Uduvudu: a Graph-Aware and Adaptive UI Engine for Linked Data

Michael Luggen
eXascale Infolab
University of Fribourg
Fribourg—Switzerland

Adrian Gschwend
Division of Computer Science
Bern U. of Applied Sciences
Biel—Switzerland

Bernhard Anrig
Division of Computer Science
Bern U. of Applied Sciences
Biel—Switzerland

Philippe Cudré-Mauroux
eXascale Infolab
University of Fribourg
Fribourg—Switzerland

ABSTRACT

Creating good User Interfaces (UIs) to render Linked Data visually is a complex task, often involving both UI and Linked Data specialists. The resulting solutions are typically application-dependent and difficult to adapt or reuse in a different context. To tackle this problem, we propose Uduvudu, a flexible, open-source engine to visualize Linked Data. Our engine is built in JavaScript and runs in the browser natively. Non-specialist users can use Uduvudu to describe recurring subgraph patterns occurring in their data. They can then flexibly and automatically extract, transform, and visually render such patterns in multiple ways depending of the usage context. Uduvudu is intuitive, flexible, and efficient and makes it possible to jump-start the development of complex user interfaces based on Linked Data without the need of data specialists.

Keywords

Templating, Visualization, User Interface, Development Process, Linked Data, RDF

1. INTRODUCTION

Companies are progressively adopting Linked Data in their internal IT ecosystems. As a result, they are increasingly interested to leverage Linked Data to build better User Interfaces (UIs), both for their employees and for their customers. Ideally, there should be as little information loss as possible when exporting the Linked Data to the UI. Often, however, Linked Data is transformed into less expressive data formats that are then rendered in a simplistic fashion in the interface. The exported Linked Data typically looses some of its structure, its semantics and/or some of its links. Ultimately, it becomes impossible to use the representation in the interface to write data back into the graph. Also, handling arbitrary Linked Data in order to render it properly in a UI is a singularly tedious task. We identify those two issues as important barriers precluding the broader adoption of Linked Data formats in the industry. Note that we make a difference between Linked Data visualization—where the goal is to build interfaces to easily navigate or summarize large quantities of data, and Linked Data rendering—where the goal is to select and individually render key values from the data. We specifically target the latter problem in this work, and propose a framework to easily create Linked Data UIs.

Our framework, called Uduvudu, considerably speeds up the development of Linked Data UIs by streamlining the data extraction, transformation, and rendering process through an explicit workflow. It allows several types of experts including data specialists, user interface experts, and graphical designers, to efficiently collaborate to produce the UI. Our framework is furthermore flexible, in the sense that it can render any valid RDF data irrespective of its schema or missing values, and produces reusable UIs and patterns that can be applied to different contexts and data.

We introduce a UI Creation process which enables teams to share the work of building good user interfaces. To support this process, we contribute an architecture dedicated to Linked Data. By creating an intermediate tree representation (Figure 5), extracted from the input graph (Figure 4) to build the resulting visualization (Figure 6), our architecture ensures the reuse of templates.

The rest of this paper is structured as follows. We start by reviewing related work in the fields of Linked Data visualization and rendering in Section 2. Then, we give an overview of the UI creation process we designed for our framework in Section 3. We introduce our framework architecture and its components in Section 4. Finally, we discuss a series of uses-cases for which Uduvudu was successfully used in Section 6, before concluding in Section 7.

2. RELATED WORK

There is a multitude of visualization approaches discussed in the scientific literature. Below, we provide an overview on the available frameworks focusing on the presentation layer of Linked Data. We also discuss the well-known Fresnel framework and how it compares to Uduvudu.

Copyright is held by the author/owner(s). WWW2015 Workshop: Linked Data on the Web (LDOW2015).

source: https://doi.org/10.24451/arbor.7517 | downloaded: 26.3.2020

	Uduvudu	Callimachus	Balloon Syn.	Fresnel	Exhibit	LESS
Level of Template	Subgraph	Application	JS Selector	Subgraph	Subgraph	Projection
Description of T.	underscore.js	RDFa Templates	Handlebars	Fresnel Formats	Exhibit Lenses	Smarty
Recursive Use of T.	Y	N	N	Y	N	N
Context Awareness	Y	N	N	N	N	N
Separation of Concerns	Y	N	Y	Y	N	Y
Editor	Y	Y	N	N	N	Y

Table 1: Comparison of frameworks on their template capabilities.

