable way using RDF, a machine can interpret this and execute the right actions.

Some user feedback was gathered as an evaluation of the built applications. As they involve the more complete applications as extended in the next chapter, the full and in-depth discussion of the user feedback is given at the end of that chapter. However, it is worth mentioning that all participants successfully managed to fill in a form that was rendered from such a form description using the FormRenderer app. They mentioned that the form was easy to understand and that they did not notice that Linked Data and Solid were used in the background. This shows that describing the form in a declarative way is a working approach to creating forms for producing RDF in multiple viewing environments.

Chapter 6: Schema Alignment and Footprint Tasks

The second research question is: “How can machines be controlled in a declarative way to perform **schema alignment and footprint tasks** by the use of reasoning?”. This question will be answered in this chapter through the implementation of a series of applications that perform reasoning tasks related to schema alignment and footprinting. First, the architecture of the applications will be explained. An approach to this architecture will then be presented as a bridge to the application’s implementation. Then, the implementations of the applications will be described in detail after which the applications will be evaluated. The chapter concludes with a discussion section.

Imagine a world where Alice made a to-do list written in a certain language A, and Bob wants to display this to-do list in his own application, however, his application only understands language B. This is a problem because now Bob cannot display Alice’s to-do list in his application. This problem can be solved by translating Alice’s to-do list from language A to language B before displaying it in Bob’s application. This translation can be done manually by Alice, but this is a lot of work and it is not scalable. It would be better if this translation could be done automatically by a machine, with the use of a kind of dictionary that translates from language A to language B. In the next section, an architecture will be proposed that solves this problem. First, the architecture will be applied to the to-do list example, after which the same concepts will be applied to the more complex and general form-related apps.

1. Architecture

1.1. To-Do List Example

The user interacts with the to-do app via the browser. The URL to the *dataset resource* is necessary to be provided to the app. This is the resource that contains the to-do list and where the user’s updated to-do items will be saved. The to-do items can be described in any ontology. However, the app cannot understand every possible ontology, so the concept of a set of N3 conversion rules is introduced. These N3 rules are rules which map any vocabulary to the base vocabulary which the to-do app can understand. It can be seen as the dictionary that translates from language A to language B. This way, the app can understand any vocabu-

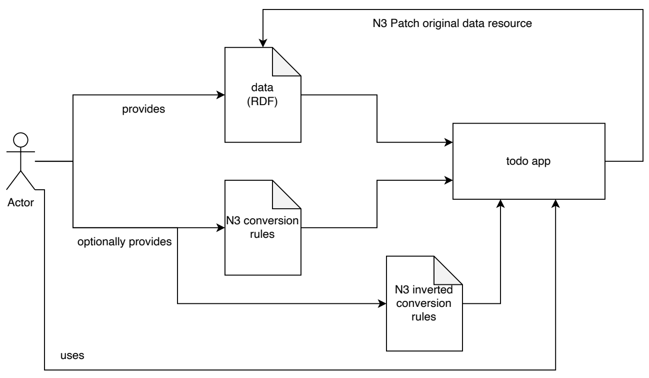


Figure 9: Users interact with the to-do application by providing a URL to a dataset resource and optionally a set of N3 conversion rules and a set of N3 inverted conversion rules that translate the vocabulary of the dataset resource into the vocabulary of the to-do application and vice versa. The to-do application updates the dataset resource using N3 Patch Requests.

lary that is passed to it as long as there is a set of rules that maps it to the base vocabulary. The process of mapping any vocabulary to the vocabulary that the app understands is called a *schema alignment task* and belongs to the reasoning part of the three-part view. The result of this schema alignment task is a set of to-do items that the app can understand. When the user now uses the application to add a new to-do item or to mark an item as done, new triples will be added to the dataset resource. These triples are in the vocabulary that the app understands, but this is not the same vocabulary as the one of the dataset resource. Therefore, the app needs to convert these triples back to the original vocabulary of the dataset resource. It uses the set of *N3 inverted conversion rules* to do this.

1.2. Form Generator

A form generator app is the first application in the forms flow and is used to generate an RDF description of a form in a certain UI ontology. This application lets one intuitively create such a form description requiring as little prior knowledge about RDF as possible from the user interacting with the app. By using drag-and-drop one can easily add, edit, reorder, and remove fields to an existing form, or of course create a new form from scratch, using one of the supported vocabularies.

The form generator app with Solid was made as a first step to getting in touch with all the existing technologies and getting familiar with the topic of defining forms in RDF. Unlike the idea of the thesis, the first version of this app used only one single vocabulary to define the whole form. However, this constraint shows that those vocabularies are missing the ability to clearly define the actions that should happen at certain events. It works to define how the form should look, but a way to put logic in the form was missing. This also shows the need for the research that is being done in this thesis. Although there was previously stated that all UI elements are defined in RDF using only one single ontology, the app supports multiple ontologies in which this RDF representation can be stored. For this, when making a new form, one can choose between 3 different UI ontologies being Solid-UI [6] by the SolidOS team, SHACL [5] by the W3C and Beeke's RDF-Form [9]. As a form generator application, we do not want to choose the vocabulary for the user, but we want to give the user the freedom to choose the vocabulary that suits his needs best. It could be that the user wants to build a form using some specific vocabulary because he wants to use the form with a specific form renderer that only supports that specific vocabulary. Or it could be that the user needs some specifics of a vocabulary like a min count and max count that is not present in the other vocabulary. By supporting all three vocabularies, we impose as few boundaries as possible on the user.

In addition to describing how the form should look like in the form description, the form generator should also be able to describe what should happen on certain events. For example, when the user clicks on the submit button, the form description should contain a description of what should happen. These policies have to be defined in the form description by the form generator application.

1.3. Form Renderer

The form renderer app is the second application in the forms flow and is used to render a form. In addition to rendering the form what already existed since the previous chapter, the form renderer app now also has to be able to execute the policies that are defined in the form description. The process of executing these policies is called the **footprint tasks** and belongs to the reasoning part of the three-part view. In this example, one kind of policy should be defined in the form description, namely the policy that should be executed when the user clicks on the submit button. There can be multiple different actions as part of this policy. An action that should be executed is to send the form data to a certain URL. This URL, HTTP method, and Content-Type are defined in the form description. Another action that should be supported is redirecting the user to a certain URL. In this case, the URL is also defined in the form description as part of the policy.

Just as was the case in Subsection 6.1.1, the form renderer app needs to be extended with *schema alignment tasks* to be able to understand any ontology for which a mapping to the ontology that the form renderer app understands exists. A new input parameter is therefore needed to be able to provide the form renderer app with the set of conversion rules.

2. Approach

2.1. To-Do List Example

The base vocabulary that the app understands is chosen to be `http://www.w3.org/2002/12/cal/ical#` (`cal`). This is an already existing ontology that is used to describe events and tasks. Which is the most important is that it can be used to describe to-do items. The exact vocabulary that the app is using is not of high importance, as long as it can be used to describe to-do items because the whole point is to be able to use any ontology. In the case that the user wants to use a different ontology, the user must provide a set of N3 rules that maps the ontology to the base ontology. An example of such a rule is given in Listing 9 where the `cal` vocabulary is mapped to the `ncal` (`https://www.semanticdesktop.org/ontologies/2007/04/02/ncal/`) vocabulary. In this example, the triple `?uri ncal:summary ?name` is mapped to the triple `?uri schema:text ?name`. This means that the app will understand the `ncal` vocabulary as if it was the `cal` vocabulary, assuming that such rules are provided for each triple that is used in the ontology to describe a to-do item.

```
@prefix cal: <http://www.w3.org/2002/12/cal/ical#>.
@prefix schema: <http://schema.org/>.
@prefix ncal: <https://www.semanticdesktop.org/ontologies/2007/04/02/ncal/>.

{
    ?uri ncal:summary ?name.
} => {
    ?uri schema:text ?name.
}.
```

Listing 9: Example of a N3 rule to go from the cal vocabulary to the ncal vocabulary.

Next to the N3 conversion rules, the app also requires a set of inverted N3 conversion rules to go back to the original ontology when changes are made to the to-do items. This is e.g. when a new to-do item is added to the list, or when an item is marked as done. In the most simple case, the inverted N3 rules will be the same as the N3 rules but with the rule premise and the rule conclusion reversed. However, in some situations, this will not suffice. More specifically, in the case of marking a to-do as done or not done. When using some vocabulary, both triples should be inserted and deleted, while in other vocabularies, only triples should be inserted or triples should be deleted as is the case with the `cal` vocabulary that is used in the example to-do app.

2.1.1. Policies to Describe What Should Happen on Toggling To-Do Status

One problem that arises is that the vocabulary that the app uses only uses one single triple with the predicate `cal:completed` with the DateTime as the object to describe whether a to-do item is done or not. However, the `ncal` vocabulary uses two triples with the predicates `ncal:completed` and `ncal:todoStatus` to describe whether a to-do item is done or not. This means that in the second vocabulary, there also exists a triple `?uri ncal:todoStatus ncal:inProcessStatus` in the case that the item is not completed yet, which is not present in the first vocabulary. To phrase it differently, to go from not completed to completed in the app vocabulary `cal`, only the triple `?uri cal:completed ?time` needs to be inserted, while in the `ncal` vocabulary, both `?uri ncal:completed ?time` and `?uri ncal:todoStatus ncal:completedStatus` need to be inserted, and on top of that, the triple `?uri ncal:todoStatus ncal:inProcessStatus` needs to be deleted. This is a problem because according to the app vocabulary, there are no triples to be deleted in this case. There are thus no triples to be used in the rule premise of a hypothetical rule to map from the app vocabulary to the `ncal` vocabulary, or any other vocabulary that also requires a triple to be deleted when marking a to-do item as done. This problem calls for a more complex architecture to be able to handle this kind of situation. To put as few limitations as possible and to tackle this problem, **policies** are introduced. Policies are the second half of the reasoning part of the three-part view. They are called the **footprint tasks** and describe what should happen when a certain action is performed. In this case, the action is marking a to-do item as done or not done. These policies are used to describe the exact changes that need to be made to the data when a to-do item is marked as done or not done. That is, the policies describe the exact triples that need to be inserted and deleted.

```
@prefix ex:   <http://example.org/> .
@prefix cal: <http://www.w3.org/2002/12/cal/ical#>.
@prefix ncal: <https://www.semanticdesktop.org/ontologies/2007/04/02/ncal/>.
@prefix pol: <https://www.example.org/ns/policy#> .
@prefix fno: <https://w3id.org/function/ontology#>.

{
    ?id ex:event ex:MarkCompleted.
    ?id cal:completed ?completedAt.
} => {
    ex:CompletedPolicy pol:policy [
        a fno:Execution ;
        fno:executes ex:updateResource ;
        ex:insertTriples [
            ncal:completed ?completedAt ;
            ncal:todoStatus ncal:completedStatus
        ] ;
        ex:deleteTriples [
            ncal:todoStatus ncal:inProcessStatus
        ] ;
        ex:subject ?id
    ] .
}.
```

Listing 10: Example of a N3 rule describing policy to mark a to-do item as done.

