DFDP: A Declarative Form Description Pipeline for Decentralizing Web Forms

Ieben Smessaert[1], Patrick Hochstenbach[1,2], Ben De Meester[1], Ruben Taelman[1] and Ruben Verborgh[1]

[1]*IDLab, Department of Electronics and Information Systems, Ghent University - imec, Ghent, Belgium*
[2]*Ghent University Library, Ghent, Belgium*

Abstract
Forms are key to bidirectional communication on the Web: without them, end-users would be unable to place online orders or file support tickets. Organizations often need multiple, highly similar forms, which currently require multiple implementations. Moreover, the data is tightly coupled to the application, restricting the end-user from reusing it with other applications, or storing the data somewhere else. Organizations and end-users have a need for a technique to create forms that are more *controllable*, *reusable*, and *decentralized*. To address this problem, we introduce the *Declarative Form Description Pipeline* (DFDP) that meets these requirements. DFDP achieves controllability through end-users' editable declarative form descriptions. Reusability for organizations is ensured through descriptions of the form fields and associated actions. Finally, by leveraging a decentralized environment like Solid, the application is decoupled from the storage, preserving end-user control over their data. In this paper, we introduce and explain how such a declarative form description can be created and used without assumptions about the viewing environment or data storage. We show how separate applications can interoperate and be interchanged by using a description that contains details for form rendering and data submission decisions using a form, policy, and rule ontology. Furthermore, we prove how this approach solves the shortcomings of traditional Web forms. Our proposed pipeline enables organizations to save time by building similar forms without starting from scratch. Similarly, end-users can save time by letting machines prefill the form with existing data. Additionally, DFDP empowers end-users to be in control of the application they use to manage their data in a data store. User study results provide insights to further improve usability by providing automatic suggestions based on field labels entered.

Canonical version: https://smessie.github.io/Article-ESWC2024-DFDP/
DOI: 10.5281/zenodo.10285192, 10.5281/zenodo.10285210, 10.5281/zenodo.10285224
URL: https://w3id.org/DFDP

1. Introduction

Web forms are a part of our daily lives: taking a survey, filling out shipping information for an online order, or filling out an administrative request. End-users regularly have to re-enter

The Second International Workshop on Semantics in Dataspaces, co-located with the Extended Semantic Web Conference, May 26 – 27, 2024, Hersonissos, Greece
ieben.smessaert@ugent.be (I. Smessaert); patrick.hochstenbach@ugent.be (P. Hochstenbach); ben.demeester@ugent.be (B. D. Meester); ruben.taelman@ugent.be (R. Taelman); ruben.verborgh@ugent.be (R. Verborgh)
0009-0004-5281-0723 (I. Smessaert); 0000-0001-8390-6171 (P. Hochstenbach); 0000-0003-0248-0987 (B. D. Meester); 0000-0001-5118-256X (R. Taelman); 0000-0002-8596-222X (R. Verborgh)

the same data, without the ability to prefill a similar form with pre-existing data. In addition, the data will almost always be stored on the service provider's server, and the end-user cannot access it again or choose to store it somewhere else. If an end-user requires a form similar to an existing one, created by someone else or with another application, they will often have to build a new one from the ground up without the ability to copy and modify an existing form.

Current Web forms are meant to be used (1) with one endpoint, (2) for one (Web) display, (3) with one workflow in mind, and (4) without a means to send and receive the data in another way. First, when a form is submitted, the data is sent to the same location, and the user has no control over where the data is stored. Typically, the application developer decides where and how to store the data – resulting in poor data control for users [1] – and each application has a custom non-standardized API – hindering operability between applications [2]. Second, during form filling, it is not possible to use an alternative environment, with disparate display contexts, or to opt for a different website due to a preferred interface. Such an alternative environment may be subtle (e.g. a dedicated mobile version, or a version that integrates seamlessly into a company's ecosystem). For the sake of clarity, we will use an extreme example of a command line and a Web view to illustrate this problem in the remainder of the paper. Third, form applications designed for one specific use case cannot typically be repurposed for another, e.g. a shipping information form for customers during a purchase cannot usually be repurposed to submit a support request. Fourth, it is not possible to reuse data across multiple applications as data from one application cannot be written or read by another.

A requirement for solving these problems is decoupling the data from the application. Solid is a specification for decentralized data storage that is decoupled from the application. This allows the form description to be stored separately from the form renderer application, which in turn improves end-user control over their data. Several solutions have been proposed using Solid [3, 2] and Linked Data, but none of them fully semantically describe a declarative form from elements to actions. A couple of notable exceptions exist [4, 5], but these works are mainly theoretical approaches to the problem.

We introduce the *Declarative Form Description Pipeline* (DFDP), looking at forms as 3 parts: *display*, *shape*, and *reasoning*. In the DFDP, a *form generator* outputs the form description that serves as input to a *form renderer. Display* defines how the form elements should look, *shape* defines the expected data structure, and *reasoning* does footprint and schema alignment. We use a *footprint* to describe what to do with the filled-in data at certain events (e.g. submission). These 3 parts provide a fully declarative description of the form. To fully decouple description from application, the application needs to understand any form expression vocabulary. This necessitates schema alignment to map the description to a vocabulary the application understands, and removes the assumption that all applications will use the same language.