Most of the related work focused so far on the exploration and visualization of Linked Data. The focus of many systems is hence either on how to effectively explore large quantities of Linked Data, or on how to compactly aggregate large data to visualize it. Less attention has been given to the case where a data publisher already knows exactly which parts of the data he/she wants to publish.

Table 1 summarizes how Uduvudu's template capabilities compare to existing systems. As the table indicates, Uduvudu is, to the best of our knowledge, the only framework allowing the flexible description and recursive use of context-aware templates to visualize Linked Data (more on those topics in Section 4). In addition, Uduvudu imposes a strict separation of concerns (see Section 3) w.r.t. the data selection, the matching of available templates and the rendering based on the context. Finally, it features a full-fledged and intuitive editor, presented in more detail in Section 5.

2.1 Overview

Brunetti *et al.* [4] recently introduced the Formal Linked Data Visualization Model. The framework identifies a visual processing pipeline composed of three main components (Analyzer, Visualization Transform and Visualizer). The last two steps of the architecture match our own approach (their first step being out of scope for this work). Their approach analyze Linked Data structures with the goal of choosing the possible visualizations. Our implementation has been partly influenced by this approach, with the important difference that the goal of Uduvudu is not necessary to aggregate data for visualization, but rather to create a flexible framework to render specific portions of Linked Data. Their paper also gives an extensive overview of the different visualization libraries used in the Linked Data context.

2.2 Fresnel

Fresnel [8] is a well-known approach in the field of RDF visualization and presentation. Fresnel was originally designed to create common styles for the description of classes in RDF browsers. Because of this, it inherently assumes that an ontology is present in the system in order to visualize the data. Fresnel allows to specify in a detailed manner how to render data value properties. Creating more elaborate visual representations is however often complex, since the framework never was intended for such jobs.

We experienced further problems when working with Fresnel. First, ontologies typically do not provide a way to indicate if an attribute might be optional. While this is expected for RDF data processing, this poses problems at the presentation layer where it is crucial to know precisely which information is available. The rendering of a complete address (with street, city, and country values) can for instance be quite different from an address containing only the country or city name. Other researchers have also struggled with this limitation of Fresnel [5].

Through a mechanism called *lenses*, users can define which properties have to be selected in an RDF graph. The user has to specify which lenses shall be used for which graph as well as to indicate all the corresponding properties. From our perspective, the selection of which information is to be rendered should be part of the application logic instead, e.g., by providing a declarative representation (e.g., a SPARQL query) of the targeted data upfront.

The formatting step in Fresnel uses Formats to do simple string manipulations and Cascading Style Sheets to describe the rendering. While this approach makes it possible to create fairly complex renderings, it is also in our opinion intimidating for the designers, who most often do not have a deep understanding of the OWL syntax. Also, it is not possible to inject or attach transformation code to create complex renderings and visualizations. Finally, the output generation step of Fresnel is using the prepared information from the formatting step to render the visualization. Hence, it it not possible to change the representation based on the context, which is a very important requirement in practice and has been one of the original ideas behind the Semantic Web [3]. We think that the context of the presentation layer should be incorporated into the framework itself, and that is should support the developers in their task of creating adaptive user interfaces.

2.3 Template-Based Presentation

In [7], Khalili *et al.* present an extensive overview of how user interfaces relate to ontologies. In this work, we do not focus on the creation, completion or interaction with ontologies. Instead, our focus lies on the rendering of Linked Data. Furthermore, our framework does not even require a valid ontology or schema to begin with.

The use of templates for presentation of semantic data was introduced through Lens Templates in Exhibit [6]. They allow to create complex HTML structures that are familiar to web designers. Because Exhibit uses a sub model of RDF which is defined as JSON structures, there was no need to tackle potentially cyclic graph structures. The templates in Exhibit are coupled to the application (defined in html), which requires some interaction with the developer in order to be reused.

Templates used to render RDF data directly were introduced through LESS [1], which also partially removes the need of

ontologies to create RDF visualizations. Based on a use-case defined through a SPARQL query, a template is mapped to render the output. The authors use an extended version of a Web template language (Smarty) to then render the results. In addition, they propose a repository of templates which can be reused.