The N3 rule in Listing 10 displays an example policy that is executed when a to-do item is marked as done. The rule premise describes the event and the rule conclusion describes the policy that should be executed when this event occurs. The policy also describes the triples that need to be inserted and deleted and the subject of the triples that need to be updated. The policy describes that the triples `?uri ncal:completed ?time` and `?uri ncal:todoStatus ncal:completedStatus` need to be inserted, and the triple `?uri ncal:todoStatus ncal:inProcessStatus` needs to be deleted. The subject of the triples that need to be updated is the to-do item that is marked as done. As can be seen, the rule premise not only contains a triple describing the event but also the data that is used to execute the policy.

2.1.2. FnO as Policy Language

To describe policies, two languages are needed: a *rule language* and a *policy language* to describe what actually should happen when a policy is executed. As rule language, N3 is used. This is the same language that is used to describe the conversion rules in the schema alignment tasks and their N3 rules do exactly what is needed. To describe the policy, a basic version of the FnO ontology, which was described earlier under Section 3.9, is used. In the example in Listing 10, `fno:Execution` describes the policy as executing the `ex:updateResource` function. The function `ex:updateResource` is a function that is used to update a resource in the user's Solid pod. `ex:insertTriples` and `ex:deleteTriples` are extra predicates used to describe the triples that need to be inserted and deleted and can thus be seen as arguments to the function `ex:updateResource`. The subject of the triples that need to be updated is described by the predicate `ex:subject`.

In this architecture, the choice was made to use FnO and not e.g. Hydra which was described in Subsection 2.1.5. The reason for this is that Hydra is a vocabulary that is designed to be used in the context of describing a Web API on the server side. Its intended use is to describe the operations to the client that can be performed on a resource, and the data that is returned by these operations. However, this is not what policies are. Policies should describe the client-side operations that need to be performed when a certain event occurs. Furthermore, this can be much more than just performing an HTTP request to the server. Hydra does not allow to describe something else than an HTTP request, while FnO allows to describe any kind of operation. Next to that, FnO is being developed by people in the same research group as where this thesis was written. This is also an additional plus that only strengthens the choice for FnO.

2.2. Forms Flow

Because there should be put as few restrictions as possible on the form description, the form renderer app should be able to render any form description using any ontology. As it is unfeasible for an application to understand all the different ontologies that exist and will ever exist, another approach is needed. The concept of a letter and a dictionary is used to solve this problem. One vocabulary is chosen as the base vocabulary understood by the form renderer application and then together with the form description described in this vocabulary, a dictionary or set of N3 conversion rules is passed along. These N3 rules are rules which map any vocabulary to the base vocabulary. This way, the form renderer can understand any vocabulary that is passed to it as long as there is a dictionary that maps it to the base vocabulary.

2.2.1. Policies to Describe What Should Happen on Submission

To describe what should happen when the user submits the form, *policies* are used. Policies are the second half of the reasoning part of the three-part view. They are called the **footprint tasks** and describe what should happen when a certain action is performed. In the case of a form, the action is submitting the form. This action is defined in RDF as the triple `?id ex:event ex:Submit.`. Just as with the policies to describe what should happen when a to-do item is marked as done in Subsubsection 6.2.1.1, the policies to describe what should happen when a form is submitted are also described with N3 as rule language and FnO as policy language. This use case involves two specific types of policies. An example is shown in Listing 11. However, the generic structure from which they are built allows other types of policies to be easily defined. The `fno:executes` predicate is used to describe which function should be executed or thus what type of policy it is. The `ex:httpRequest`, of which the first rule is an example, is a policy that describes that an HTTP request should be performed. The other triples defined in the policy describe the arguments that are needed to perform the HTTP request. The `ex:method` predicate describes the HTTP method that should be used, the `ex:url` predicate describes the URL to which the request should be sent, and the `ex:contentType` predicate describes the content type of the request. The `ex:redirect` policy is a policy that describes that the user should be redirected to another page after the form is submitted successfully. The `ex:redirect` policy only needs one argument, which is the URL to which the user should be redirected. The second rule is an example of such a type of policy.

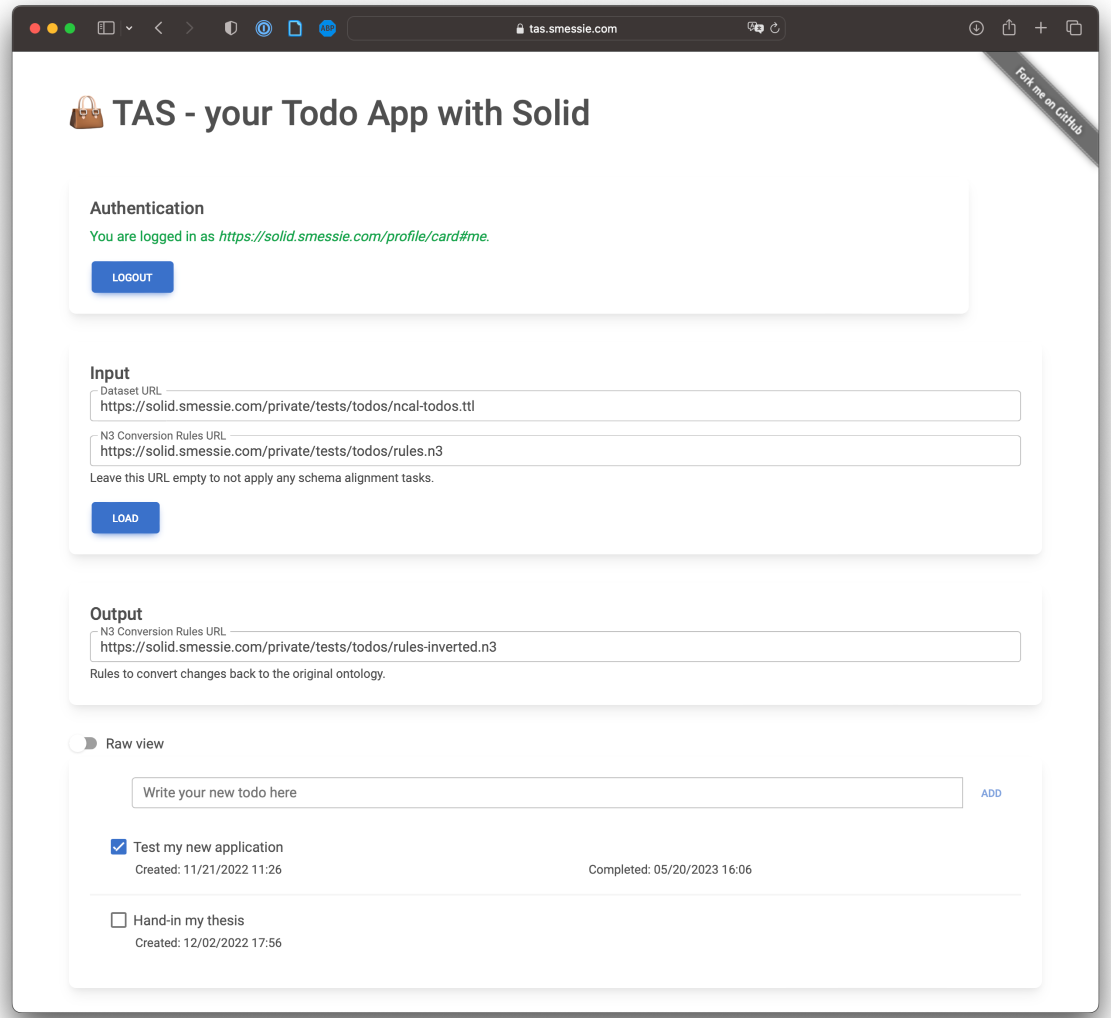
```
@prefix ex:   <http://example.org/> .
@prefix pol: <https://www.example.org/ns/policy#> .
@prefix fno: <https://w3id.org/function/ontology#>.

{
  ?id ex:event ex:Submit.
} => {
  ex:HttpPolicy pol:policy [
    a fno:Execution ;
    fno:executes ex:httpRequest ;
    ex:method "POST" ;
    ex:url <https://httpbin.org/post> ;
    ex:contentType "application/ld+json"
  ] .
} .
{
  ?id ex:event ex:Submit.
} => {
  ex:RedirectPolicy pol:policy [
    a fno:Execution ;
    fno:executes ex:redirect ;
    ex:url <https://smessaert.be>
  ] .
} .
```

Listing 11: Example of N3 rules describing different policies to be executed on the form submission event.

3. Implementation

3.1. To-Do App With Solid

The next step is to apply this reasoning in the browser to a concrete, but still simple, example application. For this, a simple to-do application is created that uses the Solid protocol to store the data in a Solid pod. A screenshot of the application is shown in Figure 10. The application is created using the Vue framework and the Vite build tool so the same workaround to make Comunica work with Vue is used as described earlier in Subsubsection 5.2.1.1. The different input resources as mentioned earlier in Subsection 6.1.1 are passed to the application by using their URL. The application will then fetch the data from the given URL. Authentication is implemented in the same way as the FormRenderer app as discussed in Subsubsection 5.2.1.2. The user can log in via the Solid IDP and the application will then fetch the data from the given URL which can be a private resource that the user has access to in a certain Solid pod.



TAS - your Todo App with Solid

Authentication

You are logged in as *https://solid.smessie.com/profile/card#me.*

LOGOUT

Input

Dataset URL: https://solid.smessie.com/private/tests/todos/ncal-todos.ttl

N3 Conversion Rules URL: https://solid.smessie.com/private/tests/todos/rules.n3

Leave this URL empty to not apply any schema alignment tasks.

LOAD

Output

N3 Conversion Rules URL: https://solid.smessie.com/private/tests/todos/rules-inverted.n3

Rules to convert changes back to the original ontology.

Raw view

Write your new todo here ADD

☑ Test my new application
Created: 11/21/2022 11:26 Completed: 05/20/2023 16:06

☐ Hand-in my thesis
Created: 12/02/2022 17:56

Figure 10: Screenshot of the implemented Todo App with Solid.

3.1.1. Parsing the Data to the To-Do Items

As the to-do application only understands the `cal` vocabulary, the data needs to be aligned to this vocabulary before it can be used by the application. Therefore, schema alignment tasks come into play. This is done by applying the N3 conversion rules to the data. This is implemented by using the N3 rules as a query and the to-do resource data as data to execute the query on. This reasoning is then executed in the browser using the EYE-JS library and the results are then used by the application as the input data for the to-do items. However, just like Comunica, EYE-JS did not work out of the box with Vue, so some workarounds had to be made to make it work which will be described here. The problem lies in the fact that ES2020 is not fully supported by default in Vite 3 yet while EYE-JS requires ES2020. Vite 3 is however the build tool used by Vue to build the application. To add support for ES2020 in Vite 3, this target had to be explicitly configured in the `vite.config.js` file as shown in Listing 12 [54].

```
import { defineConfig } from 'vite';
import vue from '@vitejs/plugin-vue';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  optimizeDeps: {
    esbuildOptions: {
      target: ['es2020', 'safari14'],
    },
  },
  build: {
    target: ['es2020', 'safari14'],
  },
});
```

Listing 12: Configuration changes in vite.config.js to make EYE-JS work with Vue [54].