In the context of dataspaces, declarative form descriptions demonstrate their relevance by complementing the existing well-defined dataspaces data model with a declarative definition of the form as an interface that prompts the user for data input. For instance, within the health dataspace, each hospital can use its own form for medical data entry. By embedding semantics into the inserted data, alignment across all hospitals' data can be achieved.

The paper is structured as follows. We first discuss related work (Section 2) and present a motivating example and requirements (Section 3). We then introduce the architecture and discuss its implementation (Section 4 and 5). We evaluate in Section 6 and present conclusions and future work (Section 7).

2. Related Work

2.1. Standalone Technologies

XForms [6] was introduced in 2003 as a standalone technology for collecting inputs from Web forms. The specification became a W3C Recommendation in 2009. In 2010, work started on XForms 2.0 [7]. The XForms architecture separates *presentation*, *content*, and *purpose*, in a way that closely resembles the *display*, *shape*, and *reasoning* architecture introduced in this paper. Web forms are structured using XML, with elements and attributes specifying constraints. XForms lacks reasoning capabilities and Semantic Web functionalities for producing and understanding RDF data, limiting its potential in our research focus.

An alternative technology for form-based RDF data editing and presentation is RDForms [8], comprising two main components: the *RDForms library* parse, serialize, and manipulate RDF graphs, and *RDForms templates* ensure correct RDF expression production and manipulation. However, the system lacks reasoning capabilities. In this paper, we propose adding schema alignment and event-driven actions to achieve a fully decoupled declarative form solution.

2.2. Ontologies

The Shapes Constraint Language (SHACL) [9] is a W3C Recommendation to define and validate RDF graphs based on conditions declared in shapes. These shapes serve as descriptions of the data graphs they validate and can also define display fields for forms. Property shapes include non-validating properties (e.g. for supplementary form construction information), and validating properties (e.g. for specifying datatype and minimum count constraints).

Solid-UI introduces foundational components for user interface widgets and utilities tailored for Solid applications, and a UI ontology [10]. Solid-UI Forms primarily addresses form display aspects and associates semantic meaning with form fields via the `ui:property` predicate. Additional properties can be employed to specify field behavior.

Other efforts (e.g. RDF-Form [11]) also describe forms in RDF, enabling semantic representation of form fields, addressing display and shape aspects in the three-part view. Additionally, we aim to describe footprints that outline actions for events (e.g. submission), such as data storage location or required updates to other documents [5]. Describing this as footprints enables machine interpretation, ensuring actions aren't bound to specific applications. Consequently, data handling becomes standardized across applications, fostering real interoperability. However, existing ontologies lack a mechanism for describing footprints, leaving a gap in achieving a fully declarative form description solution.

3. Motivation And Requirements

In this section, we discuss the main requirements for decentralized and declarative Web forms using a use case involving two personas, "Alice" and "Bob", each with their own objectives about the forms they like to publish, but both requiring full control over where and how the resulting RDF is stored.

Alice, a Web-savvy user, owns a personal data store for storing RDF data. She is a researcher seeking to include a list of her publications within her personal data store as an integral part of her CV. Companies typically request CVs without providing a tailored form for their creation. Rather than relying on a pre-existing CV application with a fixed data structure, Alice prefers to use her own rendering. This allows her to employ an existing CV data model while also incorporating her own metadata. Consequently, companies will retrieve richer data.

To create a form to populate her data store, Alice uses *AliceForms*, which shares similarities with Google Forms. *AliceForms* enables form creation in a user-friendly manner with a simplified data model. Importantly, Alice retains control over where the form description and resulting RDF data are stored, ensuring control over the data she enters — an option not available in centralized platforms like Google Forms.

When Alice wants to create RDF data, she opens the form description in the renderer application *HtmlForms*, providing her with an HTML rendered version of the form description. When Alice fills out the form, the RDF data will be stored in her data store. *HtmlForms* can be automatically prefilled with RDF data to save Alice from repeatedly entering the same data.

Bob, a friend of Alice, wants to invite her for lunch and is searching for a suitable date. He uses his Web application *BobForms* to generate a form description, configuring it to save the resulting RDF data to his data store. Alice, who prefers command-line tools, receives a link to Bob's form description. Using her command-line application *TextForms*, Alice accesses the form description, which is then rendered in a text-based format. After completing the form, the data is sent as RDF data to Bob's data store.

Alice loves Bob's form and decides to adapt it to her own needs, effortlessly transitioning from an end-user to a form developer. She opens the form description in *AliceForms* and modifies the RDF data model to suit her requirements. She incorporates logic to ensure that any submission sends RDF data to her data store and notifies Bob's data store Inbox. Furthermore, she configures a redirect to a custom page she has designed. As a result, the form description now includes additional fields and logic tailored to Alice's specifications. A sequence diagram of this example is included in Figure 1.

In a perfect world, Alice and Bob use a single form definition ontology. However, many RDF data models describe the same type of data. To be truly interoperable, applications such as *HtmlForms* and *TextForms* should contain some schema alignment capabilities to provide a translation between different form definition expressions.