Both frameworks allow to recursively include other templates. Uduvudu enforces the possibility of creating fine-grained templates at all levels, without requiring any ontology or schema beforehand. We think that the potential reuse of these smaller templates is higher than coarser-grained mechanisms.

Callimachus [2] provides a full-stack solution allowing to build complete applications. As a part of the stack, a template-based mechanism to create RDFa templates is introduced. These templates are highly-coupled to the applications at hand. This differs to Uduvudu where templates are matched to the data at hand while Callimachus fetches the necessary data to fill a template (through creating a SPARQL request).

Balloon Synopsis [9] is an RDF Browser framework where templates are coupled with JavaScript functions to discover matching data structures. In Uduvudu, the matching part is decoupled from the templates. Furthermore, no knowledge of any programming language is required when matching structures. Finally, Uduvudu is designed as an engine to render general UIs, where a RDF browser might be one of its use-cases but not the sole focus.

Our framework extends the aforementioned approaches in several ways, most prominently by the automatic selection of the templates based on the input data, by the creation of a more flexible rendering process, and by templates that can match incomplete data.

In addition, the ability of Uduvudu to flexibly combine templates to create bigger structures makes it possible to adapt the presentation according to the availability of data facts.

3. LINKED DATA UI CREATION PROCESS

In order to tackle the challenges described above, we adapt a process workflow to streamline the creation of UIs for Linked Data. Our process is driven by three different actors:

The data expert is the person (or group) having knowledge on the Linked Data that needs to be rendered.

The UX specialist is the user experience person (or group) knowing how to best transform and restructure the data for the application at hand.

The graphical designer, finally, is the person (or group) specialized in creating effective visual renderings of data structures.

Those three roles might be assumed by the same person in small settings, but will typically be assumed by distinct persons in larger companies. Our UI creation process is given in Figure 1. The overall process can be summarized as follows:

- First, the data expert gathers and exports the data that needs to be rendered.
- The selected data is then passed to the UX specialist, who restructures it for rendering purposes.
- Finally, the graphical designer takes the various data structures built by the UX designer and builds the visual rendering of the data.

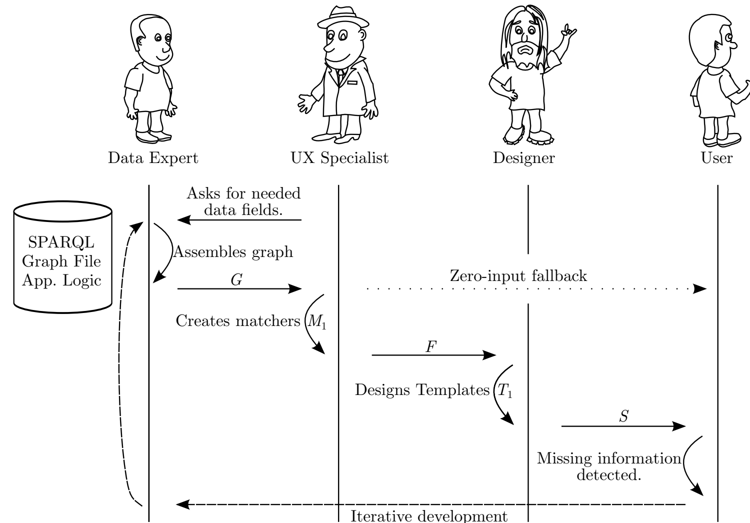

Figure 1: The UI creation process is split in multiple roles.

The Linked Data UI creation process we propose has a number of distinct advantages, including:

Clear separation of roles: involving the data expert—or the other specialists—throughout the entire process can be very costly in practice [4]. Instead, our process introduces a clear separation between the tasks undertaken by the different specialists, resulting in a better repartition of work and in increased autonomy for the experts.

Iterative development process: new elements can be added to each task without blocking the other tasks. That way, the experts can iterate on their own design autonomously without consulting the other experts.

Highly reusable outcome: as the the three tasks are partially decoupled, all the individual pieces of data, structures and templates created during the process can be reused and adapted later to another context, data, or application flexibly.

Zero-input fallback: any valid Linked Data provided by the data specialist can be rendered without any additional processing from the UX specialist or from the graphical designer (in that case, the data is simply rendered as lists of strings displayed at the bottom of the rendering).