To go from the data to the to-do items, the data needs to be parsed. For this, a SPARQL query is used on the data to get the to-do items. To execute this query, the Comunica query engine is used. Comunica was earlier described in Section 3.7. All Comunica queries are executed in the browser with input data that is passed as text to the query engine. The data is fetched from the given URLs separately using the authenticated session because this allows for manipulations of the data first, such as the aforementioned schema alignment tasks, before passing it to the query engine.

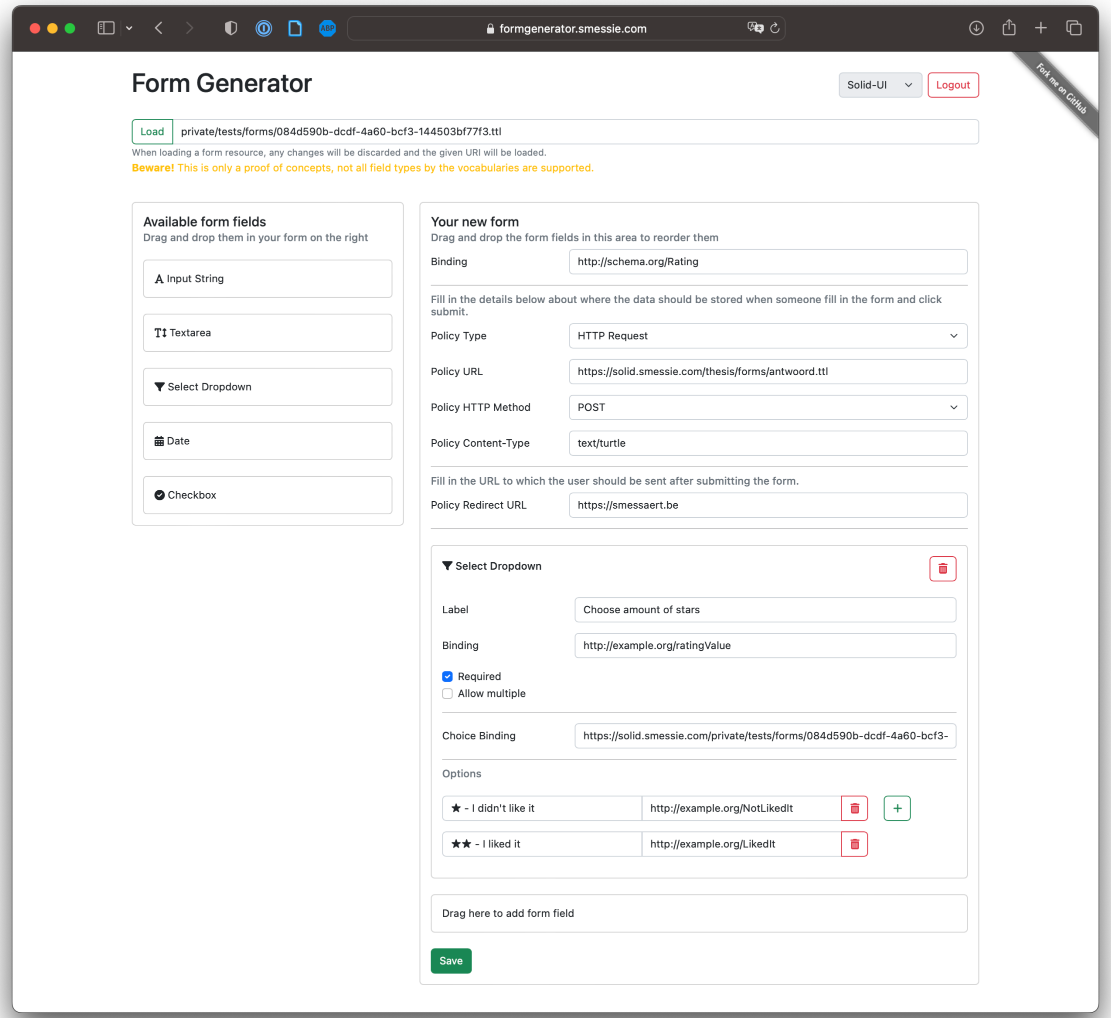
3.1.2. Adding New To-Do Items

Adding new items can be implemented quite straightforwardly by generating the new triples that should be added to the data resource. However, just like the data needs to be aligned to the `cal` vocabulary when loading into the application, the new triples also need to be aligned to the vocabulary of the data set while writing back to the resource. This is an inverted alignment for which the inverted N3 conversion rules should be used. This is the third and last input resource that is passed to the application. Reasoning with the help of EYE-JS is used for this as well. The work is not done after describing which triples should be inserted and deleted. These updates need to be performed as well. The new triples are added to the data resource by using a N3 Patch request as explained in Subsection 3.5.1. To perform this N3 Patch, the authenticated session retrieved from the Solid IDP is used. First, a SPARQL Update [55] request was considered instead of N3 Patch as this used to be the way to go to update data in a Solid pod. It is often used in existing applications but although it is still supported by most if not all Solid servers, it is no longer part of the Solid specification. The only official way to patch data in a Solid pod is now using N3 Patch. In recognition of the importance of maintaining compliance with the latest standards, the decision was thus made to use N3 Patch instead of SPARQL Update.

3.1.3. Marking To-Do Items as Done or Not Done

To implement toggling to-do statuses, first, the triples are formed that will function as the N3 rule premise of the rule as shown in Listing 10. These two triples will be used as data, and the N3 rules that include this rule with the policy are altogether used as the query for the reasoning with EYE. When the data vocabulary is the same as the app vocabulary, i.e. `cal`, the user does not have to provide N3 conversion rules and there are thus no rules to use as the query. That is why the implementation provides default triples to be inserted and deleted as well. In the case of marking an item as done, the default to-be-deleted triples are empty, and the default to-be-inserted triples are equal to the `?id cal:completed ?date` triple. In the case of marking an item as not done, this is the other way around with the default inserted triples being empty. These default triples are then used as the body of the N3 Patch request, otherwise the triples resulting from the reasoning task are being used. Just as the N3 Patch for adding a new to-do item is authenticated, the 3N Patch for toggling this to-do status is also authenticated. The policy retrieved from the rule closure is parsed using a SPARQL query with the help of Comunica in the same way as the to-do items were initially parsed – although with a different SPARQL query, of course.

3.2. Form Generator


formgenerator.smessie.com

Form Generator

Solid-UI | Logout

Load | private/tests/forms/084d590b-dcdf-4a60-bcf3-144503bf77f3.ttl

When loading a form resource, any changes will be discarded and the given URI will be loaded.

Beware! This is only a proof of concepts, not all field types by the vocabularies are supported.

Available form fields

Drag and drop them in your form on the right

- Input String
- Textarea
- Select Dropdown
- Date
- Checkbox

Your new form

Drag and drop the form fields in this area to reorder them

Binding	http://schema.org/Rating

Fill in the details below about where the data should be stored when someone fill in the form and click submit.

Policy Type	HTTP Request
Policy URL	https://solid.smessie.com/thesis/forms/antwoord.ttl
Policy HTTP Method	POST
Policy Content-Type	text/turtle

Fill in the URL to which the user should be sent after submitting the form.

Policy Redirect URL	https://smessaert.be

Select Dropdown

Label	Choose amount of stars
Binding	http://example.org/ratingValue

[✓] Required

[ ] Allow multiple

Choice Binding	https://solid.smessie.com/private/tests/forms/084d590b-dcdf-4a60-bcf3-

Options

★ - I didn't like it	http://example.org/NotLikedIt
★★ - I liked it	http://example.org/LikedIt

Drag here to add form field

Save

Figure 11: Screenshot of the implemented FormGenerator application.

The FormGenerator application provides an implementation of an app that allows users to build a form description. Furthermore, it can also be used to load an existing form description and edit it. As this app only functions as a proof of concept, only a limited selection of form elements is supported. Only the following form elements are supported: text input, text area (if supported by the ontology), checkbox, date field, and select. However, more field types could be added in the same way as the ones that are already supported. A screenshot of the application is shown in Figure 11.

3.2.1. Ember

This app is developed using EmberJS using Redpencil's ember-solid library and rdflib.js (see Section 3.8) which makes it easy to read from and write back to resources in a user's Solid pod just by interacting with JavaScript objects as EmberJS models. To style the layout of the application, the well-known CSS framework Bootstrap 5 is used. The drag-and-drop functionality is implemented using the ember-drag-drop add-on [56]. Authentication is handled by ember-solid as well and when the user browses to the app, he is redirected to the Solid IDP to log in.

Just as changes to the configuration of the Vue app were needed to make Comunica and the EYE reasoner work with Vue, changes to the configuration of the Ember app were needed to make Comunica and the EYE reasoner work with Ember. To help people who want to reproduce this thesis, or to do similar things, the changes that were made to the configuration of the Ember app in `ember-cli-build.js` are listed below in Listing 13.

```
'use strict';
const EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function (defaults) {
  let app = new EmberApp(defaults, {
    autoImport: {
      webpack: {
        node: {
          global: true,
        },
        resolve: {
          fallback: {
            fs: false,
            crypto: false,
            path: false,
          },
        },
      },
    },
  });
  return app.toTree();
};
```

Listing 13: Configuration changes in ember-cli-build.js to make EYE-JS and Comunica work with Ember.

3.2.2. Tackling Problems With N3 Rules

However, as easy as it is to use ember-solid for basic Solid functionality, as hard it is to do more sophisticated things that do not belong to typical operations. These limitations became apparent when implementing the part of defining the N3 rules for the policies. First, let's state two things. First, it is known that the rules defining the policies are stored in the same resource as the rest of the form description, so only one URL must be provided as input to the form renderer. Second, rdflib.js is used by ember-solid to parse the form description, and it is known that rdflib.js does not support N3 rules. Although rdflib.js' parser to parse resources is called the n3parser, it is unable to parse N3 rules. The problem goes further than this, as it is unable to parse the rest of the resource as soon as it contains N3 rules. This is not the only problem with N3 rules though; inserting and deleting N3 rules in and from a resource is unable with both SPARQL Update and N3 Patch. This is because N3 rules are not standard triples, but a N3 statement where the subject and object constitute quoted graphs that are not supported in any of the former and current Solid data manipulation protocols [2].

To tackle both problems, a similar combined type of solution to both problems has been used. The only way to insert and delete N3 rules appears to be using an HTTP PUT request with the newly updated resource as the body. To insert a new N3 rule, the resource is retrieved, the new N3 rule is locally added to the resource, and the resource is PUT back to the server. In the same way, to delete or update an existing N3 rule, the resource is retrieved, the N3 rule is adjusted or removed from the resource, and the resource is PUT back to the server. To implement this, on retrieving the resource, the N3 rules have to be parsed so it is known which N3 rules are already present in the resource to be able to update or delete them. This is done by using a RegEx to match N3 rules in the resource. The RegEx used to match N3 rules is shown in Listing 14. The RegEx is not perfect, but it works for the current use case. It is not perfect in the sense that it will also match N3 statements that use a different namespace than the `log` vocabulary used in the N3 rules' predicates. That is because the RegEx does not check the prefix in the case that `:implies` is used, allowing a user to use a different prefix for the `http://www.w3.org/2000/10/swap/log#` namespace. However, one can see this even as a feature, as the N3 parser will not only have a problem with actual N3 rules but also with other N3 statements that look like N3 rules – statements with the same syntax but another predicate. To make the parser work, these N3 statements should be filtered out before the N3 parser is called as well. Next to the `log:implies` predicate, the RegEx also supports matching N3 rules which use the `=>` syntactic sugar as well as the full predicate without the use of a prefix.

```
/\{[^{}]*}\s*(=>|[^\s{}:]*:implies|
<http:\/\/www.w3.org\/2000\/10\/swap\/log#implies>)\s*{[^{}]*}\s*\./g
```

Listing 14: RegEx to match N3 rules.