In this paper, we examine the following three research questions (RQs) to address the issues raised by the above scenario. We then further explain each RQ, comparing it to the current state of the art. Each RQ's requirements are discussed, and summarized in Table 1.

- RQ1: How can form developers create forms to declaratively control machines for producing RDF in **multiple viewing environments** (e.g. Web pages vs text-based command line)?
- RQ2: How can machines **translate form descriptions** decoupled from the application into a vocabulary that the application understands?

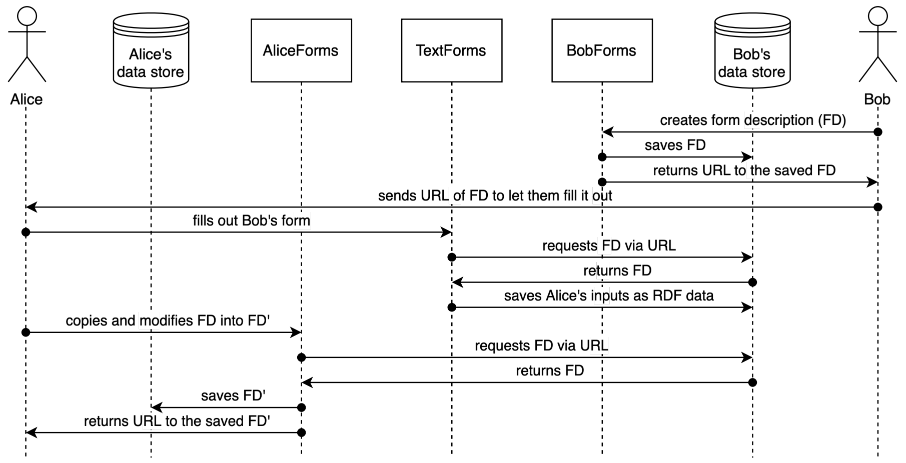

Alice
Alice's data store
AliceForms
TextForms
BobForms
Bob's data store
Bob
creates form description (FD)
saves FD
returns URL to the saved FD
sends URL of FD to let them fill it out
fills out Bob's form
requests FD via URL
returns FD
saves Alice's inputs as RDF data
copies and modifies FD into FD'
requests FD via URL
returns FD
saves FD'
returns URL to the saved FD'

Figure 1: The motivating example of Bob and Alice involves interactions between Alice, her data store, AliceForms, TextForms, BobForms, Bob, and his data store.

- RQ3: How can machines **perform controllable and reusable actions** on the filled out data?

3.1. Multiple Viewing Environments

To enable simple reusability, forms need declarative descriptions (R1). This streamlines making adjustments by parsing, adjusting, and updating the description accordingly. XForms 2.0 does this by declaratively specifying the form in XML. In Solid-UI, forms are expressed using Linked Data in the Turtle format.

To allow machines interpreting the form and automatically prefill data, the declarative form description must be machine-interpretable (R2). Solid-UI's ontology semantically describes the elements' meanings, attaining machine interpretability. XForms 2.0 allows containing semantics in the form description using vocabularies, making it possible for machines to interpret the form. Additionally, the form description should be human-interpretable (R3) to manually fill out the form, a requirement that all state-of-the-art applications fulfil.

To allow Alice to choose the application in which she fills out the form, regardless of the environment, the form description must be renderable in multiple viewing environments (R4). No current application offers this capability.

3.2. Schema Alignment Tasks

To enable different applications to render the same form description, schema alignment is needed (R5) to decouple the application's vocabulary and the form description's vocabulary. Currently, no state-of-the-art application supports this: XForms 2.0 only works with its own XForms vocabulary, formatted in XML and Solid-UI Forms only works with its UI ontology.

Table 1
Summarized requirements based on the research questions.

N°	Category	Requirement
R1	Multiple View Environments	Declarative description of the form
R2	Multiple View Environments	Machine-interpretable description
R3	Multiple View Environments	Human-interpretable description
R4	Multiple View Environments	Render one form description in multiple view environments
R5	Schema Alignment Tasks	Schema alignment execution
R6	Footprint Tasks	Declarative description of the policies
R7	Footprint Tasks	Footprint execution

3.3. Footprint Tasks

A fully declarative form necessitates not only declarative descriptions for its elements, but also for the actions triggered by specific events. Thus, it is crucial to have declarative policies (R6) that describe the actions to be taken, along with the execution of these footprint tasks (R7). XForms 2.0 only allows specifying the submission endpoint using the Submission XML element, but no other policies can be specified. Solid-UI Forms has only limited support to execute or specify any footprint tasks by the use of the extra layer solid-ui-components [12], currently under development.

3.4. Requirements Summary

The scenario shows the need to FAIRly [13] describe the form description independent of the application context.[1] The requirements of the previous scenario are grouped into three categories and summarized in Table 1.