To fix the other problem where the N3 parser is unable to parse the rest of the resource as soon as it contains N3 rules, the previous solution is further extended. Because ember-solid fetches the resource on its own, a modified version with the rules filtered out cannot be passed along to ember-solid. To, however, achieve this behavior, some extra intermediate steps are performed. They are listed below.

1. The resource is fetched (GET) in the app as was done in the former solution.
2. The N3 rules are filtered out using the RegEx from Listing 14.
3. The filtered resource is PUT back to the server.
4. The resource is fetched again, this time without the N3 rules, by ember-solid.
5. Ember-solid uses this modified resource to parse the form description.
6. The resource is fetched (GET) again from the server.
7. The original resource including the N3 rules is restored from the fetched resource by adding the N3 rules and missing prefixes and is PUT back to the server.
8. The user can now interact with the form.

When ember-solid wants to store changes to the resource, the resource on the server must be the same as it was when it was fetched initially. Otherwise, rdflib.js will complain. However, the resource on the server is not the same as it was when it was fetched, as the N3 rules were filtered out and then added back to the resource on the server after it was loaded by ember-solid. To get around this, the same steps are performed again before the changes are patched to the server. Step 5 then becomes: ember-solid uses rdflib.js to perform a SPARQL Update query to update the resource with the changes made by the user, the other steps remain the same. While doing step 6 in the second scenario, adjustments made by the user while interacting with the form description are then made concurrently to the rules while adding them back to the resource. Doing so saves an extra PUT request to the server, as the rules have to be added back to the resource anyway. The step of updating the correct rule out of all possible rules existing in the resource is implemented by filtering out any existing N3 rule defining the policies that the user can enter in the FormGenerator app. Then, the rules formed out of what the user entered in the FormGenerator app are added to the resource, regardless of whether the user made adjustments to it or not because any existing rule will be filtered out anyway.

Step 7 includes restoring the prefixes used in the N3 rules, as these prefixes might be no longer defined in the resource if the prefix is not used anywhere else in the resource. This issue occurs after the resource is updated by rdflib.js, in this case, this is an unwanted side effect of rdflib.js that is overcome by restoring the prefixes. To associate the correct namespaces with the prefixes, the prefixes defined in the original resource from step 2 are used. In fact, not only the rules are matched in step 2, but also the prefixes are matched and stored in a variable. In step 7, all prefixes matched in step 2 are added back to the resource if they do not exist in the refetched resource from step 6.

3.2.3. Parsing the Policies

In the previous section, step 2 was defined as filtering out the N3 rules from the resource. These rules potentially contain the policies that the user can define in the FormGenerator app. To allow the user to adjust these policies, on loading an existing form description, the policies should be parsed and entered into the appropriate input fields. The first step was already described, which is filtering out the N3 rules from the resource. Now, out of all these rules, the rules that define the relevant policies should be selected. This is done by performing a reasoning step on each rule individually to find out which rules define a policy. For this, the EYE reasoner in the browser is used. The input data is set as the rule premise that the wanted policy should have, i.e. `_:id <http://example.org/event> <http://example.org/Submit> .`. The rule itself is set as the query for the reasoning step. In case the rule is relevant, the reasoner will output the desired policy as a rule conclusion. Next, this policy is passed along to the Comunica engine to parse it using the SPARQL query shown in Listing 15. This outputs the wanted values as JavaScript objects, which are then used to fill in the input fields in the FormGenerator app. The method and content type are marked as optional in the query, as they are only required in the case of an *HTTP request policy*. The other possible policy is a *redirect policy*, which does not require these values. By defining these values as optional, the query can be used for both types of policies. This already shows that the query might undergo some changes in the future, to support more types of policies.

```
PREFIX ex: <http://example.org/>
PREFIX pol: <https://www.example.org/ns/policy#>
PREFIX fno: <https://w3id.org/function/ontology#>

SELECT ?executionTarget ?method ?url ?contentType WHERE {
  ?id pol:policy ?policy .
  ?policy a fno:Execution .
  ?policy fno:executes ?executionTarget .
  ?policy ex:url ?url .
  OPTIONAL { ?policy ex:method ?method } .
  OPTIONAL { ?policy ex:contentType ?contentType } .
}
```

Listing 15: SPARQL query to parse the policies.

3.2.4. Prefix.cc

When building the form, the user will have to specify the bindings for each field. The binding is a URI that uniquely identifies the field. This URI will be used to identify the field when the form is rendered. To ease the process of creating a binding, one can enter the binding using a prefix and then it will automatically be expanded to the full URI using the prefix.cc API. For example, if you enter `ex:MyField` as the binding, it will automatically be expanded to `http://example.org/MyField`. If prefix.cc does not recognize the prefix, the full URI can still be entered manually.

3.3. Form Renderer

Now that a form description has been created, it can be rendered to the user. The FormRendered app as was introduced earlier in the previous chapter is used and further extended to support these new features. The third input URL, previously ignored for the moment, is now used to provide the conversion rules for the schema alignment tasks. All principles that are discussed below also apply to the FormCli app, as it is just another implementation of a form renderer app in another viewing environment. They will not be discussed separately because these changes do not significantly affect the layout or interaction with the user, only the way the application works underneath in terms of interaction with the data, which is the same for both apps.

3.3.1. Parsing the Form Description

The idea is that the user can specify a form description using any vocabulary, and then specify conversion rules to convert this form description to the vocabulary that the app understands. Adding this feature requires changes to the implementation before the form description can be parsed, as described in the previous chapter. That is because when parsing the form description, the assumption is made that the form description is already in the vocabulary that the app understands because the SPARQL queries to parse the form description are written in this vocabulary. In the case that N3 conversion rules are specified, these rules are applied to the form description. This is done by using the EYE reasoner in the browser to perform a reasoning step on the form description with the N3 conversion rules as query data. This translation step is performed in the same way as was the case for the to-do app example explained in Subsubsection 6.3.1.1: the form description is used as data and the conversion rules are used as the query for the reasoning step. This outputs the translated form description that the app is supposed to understand and is to be used for rendering the form.

3.3.2. Determining What Subjects for the Data to Use

In the case that a resource is passed along containing pre-existing data to fill into the form, the subject URI to use when writing the new data is easy to decide. This existing subject URI can simply be used again. However, in the case that no resource is passed along, deciding on the subject URI to use is not as easy. Also in the case that multiple subjects are contained in the resource that both conform to the structure and target class of the form, as was described earlier in Subsubsection 5.2.1.4, the subject URI to use is not clear. To solve these problems, multiple possibilities were considered.

1. a new random UUID is generated and used as the subject URI using the `urn:uuid:` namespace [57]
2. the user is asked to enter a subject URI
3. the URI used as the URL in the HTTP Request policy is used as the subject URI
4. one of the existing subjects in the data resource is used as the subject URI
5. a blank node is used instead of a subject URI
6. define the subject URI to use in the form description

Using blank nodes is mostly not the solution. This would lead to not being able to point to the data using a URI, as this URI does not exist. This would also make it impossible to refer to the data from other resources. Using the URI to which the data is posted is also not a good solution, as this URI is not necessarily meaningful or even a unique URI. Consider for example the case that the data is posted to a generic endpoint like `https://example.com/api/submit-form`. There is no guarantee that this URL is unique and it thus should not be used as the subject URI. Defining the subject URI to use in the form description is also not an ideal solution, because the different ontologies used to describe forms do not all have a property to define the subject URI to use. This would mean that an extra property would have to be added to the form description, one that is not defined in the ontology and thus not expected by all different apps that build or use the form descriptions. This leads to options 1, 2, and 4 as the remaining options.

Using a random UUID is a good solution, as it is guaranteed to be unique and it is a perfect solution for the case that no data is passed along. It can be used perfectly as a default subject URI in case the user does not know which subject URI to use. Asking the user to enter a subject URI is also a good solution, as it allows the user to enter a meaningful subject URI himself. However, only supporting this option would make it require the user to know what a subject URI is and how to enter it. This would make it very hard for a user to use the form renderer app without any knowledge of the Semantic Web. This is to be avoided, as the goal is to make the form renderer app as easy to use as possible for everyone. Using one of the existing subjects in the data resource is also a good solution, especially in the case that there is only one such subject. This would also be the expected behavior when editing data, as the user would expect the data to be saved to the same subject as it was retrieved from. However, when there are multiple subjects in the data resource, something should be done to decide which subject to use. This could be done by asking the user to select one of the subjects, but this would require the user to know what a subject is and how to select one.

By going through all the options, it becomes already clear that there does not seem to be a perfect solution. However, the best solution seems to be to combine the different options and use a random UUID as the default subject URI, but allow the user to select one of the existing subjects in the data resource or enter a subject URI himself. This way, the user can enter a meaningful subject URI himself, but if he does not know what a subject URI is, he can just use the default subject URI if there is no data passed along that contains a pre-existing subject URI.

3.3.3. Parsing and Executing the Policies on Submit

When the user submits the form, the policies defined in the form description should be executed. This is done by first parsing the policies from the form description. This is done the same way as it was done in the FormGenerator app described in Subsubsection 6.3.2.3 by first using reasoning to obtain the relevant N3 rules containing the policies as rule conclusions and then parsing the policies using the SPARQL query given in Listing 15. Next, the parsed policies are executed. Currently, HTTP request policies and redirect policies are supported. The policies are executed by looping over the list of parsed policies and performing the corresponding HTTP request or redirect. In the case of an HTTP request policy, the request is immediately performed using the fetch API. In the case of a redirect policy, the specified URL is kept in memory and the user is redirected to this URL after all policies have been executed. Because of this, the user will be redirected to the last specified URL in the list of policies. It, however, does not make sense to specify multiple redirect policies, so this is not a problem.

As the body of both the HTTP request and redirect policies, the data entered by the user is passed along as Turtle. The Turtle data is formed by looping over all fields in the form and generating a triple for each entered value taking into account the type of the input field. In a later stage, it could be a nice feature to take into account the content type specified in the policy and to pass along the data in the specified content type. Nonetheless, as this does not belong to the goal of this thesis, stuff is kept simple and the data is always passed along as Turtle. After all, the point here is to show that policies can be executed based on the declarative form description.

One of the underlying ideas of the thesis that was mentioned in the introduction was the fact that the users filling in the form should be able to decide where the data is stored. In this architecture, this means that the user should be able to influence the URL used in the HTTP Request policies. This was also considered in the implementation of the FormRenderer app. More precisely, the idea of displaying the defined policy URL next to the submit button of the form was considered to allow the user to customize it. However, multiple such HTTP Request policies can exist in the form description and displaying all of them would make the form look very cluttered. In a more general scenario where the user is asked to fill out the form, the person taking the form will decide where the data is stored, not the person filling out the form. Therefore, it was decided not to implement this feature in the FormRenderer application. After all, the form description resource is available to the user filling out the form, so the user can still edit the form description resource, change the policy URL to their liking, and use that changed form description resource to render the form.

3.3.4. Schema Alignment Example

For completeness, an example will be discussed here to show how the schema alignment works. Consider the form description given in Appendix B under Section B.1. This is a form description written using the SHACL ontology. Assume now that we are in a similar scenario as the FormRenderer application. This means that we only understand the Solid-UI ontology and the form description is written using the SHACL ontology. To be able to render the form, we first need to align the SHACL ontology with the Solid-UI ontology. This is done by using the N3 rules given in Appendix B under Section B.2. This is an example of N3 conversion rules that can be used to convert a form description written using the SHACL ontology to a form description written using the Solid-UI ontology. This example is not complete, but it is sufficient to illustrate the idea and supports all form fields and the options that are used in the FormGenerator and FormRenderer applications. By applying these N3 conversion rules to the form description, we obtain an equivalent form description written using the Solid-UI ontology. The resulting form description in the Solid-UI ontology is given in Appendix B under Section B.3. All entered values are still represented in the new form description, but the triples are now described using the Solid-UI ontology instead of the SHACL ontology.

4. Evaluation

In this section, the proposed architecture and implementation in order to answer the research question will be evaluated. One way of doing this is by having a look at the proof of concept apps that were built as part of this thesis to research the feasibility of the proposed architecture in practice. In what follows, the user experience of the two apps that are part of the more complex scenario will be discussed. These are the apps that are relevant to the end goal with the three-part view on Solid Web Forms, being the FormGenerator and the FormRenderer app. The FormCli app is not considered as this is just another more complex version of the FormRenderer app where one interacts with the app through the command line instead of a graphical user interface. This requires the user to have a certain level of technical knowledge which is an assumption that does not want to be made here for the form rendering part.

This user experience was evaluated by letting people interact with the apps and asking them to give feedback on their experience. For this, the users were provided with a scenario explaining what they were supposed to do with the app. This evaluation was split up into two parts, one for the FormGenerator app and one for the FormRenderer app.

4.1. Form Generator

The scenario for the FormGenerator app can be found in Appendix A under Section A.1.

Next to this written scenario, the users were also provided with a logged-in version of the FormGenerator app on https://formgenerator.smessie.com with the URL for the form location already filled in with the use of the `?form=` query parameter. Lastly, the users were also provided with a list of bindings that they could use to create the form. Here, the assumption was thus made that users interacting with this app would have some knowledge of Linked Data and the Semantic Web.

This scenario was used to evaluate the FormGenerator app with 8 users. Only people with some technical background were asked to participate in this evaluation. The feedback that was received from these users was that the app was easy to use, especially because of the drag-and-drop functionality. Also, the ability to reorder the fields by the use of drag-and-drop was seen as a nice feature. However, some noted that it would be helpful to also be able to drop a field directly in between two other fields. Multiple respondents mentioned that it would be nice to have a live preview of the form while creating it.

Even though all the positive experiences, the users did not like the fact that they had to use the bindings to create the form. They did not understand what these bindings were and even though the list of bindings they could use was provided, some expressed difficulties in finding the right binding for the right field. They rightly noted that as a restaurant owner, they don't want to know what bindings are. Additionally, after mistyping the binding, someone expressed the wish to have some sort of validation or auto-completion on the bindings to make sure that the binding is correct. While this would be a nice feature to have, this would require having all bindings be defined and then having the app check if the binding is correct. For cases where the binding definition does not exist, e.g. when the `ex:` namespace is used, this would not be possible. By just returning a warning message when the binding is not correct, the user can still continue to create the form, so this should not be a big issue. This, together with automatically suggesting a binding based on the entered field name, is out of scope for this thesis but is a useful idea for future work. Next to bindings, also the choice of vocabulary was confusing for the users. They did not understand why they had to choose between SHACL, Solid-UI, and RDF-Form and on what this choice was based. This is a valid remark as this choice is not based on anything and is just a remnant of the initial idea to have the FormGenerator app be able to generate forms based on different vocabularies. However, this option could still be useful for people with more technical knowledge that want to create a form based on a specific vocabulary.

When building a form for the SHACL vocabulary, marking a field as required and allowing multiple answers is done by specifying the `sh:minCount` and `sh:maxCount` properties. Asking users to enter the "min count" and "max count" for a field was confusing to them because they did not know what it meant. Lastly, some people noted that they expected the possibility to enter a radio button field to be able to select the score for the review. The initial idea was that this should be defined using a dropdown field, but using a radio button field would indeed be more intuitive for the user. As these apps are just a proof of concept with only a limited amount of field types implemented to show the feasibility of the proposed architecture, the radio button field is one of the field types that is not implemented yet. However, this is something that should be added in the future when building a more complete version of the FormGenerator app. Therefore, this point of feedback was, even though it was a valid remark, not considered a negative point for the FormGenerator app. Overall, the feedback on the FormGenerator app was positive and 6 out of the 8 users were able to create the form without any issues besides the difficulties with the bindings. Note however that only users with some technical background were asked to participate in this evaluation. A better way of handling the bindings more abstractly is something that should be considered for future work.

4.2. Form Renderer

The scenario for the FormRenderer app can be found in Appendix A under Section A.2.

The assumptions made for this scenario were the following:

- The user has a Solid Pod and is logged in to the FormRenderer app. This was done by providing the users with a logged-in version of the FormRenderer app on https://formrenderer.smessie.com.
- The conversion rules to go from the form description vocabulary to the form rendering app vocabulary (Solid-UI) exist and are defined in the FormRenderer app. This was done by providing the URL to the conversion rules resource by using the `?rules=` query parameter.
- The form description resource exists and is accessible by the FormRenderer app. This was done by providing the URL to the form description resource by using the `?form=` query parameter.

This scenario was then used to evaluate the FormRenderer app with 11 users. No distinction was made among users; people without a technical background also participated in the evaluation. The feedback that was received from all these users was that the app was straightforward to use, easy to use, and clear. There were no real critical issues mentioned, as to what the users get to see, it is a very simple app that does what it is expected to do. Extra points for improvement given by the users were the idea of automatically hiding the input panel when the required input fields were already filled in via the URL query parameters to reduce the amount of technical information shown to the user. After all, it does not make sense to show the input fields for the form description resource and the conversion rules if you send it to someone who just wants to fill in the form specified by the sender. Furthermore, someone noted that they expected a multi-line text field to be used for the review field instead of a single-line text field. However, this was a consequence of the fact that the form description was built using the SHACL vocabulary and the SHACL vocabulary does not allow one to define a multi-line text field. This also immediately shows empirically by people that SHACL is a vocabulary made to express validation and not to describe the display part, as pointed out earlier. In the case of using a vocabulary to describe the display part, an ontology made for that purpose should be used, such as Solid-UI. Some mixed feedback was given on the date field. Some people liked the fact that the date field already had the separating dashes in it, while others did not like that they had to click another time on the calendar icon to be able to enter a date via the popup calendar. Furthermore, one person noted that it was unclear what the Subject URI was for, and even though for people without that knowledge there is always at least one valid suggestion that can be used, it can be confusing because they do not know what to choose. Besides that, the users did not notice that the app was using Solid and Linked Data behind the scenes and this is exactly the goal of the FormRenderer app. People who were given a form described using the SHACL vocabulary were unaware that schema alignment tasks were being performed behind the scenes. Lastly, one person noted that a "Copy URL" button next to the load button, after you entered the input fields, would be a nice addition to the app. To conclude, the feedback on the FormRenderer app was positive and all the participating users were able to fill in the form without any issues. The app was straightforward and users did not realize that Solid and Linked Data were involved behind the scenes. This is a good thing as it means that the app is easy to use for people without any knowledge of Solid and Linked Data.

5. Discussion

The concept of *schema alignment tasks* was successfully introduced and demonstrated by the To-Do App. The source code of this app can be found at https://github.com /smessie/TAS and a live version of the app can be found at https://tas.smessie.com. Now, Bob can translate Alice's to-do list into a language that his to-do app understands by just providing the app with a set of conversion rules. The translation will then be done automatically thanks to the reasoning. This concept was then later also successfully applied to the FormRenderer app allowing any form description to be inputted into the app and be rendered by the app as long as a conversion rules resource exists and is provided that can translate the form description into the form rendering app vocabulary.

Furthermore, the concept of *footprint tasks* was successfully introduced and demonstrated by the FormGenerator app. The source code of this app can be found at https://github.com/smessie/FormGenerator and a live version of the app can be found at https://formgenerator.smessie.com. Support for these was also added to the FormRenderer and FormCli apps. Because of this, actions that need to be performed in case of certain events can be defined in a declarative way in the form description allowing machines to perform these actions automatically. This was demonstrated by the form renderer apps that automatically store the form data in a Solid pod by performing the HTTP request as defined in the form description, followed by a redirect of the user to the URL that was defined in the policy in the form description as well. However, no standardized ontology to describe these events and actions exists yet. The FnO ontology was used to describe that an action should be performed, but the action itself, like the event, was not described in a standardized way. This is something that should be considered for future work.

The user experience evaluation made clear that the FormGenerator and FormRenderer apps are meant for different types of users. This was also the thought beforehand but during the evaluation, this became extra clear. The FormGenerator app was only evaluated by people with a technical background, yet they still had minor difficulties with understanding everything. The FormRenderer app was evaluated by people with and without a technical background and none of them had any difficulties with understanding the app.

The research question for this thesis was the following: “How can machines be controlled in a declarative way to perform **schema alignment and footprint tasks** by the use of reasoning?”. Given the successful implementation of the applications that demonstrate how to define policies and how to perform schema alignment and footprint tasks, in addition to the positive results of the user experience evaluation, the research question can be answered in the affirmative. Nonetheless, the results of these applications show the need for further research to further improve the perceived accessibility issues regarding bindings in order to make these technologies optimally available to all people without expecting them to have prior technical knowledge.

Chapter 7: Uniform Reasoner Interface

Reasoning is a core part of the proposed architecture. Both schema alignment and footprint tasks, as discussed in the previous chapter, require reasoning to be performed. Every use case differs from the other and therefore requires a different way of executing reasoning. Some use cases will benefit the most from reasoning in the browser, while others will be too computationally heavy requiring them to be executed remotely. Some use cases will be more performant when using a specific reasoner, while others will be more performant when using another reasoner implementation. Because of the importance of allowing developers to easily switch between reasoners as their use case changes, a uniform interface is needed to abstract away the differences between the different reasoners. This chapter will discuss the design and implementation of such a possible interface by trying to find an answer to the third and last research question of this thesis. This question was: “How can an **abstraction** be made to **run reasoning** in the browser or remotely?”. Just as was the case for the previous chapters, first the architecture of the abstraction will be discussed, followed by the implementation details and a discussion of the results to conclude the chapter.

1. Architecture

A *uniform reasoner interface* should be designed to abstract away the differences between the different reasoners. Doing so will not only allow developers to easily implement reasoning in their applications without knowing all the internal details of the reasoner, but it will also allow them to easily switch between reasoners as their use case changes. Switching between reasoners can mean switching between reasoning in the browser or remotely, or it can mean switching between reasoner implementations to improve performance. The interface should be designed in such a way that it is easy to implement for the different reasoners, but also easy to use for the developers. It should reflect all the possibilities of the reasoner, while still offering a uniform interface. In what follows, a proposal for such an interface will be discussed.