4. Architecture

As outlined in the Introduction, many Web applications tightly integrate data with the application itself, limiting end-user control over data and hindering interoperability between applications. This issue persists even in Solid applications, where data is often assumed to reside in fixed locations within the pod and to adhere to specific vocabularies. Therefore, this paper proposes dividing data into three parts: a form for display, a shape for validation, and a footprint for reasoning. Reasoning is used to determine the policy that needs to be executed based on the event specified in the footprint along with its associated policy. This three-part approach moves away from the current scenario where data is stored as an integral, tightly coupled part of the application. While end-users can continue to create web forms using familiar methods, such as drag-and-drop interfaces, storing the data decentralized in three parts allows organizations and end-users to reuse forms and data, saving time. As considerable research has been conducted

[1]The review of the adherence to the FAIR principles [13] can be found at the Wiki of the GitHub repository at https://github.com/SolidLabResearch/FormGenerator/wiki/Adherence-to-the-FAIR-Principles.

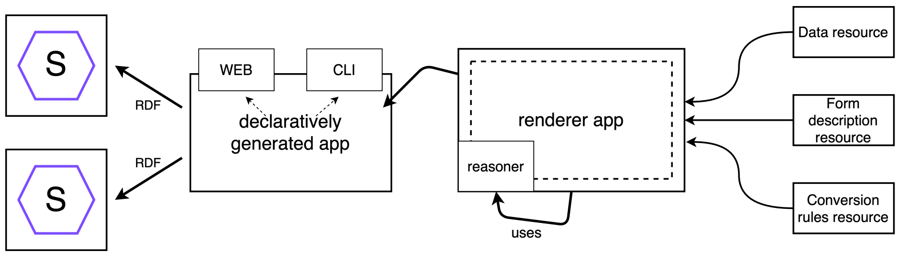

S
RDF
RDF
S
WEB
CLI
declaratively generated app
renderer app
reasoner
uses
Data resource
Form description resource
Conversion rules resource

Figure 2: A generic form renderer application can dynamically build an application for multiple viewing environments without assuming the interface and application design using the 3 inputs on the right and a reasoner to apply schema alignment and footprint tasks, to eventually store the enduser's filled in data in any data store, e.g. a Solid pod, as displayed on the left.

on RDF data validation [14, 15, 16, 17] and several implementations exist [18, 19, 20], this part is not covered in this paper.

In traditional centralized Web applications, different users interact with the same centralized Web server using different interfaces, all written for and working only with that server. Additionally, the data is stored on the application's server, outside the user's control. The Solid protocol [21] provides a standardized interface by providing a set of standard, open, and interoperable data formats and protocols [3]. As a result, different applications can use this set of protocols to work with the same data, overcoming the problem of each application using its own interface only working with that server. Because with Solid, the data is stored in a personal data store called a pod, it solves the issue of data storage outside the user's control. However, many applications are still being built with assumptions about the data stored in the pod, like the storage location and used vocabulary. Additionally, they are designed for one specific use case. In this work, we push this decentralized architecture a step further with the introduction of a declarative application that makes no assumptions about the interface and application itself. A schematic overview of the architecture is shown in Figure 2. First, a user who wants to create a form builds a declarative form description using a form generator. Then, the user sends a form description link to another user who can fill it out using a form renderer. A conversion rules resource maps the form description onto the renderer's ontology, while a data resource prefills the form automatically.

4.1. Description Of The Display

The previous problem of needing a separate application for each use case is solved by describing the user interface declaratively in the *form description resource*. This RDF resource contains both the display part and the footprint for the reasoning part. The display is the part responsible for rendering the form to the user. Web forms are typically rendered using HTML, while RDF represents the semantics of the form, not how you represent it in HTML. By declaratively describing the form in RDF, we achieve the ability to render the same form description in any environment. There already exist ontologies that can be used for this purpose, such as SHACL [9], UI ontology [10], and RDF-Form [11]. Reusing these ontologies for the display part ensures maximum compatibility with existing form descriptions. Semantic and declarative descriptions also enable machines to interpret the form's meaning, facilitating machine-driven prefilling of forms. This is achieved with the binding property on each form field, linking to the meaning of that field and serving as the predicate of the triple with the filled-in values for these fields as objects. The subject's type describing the form's meaning is equal to the form's binding. The *data resource* structure mirrors the filled-out form's output, enabling automatic prefilling of the form.

4.2. Description Of The Schema Alignment 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. To allow the application to interpret multiple ontologies and hence achieve a decoupled solution, *schema alignment tasks* are introduced translating the form description into an ontology the application understands. This enables the application to treat two definitions that semantically describe the same thing as identical. The *conversion rules resource* from Figure [**?**] is used by the renderer application to perform this mapping. By providing this resource to the application, it doesn't need to comprehend the form description's ontology. Any ontology with a mapping can be used. The renderer applies these rules using a reasoner to interpret the form description in its language. As these conversion rules can be passed as a separate resource to the application, the end-user does not necessarily have to create them himself. Ontology creators can provide mappings to similar ontologies, or application developers can provide mappings from equivalent ontologies to the one their application understands.

4.3. Description Of The Footprint Tasks

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 application of reasoning next to schema alignment. To describe policies, two languages are needed: a *rule language* and a *policy language*. The policy language describes what should actually happen when a policy is executed. The rule premise contains the event and the rule conclusion contains the policy. Policies should describe the client-side operations that need to be performed when a certain event occurs. This can be much more than just performing an HTTP request to the server, such as redirecting the end-user who filled out the form, performing an *N3 Patch* request, or sending a notification to someone's inbox.