First, the `data` and `query` parameters are needed. The `data` parameter is used to pass the data to the reasoner together with any inference rules that should be applied. The `query` parameter is optional and defines the pattern of the data that should be returned by the reasoner. By leaving this parameter undefined, all inferred facts will be returned. The data and query can be passed as a string, or it can be passed as an array of Quads. In the case of the latter, the `Quad` type of the RDF/JS library (`@rdfjs/types`) is used when working in a JavaScript or TypeScript environment. When passed as a string, the data should be formatted in the Notation3 syntax. Furthermore, the interface is designed with extensibility in mind. This is done by using a single object that contains all the additional options that can be passed to the reasoner. This object can be extended by other reasoners, allowing them to add their options. When a reasoner does not recognize or support an option, it should inform the user of this. By default, the output type should be the same as the input type. However, by passing the `outputType` option, the user can specify the output type. This option must support at least the `string` value, which will return the output as a string in the Notation3 syntax. It should also support the `quads` value, which will return the output as an array of RDF/JS Quads.

When the query parameter is left undefined, the user should have the option to execute implicit queries. This is expressed in the options object by the `output` option by defining what to output with implicit queries. The default is undefined, meaning that no implicit query is passed. The user can pass the `derivations` value to output only new derived triples. The `deductive_closure` value can be passed to output the deductive closure. To output the deductive closure plus the rules, the `deductive_closure_plus_rules` value can be passed. Finally, the `grounded_deductive_closure_plus_rules` value can be passed to ground the rules and output the deductive closure plus the rules.

Last, the option `blogic` can be defined to use blogic [58]. When true, the reasoner should use blogic, used to support RDF Surfaces [59]. When false, the reasoner should use the default reasoning algorithm. RDF Surfaces is a Notation3 sublanguage for representing a collection of zero or more RDF graphs as a sheet of paper with those RDF graphs on it. It is a language to express first-order logic in RDF that was proposed by Hochstenbach and De Roo and for which support was added to EYE. It is an implementation of the ideas of blogic by Hayes.

2. Implementation

2.1. Remote EYE Execution

At the time of starting this thesis, the EYE reasoner was not yet available as a JavaScript library. The only way to execute reasoning queries was to use the command line interface (CLI) of the EYE reasoner. To execute reasoning queries in the browser, the queries were first sent to a server, where the EYE CLI was used to execute the query, and the results were sent back to the browser. This was not ideal, as this heavily relies on a server being available. This also brings the additional cost of doing an HTTP request to the server, which would not be needed if the EYE reasoner could be executed in the browser. As a first step in the right direction, an `eye-mock` library [60] was created that provides a mock implementation of the EYE reasoner in the browser. It does this by internally doing an HTTP request to a server running the EYE CLI, just like the previous implementation. This library is just a wrapper around this HTTP request although the advantage of wrapping it in a library is that it can be easily replaced with a real implementation of the EYE reasoner in the browser. Because of that, it seems to the developer using the library that the reasoning is executed client-side in the browser, while in reality, it is still executed on the server. The goal of this EYE mock is to propose a standard interface for reasoning libraries in the browser so that the implementation of the EYE reasoner in the browser can be easily swapped with another reasoning library. Therefore, the interface as described in the previous section was implemented in this library and is displayed in Listing 16. The existing server implementation accepting HHTP requests by Van Woensel [61] was further extended to support the new interface and then used as the server for the EYE mock library. This extended version can be found at https://github.com/smessie/n3-editor-js.

This server implementation expects the data and query to be passed as a string. Furthermore, it will return the output as a string in the Notation3 syntax. The EYE mock library supports the `string` and `quads` values for the `outputType` option. When the `quads` value is passed, the output will be converted to an array of RDF/JS Quads. To do so, the N3 parser of the N3.js library [32] is used, which is a JavaScript library for parsing and serializing RDF in the Notation3 syntax. However, due to some limitations of the N3.js library [62] where the Writer does not support N3, only passing the input data and query as a string is currently supported.

```
import { Quad } from '@rdfjs/types';

export interface IQueryOptions {
    blogic?: boolean;
    outputType?: 'string' | 'quads'
    output?: undefined | 'derivations' | 'deductive_closure' | 'deductive_closure_plu
}

declare module "eye-mock" {
    export function n3reasoner(data: Quad[] | string, query?: Quad[] | string | unde
}
```

Listing 16: Implementation of the uniform reasoner interface of the EYE mock library.

2.2. EYE-JS

While working on this thesis with the earlier explained EYE mock, the EYE reasoner was made available as a JavaScript library by Wright [30]. As described earlier in Subsection 3.6.2, this library uses the new technology of SWI-Prolog in the browser using WASM. As a function of this thesis, the same uniform interface as the EYE mock was contributed to the EYE-JS library. This allows easy switching between the EYE mock and the EYE-JS library by just changing the import statement. This shows the power of the uniform interface and what is possible with it in the future if other reasoning libraries would follow the same proposed standard.

2.3. Reasoner App

The Eye Reasoner app is a playground application that allows one to execute reasoning queries over some data in the browser. The goal of this application is to research the feasibility of executing reasoning queries in the browser and to provide a simple interface to do so. This application was implemented as a second application after the FormGenerator. The outcome of this app was later used in the implementation of other applications: the to-do app and the form renderer. In addition to testing out the capabilities of reasoning in the browser and overcoming perceived challenges, it also provides a perfect tool for researchers and others engaged in reasoning to quickly and very easily run queries on their data using the EYE reasoner. The Reasoner app supports both reasoning via the browser via the EYE-JS library and reasoning via the server via the EYE mock library. The user can use the toggle to switch between the two implementations to fit their needs. Executing the reasoning in the browser will not have the overhead of an HTTP request to the server while executing the reasoning on the server will be able to execute more complex queries that would otherwise not be possible in the browser by using the full server resources.

The input data and query can be passed in the text area or by providing a URL to the data and query. In the case of the latter, authentication, as discussed earlier in Subsubsection 5.2.1.2, is implemented to allow the user to provide a URL to a private resource in a Solid pod. A toggle to enable or disable the support for blogic reasoning is also provided. This value is then passed along to the EYE reasoner. Because of this, this playground application can also be used to test out RDF Surfaces.

3. Discussion

This chapter began with the expression of the desire to be able to easily switch between different reasoning libraries by the use of a uniform interface. In Section 7.1 such an interface was proposed which then was implemented as discussed in Section 7.2. Given the successful implementation of this interface in the EYE mock and EYE-JS libraries, it is now possible to easily switch between these two libraries. This is demonstrated in the Reasoner app, where the user can switch between the two libraries by just clicking a toggle. As a developer, this switching between the 2 reasoners was extremely easy to implement as the only thing that had to be changed was the import statement. This shows the power of the uniform interface and what is possible with it in the future if other reasoning libraries would follow the same proposed standard.

The posed question "How can an **abstraction** be made to **run reasoning** in the browser or remotely?" can thus be successfully answered. The implementation of the proposed interface in the EYE mock and EYE-JS libraries shows how this abstraction can be made. Furthermore, the implementation of the Reasoner app shows how this abstraction can be used to run reasoning in the browser or remotely. The source code of this Reasoner app can be found at https://github.com/smessie/reasoner-app and a live version of this application can be found at https://reasoner.smessie.com.

As future work, it would be interesting to see this interface implemented in other reasoning libraries, especially in a library that implements a different algorithm than the EYE reasoner. This would show if there are any shortcomings in the proposed interface and would allow a developer to easily switch to another reasoner in the case that the EYE reasoner would not perform well on their use case. As additional future work, it would be nice to have an HTTP server version of the interface so that the interface can be used in a server environment as well, without the need for an additional library like the EYE mock. This interface should exist of the same parameters as the proposed interface in Section 7.1.

Chapter 8: Conclusion

We started in the introduction with the scenario where Alice tried to edit and reuse Bob's form to avoid having to build a new form from scratch or edit where the data is stored. In addition, Bob wanted to be able to render the form with his favorite application, regardless of which one Alice uses. However, they found themselves in a centralized and strongly coupled network of service providers and their respective web interfaces. In this thesis, an alternative decentralized and decoupled architecture is presented to solve Alice and Bob's problems. When Bob now sends a form to Alice, he will send the form description to Alice, and Alice will be able to render the form in her own viewing environment. Thanks to the declarativity of the form description, all semantics are contained in it, i.e. the form description contains all the information needed to render the form and to decide what to do with the submitted data. Alice's form renderer is now able to render the form without the need to make any assumptions. When Alice clicks the submit button, the form renderer will perform the footprint tasks by executing the policies that are defined in the form description. However, when Alice wants to send the data she entered somewhere else, she can edit the form description and change the policies, after which she can then use this updated form description to render the form again. This way, Alice has full control over the form and the data she enters into it.

In the same way, Alice can now reuse Bob's form description to create a new form that is similar to his form, edit it to her needs, and then send it to Charlie. Alice saved a lot of time by not having to build a new form from scratch. Furthermore, they can use the FormRenderer implemented in this thesis to not only render the form but also to fill in the form with pre-existing data. In the case that Charlie has already filled out Bob's form, and then Alice sends her form to Charlie, Charlie can use the FormRenderer to fill out Alice's form with the data he already entered in Bob's form, and only need to fill in the data that is different between the two forms.

By giving Alice the possibility to edit the form description, thus changing where the data will be stored, and also supporting multiple viewing environments, the three-part architecture allows for decentralization and decoupling of web forms. However, we realized that this move to decentralization and decoupling comes with its own challenges. Decoupling also means that another app can be used to render the form, but the ability to use any form renderer does not necessarily imply that that form renderer app will understand the same ontology. Schema alignment tasks were proposed and implemented to be able to translate a resource from one ontology to another. First, a to-do application was implemented to detail this technique. This makes the used ontology of the resource independent of the ontology used by the application, as long as both ontologies are similar enough in the sense that they can be mapped to each other.

To prove that another app can be used to render the form, a FormCli app was implemented. Both are form renderer applications, but the FormRenderer is a web application and the FormCli is a command-line application. This shows that the display part is not bound to a specific viewing environment as this proved that the same form description can be rendered both in a web browser using HTML and in a text-based terminal.

The implemented FormRenderer app implementing schema alignment and footprint tasks got a lot of positive feedback from the user-experience evaluation. It seemed to be a very intuitive form renderer, where the users did not notice that schema alignment was happening in the background. The only thing that was noticed was that the people filling in a SHACL form had no multi-line text field for the review description (because it is not supported in SHACL), while they expected one instead of a single-line text field. Next to that, the users that evaluated the FormGenerator app gave the feedback that the SHACL way of defining if a field is required or if multiple values are allowed with “sh:minCount” and “sh:maxCount” is not intuitive. This makes it clear that the SHACL ontology is not ideal for the display part. It was stated earlier, but now it is also empirically shown that the Solid-UI ontology is more natural for the display part. The FormRenderer can be used for a lot of different simple use cases, but it is not yet ready for more complex use cases. This is because it was built as a proof of concept where not all form elements are implemented yet, just as is the case for the FormGenerator.

A proof of concept was also implemented for the footprint tasks. Notation3 proved to be a viable option as a rule language, just as FnO proved to be a suitable ontology as a policy language. The thesis showed that policies can be used to describe what should happen in a given event, but no standardized way of defining such policies had been proposed. However, this should be a future work item. Furthermore, once such a policy is standardized, it would be nice to have a `policy-executor` library that handles the execution of the policy, so that applications do not have to implement this themselves. This would improve extensibility; new policies could be added in one place, the library, and all applications using that library would automatically support the new policy by simply updating the library. This opens the door for more advanced policies, and also for more standardized policies since the library could be used by many applications.

Implementing these policies with N3 showed that more work is needed regarding N3 rules. This insight was gained when it was discovered that N3 rules could not be patched with N3 Patch, nor could they be patched with SPARQL Update. Furthermore, the N3Parser cannot parse a resource that contains N3 rules (even though the library is called N3.js, and N3 rules are part of the N3 specification). This is cumbersome and must be resolved in order to make the use of N3 rules more convenient.

To execute the footprint tasks and the schema alignment tasks, reasoning was used. First, the reasoning was done remotely on a server, but later the switch was made to client-side reasoning in the browser, removing the need for a server. However, to allow easy switching between the two, a uniform reasoner interface was implemented. This interface was implemented in the EYE-JS library and the EYE mock library, which is a package around a remote execution of the EYE reasoner. A simple Reasoner application was then implemented using these packages allowing to reason over a resource giving the ability to easily switch between local and remote reasoning. This shows that such a uniform reasoner interface is possible and can be used to switch easily between different reasoners in the browser or remotely. As future work, it would be interesting to see this interface implemented in other reasoning libraries, especially in a library that implements a different algorithm than the EYE reasoner. Additionally, it would be nice to have an HTTP server version of the interface so that the interface can be used in a server environment as well, without the need for an additional library.

Another point of feedback received from the user-experience evaluation was that the users were confused by the bindings in the FormGenerator and did not fully understand them. It would be nice to further abstract the bindings away from the user so that the user does not have to think about them. This could be done by automatically generating and suggesting the bindings based on what label the user enters for the field. This is beyond the scope of this thesis but could be a topic for future research.

This thesis solved a large part of the problem to create a purely declarative way to create Solid web forms. The validation part was not extensively examined in this thesis. Some existing work on validation was discussed, but no new work was done. For future work, it could be interesting to implement the validation part in the different applications as was proposed in the architecture and see the whole architecture in action.

Bibliography

[1] T. Berners-Lee and others, “Solid.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://solidproject.org

[2] D. Arndt, W. Van Woensel, D. Tomaszuk, and G. Kellogg, “Notation3.” 2022. Accessed: Nov. 26, 2022. [Online]. Available: https://w3c.github.io/N3/spec/

[3] R. Verborgh, “Shaping Linked Data apps.” 2022. Accessed: Dec. 01, 2022. [Online]. Available: https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/

[4] E. Bruchez, A. Couthures, and P. Steven, “XForms 2.0 - XForms Users Community Group.” 2022. Accessed: Oct. 17, 2022. [Online]. Available: https://www.w3.org/community/xformsusers/wiki/XForms_2.0

[5] H. Knublauch and D. Kontokostas, “Shapes Constraint Language (SHACL).” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/TR/shacl/

[6] SolidOS, “Solid-UI.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/ns/ui#

[7] P. Hochstenbach, J. Wright, and T. Turdean, “RDF forms for Solid.” Jul. 2022. Accessed: May 24, 2023. [Online]. Available: https://solidos.solidcommunity.net/public/2022/RDF%20forms%20for%20Solid/

[8] I. Smessaert, “Google Forms but the Solid way.” Sep. 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://smessaert.be/blog/google-forms-but-the-solid-way/

[9] D. Beeke, “RDF Form.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://rdf-form.danielbeeke.nl

[10] M. Lanthaler and C. Gütl, “Hydra: A Vocabulary for Hypermedia-Driven Web APIs.,” *LDOW*, vol. 996, pp. 35–38, 2013.

[11] MetaSolutions, “RDForms - RDF in HTML-forms.” Accessed: May 08, 2023. [Online]. Available: https://rdforms.org/#!index.md

[12] W. Slabbinck, “Interoperabiliteit tussen applicaties met behulp van Solid voor het beheren van sensordata in slimme woningen /.” 2021. Available: https://lib.ugent.be/catalog/rug01:003014963

[13] E. Prud’hommeaux and J. Bingham, “Shape Trees Specification.” Dec. 2021. Accessed: Apr. 20, 2023. [Online]. Available: https://shapetrees.org/TR/specification/

[14] W. Slabbinck, “CommunitySolidServer/shape-validator-component.” Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/CommunitySolidServer/shape-validator-component

[15] Zazuko, “rdf-validate-shacl.” zazuko, Feb. 2020. Accessed: Apr. 21, 2023. [Online]. Available: https://github.com/zazuko/rdf-validate-shacl

[16] T. Bergwinkl, “shacl-engine.” rdf-ext, Jan. 2023. Accessed: Apr. 21, 2023. [Online]. Available: https://github.com/rdf-ext/shacl-engine

[17] T. Bergwinkl, “Implementing a 15x faster JavaScript SHACL Engine,” *bergis universe of software, hardware and ideas*. Mar. 2023. Accessed: Apr. 21, 2023. [Online]. Available: https://www.bergnet.org/2023/03/2023/shacl-engine/index.html

[18] T. Berners-Lee, J. Hendler, and O. Lassila, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[19] T. Berners-Lee, “Linked Data - Design Issues.” Jul. 2006. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html

[20] G. Schreiber and Y. Raimond, “RDF 1.1 Primer.” Jun. 2014. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/rdf11-primer/

[21] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “RDF 1.1 Turtle.” Feb. 2014. Accessed: Nov. 28, 2022. [Online]. Available: https://www.w3.org/TR/turtle/

[22] G. Kellogg, P.-A. Champin, D. Longley, M. Sporny, and M. Lanthaler, “JSON-LD 1.1.” Jul. 2020. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/json-ld/

[23] F. Gandon and G. Schreiber, “RDF 1.1 XML Syntax.” Feb. 2014. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/rdf-syntax-grammar/

[24] I. Herman, B. Adida, M. Sporny, and M. Birbeck, “RDFa 1.1 Primer - Third Edition.” Mar. 2015. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/rdfa-primer/

[25] D. Beckett, “RDF 1.1 N-Triples.” Feb. 2014. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/n-triples/

[26] S. Harris and A. Seaborne, “SPARQL 1.1 Query Language.” Mar. 2013. Accessed: May 22, 2023. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[27] S. Capadisli, T. Berners-Lee, R. Verborgh, and K. Kjernsmo, “Solid Protocol.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://solidproject.org/TR/protocol

[28] J. Halliday, “Tim Berners-Lee: Facebook could fragment web,” *The Guardian*, Nov. 2010, Accessed: May 23, 2023. [Online]. Available: https://www.theguardian.com /technology/2010/nov/22/tim-berners-lee-facebook

[29] R. Verborgh and J. De Roo, “Drawing Conclusions from Linked Data on the Web: The EYE Reasoner.” 2022. Accessed: Oct. 20, 2022. [Online]. Available: http://ieeexplore.ieee.org/document/7093047/

[30] W. Jesse and J. De Roo, “EYE JS.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/eyereasoner/eye-js

[31] W. Van Woensel, “jen3.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/william-vw/jen3

[32] R. Verborgh, “N3.js.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/rdfjs/N3.js

[33] P. Bonte, “RoXi.” 2022. Accessed: Mar. 05, 2023. [Online]. Available: https://github.com/pbonte/roxi

[34] S. Ceri, G. Gottlob, L. Tanca, and others, “What you always wanted to know about Datalog(and never dared to ask),” *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[35] J. Wielemaker, “SWI-Prolog in the browser using WASM - Wiki.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://swi-prolog.discourse.group/t/swi-prolog-in-the-browser-using-wasm/5650

[36] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, “Comunica: a Modular SPARQL Query Engine for the Web,” Oct. 2018. Available: https://comunica.github.io/Article-ISWC2018-Resource/

[37] R. Taelman, “Components.js.” Accessed: Mar. 29, 2023. [Online]. Available: http://componentsjs.readthedocs.io/en/latest/

[38] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern.” Jan. 2004. Accessed: Mar. 29, 2023. [Online]. Available: https://martinfowler.com/articles/injection.html

[39] redpencil.io, “Ember Solid.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://github.com/redpencilio/ember-solid

[40] Tilde, “Ember.js - A framework for ambitious web developers.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://emberjs.com

[41] Tilde, “Ember Data.” 2022. Accessed: Mar. 29, 2023. [Online]. Available: https://guides.emberjs.com/release/models/

[42] T. Berners-Lee, “rdflib.js.” 2011. Accessed: Mar. 29, 2023. [Online]. Available: https://github.com/linkeddata/rdflib.js

[43] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, “Implementation-independent function reuse,” *Future Generation Computer Systems*, vol. 110, pp. 946–959, 2020.

[44] B. De Meester, A. Dimou, and F. Kleedorfer, “The Function Ontology.” 2021. Accessed: Mar. 06, 2023. [Online]. Available: https://w3id.org/function/spec

[45] A. Miles and S. Bechhofer, “SKOS simple knowledge organization system reference,” *W3C recommendation*, 2009, Available: https://www.w3.org/TR/skos-reference/

[46] M. Vander Sande, P. Hochstenbach, R. Dedecker, and J. Werbrouck, “Orchestrator for a decentralized Web network.” 2021. Accessed: Mar. 06, 2023. [Online]. Available: https://mellonscholarlycommunication.github.io/spec-orchestrator

[47] P. Hochstenbach, “Koreografeye.” 2022. Accessed: Mar. 06, 2023. [Online]. Available: https://github.com/eyereasoner/Koreografeye

[48] T. Berners-Lee, “Linked Data Shapes, Forms and Footprints - Design Issues.” Apr. 2019. Accessed: Apr. 15, 2023. [Online]. Available: https://www.w3.org/DesignIssues/Footprints.html

[49] D. Simonds, “Break down these walls,” *The Economist*, Mar. 2008, Accessed: May 09, 2023. [Online]. Available: https://www.economist.com/leaders/2008/03/19/break-down-these-walls

[50] Inrupt, “Session Restore upon Browser Refresh — Inrupt JavaScript Client Libraries.” 2023. Accessed: Apr. 06, 2023. [Online]. Available: https://docs.inrupt.com/developer-tools/javascript/client-libraries/tutorial/restore-session-browser-refresh/

[51] S. Boudrias, “SBoudrias/Inquirer.js: A collection of common interactive command line user interfaces.,” *GitHub*. May 2013. Accessed: Mar. 27, 2023. [Online]. Available: https://github.com/SBoudrias/Inquirer.js

[52] A. Havermale, “haversnail/inquirer-date-prompt: A date prompt plugin for Inquirer.js.,” *GitHub*. Jan. 2021. Accessed: Mar. 27, 2023. [Online]. Available: https://github.com/haversnail/inquirer-date-prompt

[53] Inrupt, “Authenticate (Node.js: Single-User App) — Inrupt JavaScript Client Libraries.” Accessed: May 05, 2023. [Online]. Available: https://docs.inrupt.com/developer-tools/javascript/client-libraries/tutorial/authenticate-nodejs-script/