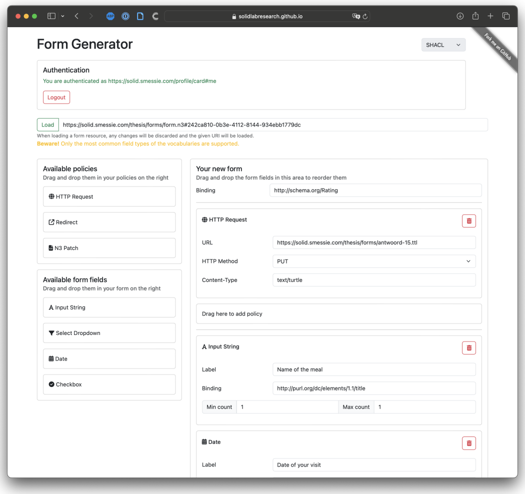
5. Implementation

We implemented three applications in TypeScript. The FormGenerator application generates a form description based on the form the user builds using drag-and-drop. The FormRenderer application and FormCli application are two applications that render a given form description in respectively a Web browser using HTML or a text-based command-line interface.

5.1. FormGenerator

The first application in the pipeline generates the declarative form description.[2] In Figure 3 a screenshot of the implemented application can be seen, showing how form developers can define policies and form fields using a drag-and-drop interface.

Describing footprints requires a rule and a policy language. As rule language, Notation3 (N3) [22] is used. The rule premise defines the event, while the rule conclusion defines the policy. We chose N3 as it proved to be a working solution for our use case and the reasoning engine EYE implementing N3 is being developed at our lab. We therefore also made the decision to use the EYE-JS library [23], a browser and node-distributed EYE reasoner via WebAssembly. By the use of reasoning, we obtain the rule conclusion which is then parsed using a SPARQL query. Querying is done using Comunica [24], a knowledge graph querying framework.


solidlabresearch.github.io

Form Generator

SHACL

Fork me on GitHub

Authentication
You are authenticated as https://solid.smessie.com/profile/card#me
Logout

Load https://solid.smessie.com/thesis/forms/form.n3#242ca810-0b3e-4112-8144-934ebb1779dc
When loading a form resource, any changes will be discarded and the given URI will be loaded.
Beware! Only the most common field types of the vocabularies are supported.

Available policies
Drag and drop them in your policies on the right
HTTP Request
Redirect
N3 Patch

Available form fields
Drag and drop them in your form on the right
Input String
Select Dropdown
Date
Checkbox

Your new form
Drag and drop the form fields in this area to reorder them
Binding http://schema.org/Rating

HTTP Request
URL https://solid.smessie.com/thesis/forms/antwoord-15.ttl
HTTP Method PUT
Content-Type text/turtle

Drag here to add policy

Input String
Label Name of the meal
Binding http://purl.org/dc/elements/1.1/title
Min count 1 Max count 1

Date
Label Date of your visit

Figure 3: Implemented FormGenerator application.

The policy we obtain is defined using a policy language. There are already existing ontologies that can be reused to describe policies, even though they were not designed for this purpose. Hydra [25] is a vocabulary to describe Web APIs in Linked Data and its intended use is to describe the server side of the API in a machine-readable way. A major limitation of our research is that it can only describe HTTP requests, while policies go beyond that. Therefore, we chose to not use Hydra.

The *Function Ontology (FnO)* [26] is used to semantically define and describe implementation-independent functions, including their relations to related concepts such as parameters, and mappings to specific implementations and executions. As FnO allows the description of any kind of operation unlike e.g. Hydra which only allows the description of HTTP requests, a basic version of this existing ontology together with the HTTP Vocabulary [27] is reused to describe the policy. A *Policy ontology* has been developed using the LOT Methodology [28] defining the missing classes and properties required for policy definition.[3] The ontology enables the description of events and the corresponding actions to be taken. The ontology was implemented using Protégé as the ontology development environment, using the OWL language. The ontology's source is available on GitHub, and the ontology itself is published using GitHub pages. A w3id is used to provide a permanent identifier for the ontology, which includes both human-readable documentation and a machine-readable file accessible via the URI using content negotiation. Hosting the source on GitHub facilitates easy maintenance, and contributions can be made through pull requests and the issue tracker. Listing 1 contains an

[2]The FormGenerator source code can be found at https://w3id.org/DFDP/FormGenerator/source and the live version at https://w3id.org/DFDP/FormGenerator/app.

[3]The Policy ontology can be found at https://w3id.org/DFDP/policy.

example of a footprint task sending an HTTP request.

When constructing the form, users must specify bindings for each field, which are URIs semantically describing the fields. Users must manually enter these bindings. To simplify this process, they can utilize prefixes, which are automatically expanded to full URIs via the prefix.cc API. As an example, `ex:MyField` will become `http://example.org/MyField`.

Listing 1: Example of N3 rule describing HTTP request policy to be executed on the form submission event.

```
@prefix ex:    <http://example.org/> .
@prefix pol: <https://w3id.org/DFDP/policy#> .
@prefix fno: <https://w3id.org/function/ontology#>.
@prefix http: <http://www.w3.org/2011/http#>.
{
  ?id pol:event pol:Submit.
} => {
  ex:HttpPolicy pol:policy [
    a fno:Execution;
    fno:executes http:Request;
    http:methodName "POST";
    http:requestURI <https://httpbin.org/post>;
    http:headers (
      [
        http:fieldName "Content-Type";
        http:fieldValue "application/ld+json"
      ]
    )
  ] .
} .
```

5.2. FormRenderer And FormCli

The next application in the pipeline renders the declarative form description and lets the user fill out that form. We implemented two versions in two different viewing environments to prove that the display part of the form description is independent of the viewing environment.[4][5] The FormRenderer application, as shown in the screenshot in Figure 4, functions in the Web browser. The FormCli application operates as a command-line application, allowing usage without a GUI. The form questions are prompted to the user one after the other. While the FormRenderer application supports authenticating with a Solid identity provider, authentication is not implemented in the FormCli application as the Solid protocol lacks proper authentication for command-line applications. We therefore consider this outside the scope of this research.

The UI ontology is chosen as the application's display ontology as it is designed specifically for defining user interfaces. Schema alignment is achieved by applying conversion rules to the form description.[6] The implementation uses N3 rules together with the EYE-JS reasoner to apply them. The resulting form description in the UI ontology is parsed by the Comunica engine via SPARQL queries.

5.2.1. Determining The Subject For The Produced RDF

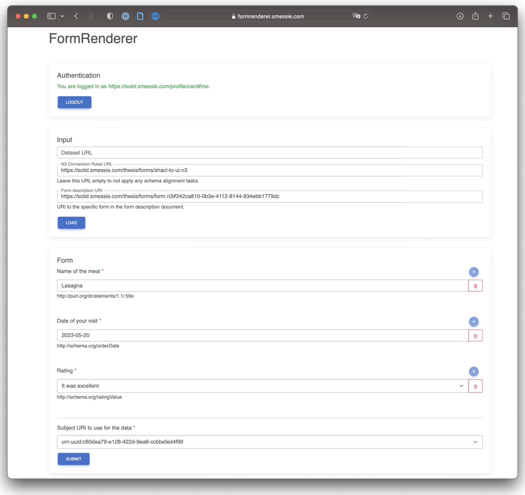
When dealing with a resource containing pre-existing data for form filling, it's straightforward to determine the subject URI for writing new data — it can be reused from the existing data. Furthermore, when no resource is provided or when multiple subjects within the resource conform to the form's structure and target class, determining the subject URI becomes ambiguous. Various solutions were explored to address this problem.

[4]The FormRenderer source code can be found at https://w3id.org/DFDP/FormRenderer/source and the live version at https://w3id.org/DFDP/FormRenderer/app.

[5]The FormCli source code can be found at https://w3id.org/DFDP/FormCli/source.

[6]An example of SHACL to UI ontology conversion rules can be found at ??.

1. Generating a new random UUID and using it as the subject URI with the `urn:uuid:` namespace [29].
2. Prompting the user to enter a subject URI.
3. Utilizing the URI from the HTTP Request policy as the subject URI.
4. Selecting one of the existing subjects as the subject URI.
5. Employing a blank node in place of a subject URI.
6. Specifying the subject URI in the form description.



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 4: Implemented FormRenderer application.

Blank nodes are often an unsuitable solution since they lack a URI, making it impossible to reference the data using a valid URI or to link to/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. Not all ontologies for describing forms have a property to define the subject URI, eliminating option 6. This leaves us with options 1, 2, and 4 as the viable choices. Using a random UUID is a feasible solution, ensuring uniqueness, and serving as an ideal default subject URI. Prompting the user enables them to enter a relevant subject URI themselves, which is also feasible. Focusing solely on this option requires users to understand subject URIs, potentially complicating application use for those new to the Semantic Web. Our goal is to ensure ease of use for all. Using an existing subject is a good option, especially when there's a single subject, aligning with user expectations for data editing. With multiple subjects, selection becomes a challenge for users unfamiliar with the concept. We propose and implement a combination of the three feasible options. Use a random UUID as the default subject URI, while enabling user selection from existing subjects or manual subject URI entry. Parsing the policies is done in the same way as in the FormGenerator application, described earlier in Subsection 5.1.

6. Evaluation

In this section, we evaluate the proposed architecture and implementation. First, we analyze conformance to the requirements in Subsection 6.1. Second, we describe the results of the user study in Subsection 6.2. We created a sustainability plan to guarantee the maintenance of the DFDP's tools, both short-term and long-term.[7]

[7]The sustainability plan can be found at https://github.com/SolidLabResearch/FormGenerator/wiki/Sustainability-Plan.

6.1. Conformance To The Requirements

In this section, we analyze the conformance of the implemented applications to the requirements defined in Table 1 under Section 3.

The first research question is: ”RQ1: How can form developers create forms to declaratively control machines for producing RDF in **multiple viewing environments** (e.g. Web pages vs text-based command line)?”. The FormRenderer and FormCli applications showcase creating a form renderer application in multiple viewing environments (R4). The form was described declaratively (R1) as the display part is fully described using the already existing UI ontology. Data can be passed along to automatically prefill a form by machines using the form fields' bindings (R2), and users can interpret the field title to manually enter a value (R3).