[54] J. Prass Martins, “Big integer literals are not available in the configured target environment (vite3 + vMoney3 + vue3).” 2022. Accessed: Apr. 06, 2023. [Online]. Available: hhttps://github.com/jonathanpmartins/v-money3/issues/70#issuecomment-1284503693

[55] P. Gearon, A. Passant, and A. Polleres, “SPARQL 1.1 Update.” Mar. 2013. Accessed: Dec. 12, 2022. [Online]. Available: https://www.w3.org/TR/sparql11-update/

[56] M. Harris, “ember-drag-drop.” 2021. Accessed: Sep. 11, 2022. [Online]. Available: https://github.com/mharris717/ember-drag-drop

[57] P. Leach and M. Mealling, “A Universally Unique IDentifier (UUID) URN Namespace.” Jul. 2015. Accessed: May 04, 2023. [Online]. Available: https://www.ietf.org /rfc/rfc4122.txt

[58] P. Hayes, “BLOGIC. (ISWC 2009 Invited Talk).” Oct. 2009. Accessed: May 12, 2023. [Online]. Available: https://www.slideshare.net/PatHayes/blogic-iswc-2009-invited-talk

[59] P. Hochstenbach and J. De Roo, “RDF Surfaces Primer.” 2023. Accessed: Apr. 06, 2023. [Online]. Available: https://w3c-cg.github.io/rdfsurfaces/

[60] I. Smessaert, “eye-mock.” 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://github.com/smessie/eye-mock

[61] W. Van Woensel, “william-vw/n3-editor-js: A Notation3 Editor in JavaScript.” Accessed: Dec. 22, 2022. [Online]. Available: https://github.com/william-vw/n3-editor-js

[62] J. Wright, J. De Roo, and R. Verborgh, “Writer does not support Notation3 · Issue #316 · rdfjs/N3.js,” *GitHub*. Dec. 2022. Accessed: May 12, 2023. [Online]. Available: https://github.com/rdfjs/N3.js/issues/316

Appendices

A. User Experience Scenarios

The following scenarios were given to the participants to measure the user experience of the implemented applications. They explain how the participants should use the applications and what they should do.

A.1. Scenario 1: The Form Generator

> You have a restaurant and you would like to offer customers the opportunity to leave a **review** on the dishes they have eaten. To do this, you create a **form** where the customer can enter the **name of the dish** along with the **date of visit**. The review consists of a score between 1 and 3 (1★ - I didn't like it, 2★★ - It was tasty or 3★★★ - It was excellent). In addition, provide an option to **substantiate their choice**.
>
> At the top right, it is free to choose between SHACL, Solid-UI and RDF-Form. This has no further importance in the construction of the form.
>
> Since Linked Data is being used in the background, each field and also the form itself must contain a **binding** to link everything to existing data. This is done through the **binding** field. Below is a list of existing bindings that can be used.
>
> - schema:Rating or http://schema.org/Rating
> - dc:title or http://purl.org/dc/elements/1.1/title
> - schema:ratingExplanation or http://schema.org/ratingExplanation
> - schema:ratingValue or http://schema.org/ratingValue
> - ex:NotLikedIt or http://example.org/NotLikedIt
> - ex:LikedIt or http://example.org/LikedIt
> - ex:LovedIt or http://example.org/LovedIt
> - schema:orderDate or http://schema.org/orderDate
>
> After the customer saves the review you want an **HTTP request** of **type PUT** to be sent to https://solid.smessie.com/thesis/forms/antwoord-x.ttl, for this the **text/turtle** Content-Type is used. Finally, you also want to redirect the customer to a **website of your choice** after completing the form.

A.2. Scenario 2: The Form Renderer

> You went to eat at a restaurant where they asked you to leave a review about the dish you ate. In the form, enter a review about a dish of your choice (e.g., what you ate today or yesterday). Then submit the form by choosing a subject URI of your choice for the data.

B. Schema Alignment Example

B.1. Form Description in SHACL

```
@base <https://solid.smessie.com/thesis/forms/description-2.n3> .
@prefix shacl: <http://www.w3.org/ns/shacl#> .
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix mu: <http://mu.semte.ch/vocabularies/ext/listNodes/> .
@prefix schema: <http://schema.org/> .

<#242ca810-0b3e-4112-8144-934ebb1779dc> a shacl:NodeShape;
    shacl:targetClass schema:Rating;
    shacl:property <#41b9c368-2b2d-4bf1-94b9-679a93297103>, <#f9cc6fbd-e844-4588
<#41b9c368-2b2d-4bf1-94b9-679a93297103> a shacl:PropertyShape;
    shacl:datatype xsd:string;
    shacl:path <http://purl.org/dc/elements/1.1/:title>;
    shacl:order 0;
    shacl:name "Name of the meal";
    shacl:minCount 1;
    shacl:maxCount 1.
<#f9cc6fbd-e844-4588-b3bd-c45a63acac03> a shacl:PropertyShape;
    shacl:datatype xsd:date;
    shacl:path schema:orderDate;
    shacl:order 1;
    shacl:name "Date of your visit";
    shacl:minCount 1;
    shacl:maxCount 1.
<#69c1cf77-4a0b-4951-a374-4612d113dade> a shacl:PropertyShape;
    shacl:path schema:ratingValue;
    shacl:order 2;
    shacl:name "Rating";
    shacl:minCount 1;
    shacl:maxCount 1;
    shacl:nodeKind shacl:IRI;
    shacl:in mu:be809b07-3ded-4a91-be53-2b923d565d5c.
<#7267c9f3-5d1d-4621-8f83-027931bf1072> a shacl:PropertyShape;
    shacl:datatype xsd:string;
    shacl:path schema:ratingExplanation;
    shacl:order 3;
    shacl:name "Argumentation";
    shacl:minCount 0;
    shacl:maxCount 1.
ex:NotLikedIt a owl:Class;
    rdfs:label "I didn’t like it".
ex:LikedIt a owl:Class;
    rdfs:label "It was tasty".
ex:LovedIt a owl:Class;
    rdfs:label "It was excellent".
```

```
mu:be809b07-3ded-4a91-be53-2b923d565d5c rdf:rest mu:3900b524-272e-44ae-89c6-0979
    rdf:first ex:NotLikedIt.
mu:3900b524-272e-44ae-89c6-09796e708780 rdf:rest mu:e81a19ad-5493-4781-86a4-66be
    rdf:first ex:LikedIt.
mu:e81a19ad-5493-4781-86a4-66be00e1f728 rdf:rest rdf:nil;
    rdf:first ex:LovedIt.
```

B.2. N3 Rules to Map SHACL to Solid-UI

```
@prefix shacl: <http://www.w3.org/ns/shacl#>.
@prefix ui: <http://www.w3.org/ns/ui#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@prefix list: <http://www.w3.org/2000/10/swap/list#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix string: <http://www.w3.org/2000/10/swap/string#> .

{
    ?uri a shacl:NodeShape;
        shacl:targetClass ?binding.
} => {
    ?uri a ui:Form;
        ui:property ?binding;

}.


{
    ?uri a shacl:NodeShape .
    ( ?field { ?uri shacl:property ?field } ?List ) log:collectAllIn _:x .
} => {
    ?uri ui:parts ?List .
}.


{
    ?uri a shacl:PropertyShape;
        shacl:datatype xsd:date;
        shacl:path ?binding.
} => {
    ?uri a ui:DateField;
        ui:property ?binding.
}.


{
    ?uri a shacl:PropertyShape;
        shacl:datatype xsd:boolean;
        shacl:path ?binding.
```

```
} => {
    ?uri a ui:BooleanField;
        ui:property ?binding.
}.


{
    ?uri a shacl:PropertyShape;
        shacl:nodeKind shacl:IRI;
        shacl:path ?binding.
} => {
    ?uri a ui:Choice;
        ui:property ?binding.
}.


{
    ?uri a shacl:PropertyShape;
        shacl:datatype xsd:string;
        shacl:path ?binding.
} => {
    ?uri a ui:SingleLineTextField;
        ui:property ?binding.
}.


{
    ?uri shacl:order ?order.
} => {
    ?uri ui:sequence ?order.
}.


{
    ?uri shacl:name ?name.
} => {
    ?uri ui:label ?name.
}.


{
    ?uri shacl:minCount ?minCount.
    ?minCount math:greaterThan 0.
} => {
    ?uri ui:required true.
}.


{
    ?uri shacl:maxCount ?maxCount.
    ?maxCount math:greaterThan 1.
} => {
    ?uri ui:multiple true.
}.

{
    ?uri a shacl:PropertyShape;
        shacl:nodeKind shacl:IRI;
        shacl:in ?options.
    ?options list:iterate ( ?i ?option ) .
    ?option rdfs:label ?label .
    ?uri log:uri ?uriString .
    ( ?uriString "-options" ) string:concatenation ?optionsUriString .
    ?optionsUri log:uri ?optionsUriString .
} => {
    ?uri ui:from ?optionsUri .
    ?optionsUri a owl:Class .
    ?option a ?optionsUri ;
        skos:prefLabel ?label .
} .
```

B.3. Resulting Solid-UI Form Description

```
@base <https://solid.smessie.com/thesis/forms/description-2.n3> .
@prefix ex: <http://example.org/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix ui: <http://www.w3.org/ns/ui#>.
@prefix skos: <http://www.w3.org/2004/02/skos/core#>.
@prefix schema: <http://schema.org/> .

<#242ca810-0b3e-4112-8144-934ebb1779dc> a ui:Form ;
    ui:property schema:Rating ;
    ui:parts (<#41b9c368-2b2d-4bf1-94b9-679a93297103> <#f9cc6fbd-e844-4588-b3bd-
<#41b9c368-2b2d-4bf1-94b9-679a93297103> a ui:SingleLineTextField ;
    ui:property <http://purl.org/dc/elements/1.1/:title> ;
    ui:sequence 0 ;
    ui:label "Name of the meal" ;
    ui:required true.
<#f9cc6fbd-e844-4588-b3bd-c45a63acac03> a ui:DateField ;
    ui:property schema:orderDate ;
    ui:sequence 1 ;
    ui:label "Date of your visit" ;
    ui:required true.
<#69c1cf77-4a0b-4951-a374-4612d113dade> a ui:Choice ;
    ui:property schema:ratingValue ;
    ui:sequence 2 ;
    ui:label "Rating" ;
    ui:required true ;
    ui:from <#69c1cf77-4a0b-4951-a374-4612d113dade-options>.
<#7267c9f3-5d1d-4621-8f83-027931bf1072> a ui:SingleLineTextField ;
    ui:property schema:ratingExplanation ;
    ui:sequence 3 ;
    ui:label "Argumentation".
<#69c1cf77-4a0b-4951-a374-4612d113dade-options> a owl:Class.
```

```
ex:NotLikedIt a <#69c1cf77-4a0b-4951-a374-4612d113dade-options> ;
    skos:prefLabel "I didn’t like it".
ex:LikedIt a <#69c1cf77-4a0b-4951-a374-4612d113dade-options> ;
    skos:prefLabel "It was tasty".
ex:LovedIt a <#69c1cf77-4a0b-4951-a374-4612d113dade-options> ;
    skos:prefLabel "It was excellent".
```