The second research question is: ”How can machines **translate form descriptions** decoupled from the application into a vocabulary that the application understands?”. N3 conversion rules can be passed along allowing reasoning to automatically translate the form description into the ontology the application understands (R5).

The third research question is: ”How can machines **perform controllable and reusable actions** on the filled out data?”. The form description also declaratively describes what should happen in case of a certain trigger (R6). Because this is described in a machine-readable way using RDF, a machine can interpret this and execute the right actions (R7).

Considering the information above, we can confidently assert that we cover all the functional requirements outlined in Table 1.

6.2. User Study

To show that the user experience is not impacted due to the Semantic Web technologies used in the DFDP, we perform a user study. The goal of this user study is to see if next to the functional requirements, the implementations are also comprehensible for end-users. We define ”comprehensible for users” as the ability for users to understand the application in such a way that they can use it correctly (i.e. accurately) without getting frustrated or giving up (i.e. in a reasonable time). In what follows, we discuss the user experience by doing a qualitative analysis [30] by the use of the *think aloud method* [31] and open-ended interviews split up into two parts: form editing (evaluating the FormGenerator) and form usage (evaluating the FormRenderer). The FormCli application is not considered as this is a more complex version of the FormRenderer application.

6.2.1. Method

Potential participants were directly contacted by the author out of which 19 took part in the study. 8 individuals with a technical background evaluated the FormGenerator application, aligning with its target audience, much like creating a Google Forms is intended for users with technical proficiency. To meet this requirement, computer science students were primarily sought as participants. The 11 participants for the form usage part were people both with and without technical backgrounds with ages spanning from 18 to 52 years. This aligns with the target group of this and similar traditional applications [32], assuming users are familiar with a computer and the Web. Both groups were provided with simple scenarios instructing them to either generate a form for a restaurant review or fill out an earlier generated form, modeled to mimic normal day tasks.[8] The FormGenerator participants were given a list of bindings to be used to create the form, fulfilling the assumption that they have knowledge of Linked Data. All necessary data elements were put in place for the FormRenderer application by giving the users a specific link autopopulating the input fields.

6.2.2. Threats To Validity

As with any study, our evaluation carries threats to validity. We identified the following external and internal threats [30] because all participants were recruited from the same country and most of them are students at the same university. We identified the external threat *Interaction of selection and treatment*. However, we do assume that users of the form editing have more technical knowledge, so this lies in line with our intentions. Additionally, basic computer and web proficiency are assumed for form usage.

The internal threat *Selection* was also identified. While most participants possess technical knowledge, this is deliberate as it is a prerequisite for the form editing part. To ensure equal knowledge of Linked Data, we provided possible bindings. To mitigate a selection bias, we also recruited some non-students and participants from different ages.

6.2.3. Results

We categorized feedback from all users for both applications and received generally positive results. Especially in the case of the FormRenderer, no participant noted any significant differences in terms of ease of use. Regarding the FormGenerator, 5 out of 8 participants rightly noted that as restaurant owners, they shouldn't need to be concerned with bindings. As a result, future research should consider the automatic suggestion of bindings based on the field label entered by the user.

7. Conclusion

In this article, we introduced a Declarative Form Description Pipeline (DFDP) offering significant enhancements in data management for both end-users and organizations. End-users benefit from increased control over their data input, facilitated by footprints that specify data handling actions. Additionally, they can reuse existing data to prefill forms and choose their preferred application to manage their data, regardless of the underlying ontology, thanks to schema alignment. This decoupling of applications and data ensures greater flexibility and autonomy for end-users. For organizations, DFDP enables better data management in terms of reusability, allowing for easy adoption and customization of existing forms without requiring additional assumptions from developers. This, in turn, improves cost-efficiency and promotes streamlined workflows. The DFDP enables the embedding of semantics into the input data, improving the alignment of data across different related forms in a dataspace. Overall, DFDP marks a

[8]The user study scenarios can be found at https://github.com/SolidLabResearch/FormGenerator/wiki/User-Experience-Scenarios.

significant step forward in enhancing data management practices with its declarative and decoupled Web forms for both end-users and organizations.

Future research could focus on automatically suggesting bindings based on field names entered by the users, which can build on *Entity Extraction* [33]. Other directions for future work may involve streamlining the schema alignment and footprint tasks to make them more abstract for developers. Embedding schema alignment as a layer between the application and the data would enable developers to automatically consider semantically equivalent ontologies as the same, resulting in real interoperability without requiring extra work from the developers. A first step could be to package the DFDP modules as libraries to ease the integration into existing Web applications.

Acknowledgments

This work is supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project (VV023/10)). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

[1] T. Berners-Lee, Three challenges for the web, according to its inventor - World Wide Web Foundation, 2017. URL: https://webfoundation.org/2017/03/web-turns-28-letter/.
[2] A. V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, T. Berners-Lee, Solid: a platform for decentralized social applications based on linked data, MIT CSAIL & Qatar Computing Research Institute, Tech. Rep. (2016).
[3] T. Berners-Lee, et al., Solid, 2022. URL: https://solidproject.org.
[4] T. Berners-Lee, Linked Data Shapes, Forms and Footprints - Design Issues, 2019. URL: https://www.w3.org/DesignIssues/Footprints.html.
[5] R. Verborgh, Shaping linked data apps, 2022. URL: https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/.
[6] J. M. Boyer, XForms 1.1, 2009. URL: https://www.w3.org/TR/xforms/.
[7] E. Bruchez, A. Couthures, P. Steven, Xforms 2.0 - xforms users community group, 2022. URL: https://www.w3.org/community/xformsusers/wiki/XForms_2.0.
[8] MetaSolutions, RDForms - RDF in HTML-forms, 2009. URL: https://rdforms.org.
[9] H. Knublauch, D. Kontokostas, Shapes constraint language (shacl), 2022. URL: https://www.w3.org/TR/shacl/.
[10] SolidOS, An ontology for user interface description, hints and forms, 2022. URL: https://www.w3.org/ns/ui#.
[11] D. Beeke, Rdf form, 2022. URL: https://rdf-form.danielbeeke.nl.
[12] J. Zucker, Solid UI Components, 2023. URL: https://github.com/jeff-zucker/solid-ui-components, original-date: 2021-04-01T16:33:22Z.
[13] M. D. Wilkinson, et al., The fair guiding principles for scientific data management and stewardship, Scientific data 3 (2016) 1–9. URL: https://www.nature.com/articles/sdata201618.
[14] D. Tomaszuk, Rdf validation: A brief survey, in: BDAS, Springer, 2017, pp. 344–355.

[15] E. Prud'hommeaux, J. E. Labra Gayo, H. Solbrig, Shape expressions: an rdf validation and transformation language, in: SEMANTiCS, 2014, pp. 32–40.

[16] D. Arndt, B. D. Meester, A. Dimou, R. Verborgh, E. Mannens, Using rule-based reasoning for rdf validation, in: RuleML+RR, Springer, 2017, pp. 22–36.

[17] I. Boneva, J. E. Labra Gayo, E. G. Prud'Hommeaux, Semantics and validation of shapes schemas for rdf, in: ISWC, Springer, 2017, pp. 104–120.

[18] W. Slabbinck, CommunitySolidServer/shape-validator-component, ???? URL: https://github.com/CommunitySolidServer/shape-validator-component.

[19] Zazuko, rdf-validate-shacl, 2020. URL: https://github.com/zazuko/rdf-validate-shacl, original-date: 2020-02-25T16:45:45Z.

[20] T. Bergwinkl, shacl-engine, 2023. URL: https://github.com/rdf-ext/shacl-engine, original-date: 2023-01-27T00:22:30Z.

[21] S. Capadisli, T. Berners-Lee, R. Verborgh, K. Kjernsmo, Solid protocol, 2022. URL: https://solidproject.org/TR/protocol.

[22] D. Arndt, W. Van Woensel, D. Tomaszuk, G. Kellogg, Notation3, 2022. URL: https://w3c.github.io/N3/spec/.

[23] W. Jesse, J. De Roo, Eye js, 2022. URL: https://github.com/eyereasoner/eye-js.

[24] R. Taelman, J. Van Herwegen, M. Vander Sande, R. Verborgh, Comunica: a modular sparql query engine for the web, in: ISWC, 2018. URL: https://comunica.github.io/Article-ISWC2018-Resource/.

[25] M. Lanthaler, C. Gütl, Hydra: A vocabulary for hypermedia-driven web apis., LDOW 996 (2013) 35–38.

[26] B. De Meester, T. Seymoens, A. Dimou, R. Verborgh, Implementation-independent function reuse, Future Generation Computer Systems 110 (2020) 946–959.

[27] J. Koch, C. A Velasco, P. Ackermann, HTTP Vocabulary in RDF 1.0, 2017. URL: https://www.w3.org/TR/HTTP-in-RDF10/.

[28] M. Poveda-Villalón, A. Fernández-Izquierdo, M. Fernández-López, R. García-Castro, LOT: An industrial oriented ontology engineering framework, Engineering Applications of Artificial Intelligence 111 (2022) 104755. URL: https://www.sciencedirect.com/science/article/pii/S0952197622000525. doi:10.1016/j.engappai.2022.104755.

[29] P. Leach, M. Mealling, A Universally Unique IDentifier (UUID) URN Namespace, 2015. URL: https://www.ietf.org/rfc/rfc4122.txt.

[30] J. W. Creswell, J. D. Creswell, Research design: Qualitative, quantitative, and mixed methods approaches, Sage Publications, Inc., 2018.

[31] M. Van Someren, Y. F. Barnard, J. Sandberg, The think aloud method: a practical approach to modelling cognitive, London: AcademicPress 11 (1994) 29–41.

[32] A. Petrosyan, Distribution of internet users worldwide as of 2021, by age group, 2023. URL: https://www.statista.com/statistics/272365/age-distribution-of-internet-users-worldwide/.

[33] P. Exner, P. Nugues, Entity extraction: From unstructured text to dbpedia rdf triples., in: WoLE@ ISWC, 2012, pp. 58–69.