4. ARCHITECTURE

The architecture we adopted for our framework was directly based on the creation pipeline described above, as well as on a series of prerequisites we elicited from data experts, UX specialists and graphical designers. It is composed of three main components: the *selector* choosing the input, the *matcher* which discovers known structures in the selected input, and the *renderer* which incorporates the visualization context to render. Figure 2 gives an overview of our architecture. We start below by introducing the prerequisites of our architecture, before giving some detail on each of its three main components.

4.1 Pre-requisites

Through our own experience and interviews with Linked Data specialists, UX experts and graphical designers, we came up with a set of pre-requisites for our architectures:

Never show URIs to non-technical end-users: They are systematically confused when confronted to *data URIs* displayed in a graphical interface. While data URIs are essential to uniquely identify entities and properties on the Web of data, they should not be used in mundane graphical interfaces where plain literals and labels should be preferred.

Leverage the extracted data structure for the rendering: the way data is extracted from the original data typically follows long deliberations and some decision process. We think that the resulting structures should be leveraged as much as possible for the creation of the final user interface.

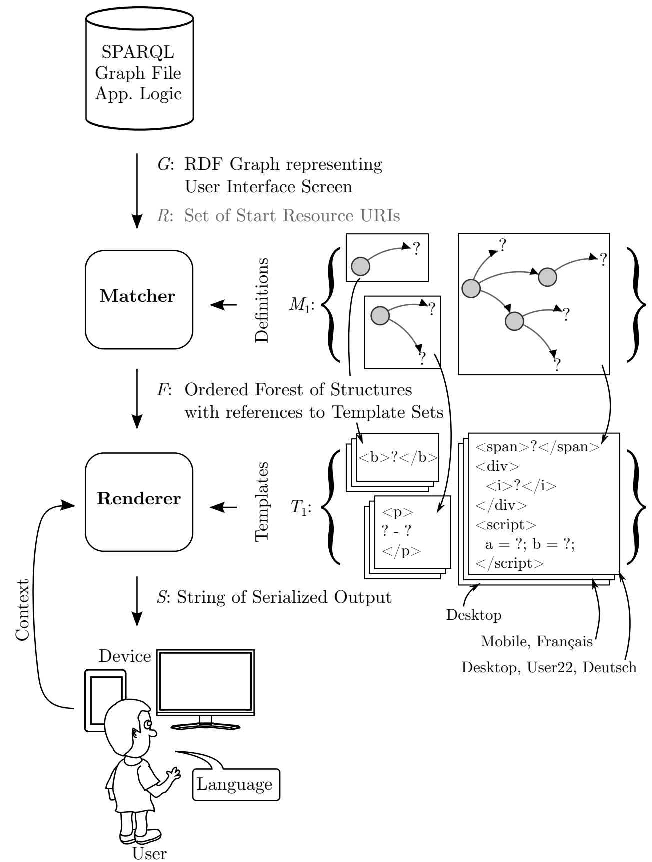

Figure 2: Overview of the Uduvudu architecture with the main components *Matcher* and *Renderer*.

Handle language transparently, until needed specifically: specific languages add a level of complexity to every application. Hence, we feel that for our process, if literals are available in multiple languages, only the one matching the context language should be presented, while still keeping the possibility to fine-tune and control the language if necessary depending on the application at hand (e.g., showing multiple languages).

Maximize the dynamic composition of information: our goal is not to come up with highly-complex renderings composed of thousands of distinct elements. Rather, we target usual business interfaces composed of a few dozens elements at a time maximum. Hence, our goal is not to come up with a highly-optimized and scalable rendering engine but rather to propose a dynamic composition model where Linked Data and templates can by dynamically recycled and integrated in order to create the final rendering.

4.2 The Data Selector

We now turn to the description of the three main components of our architecture. First, we start with the Data Selector, which decides which data that will be transformed

and rendered. Based on the use-case at hand, one needs to select and combine the relevant Linked Data from one or several sources. The decision on which part to select from the available data sources is typically an internal process, which is outside the scope of the present paper.

Any selector which supports classical Linked Data input, e.g., through a SPARQL query, an RDF/XML dump or triples serialized in one or several text files can be used. Uduvudu does not expect any inherent structure from the data (though it must be well-formed), which does not need to comply to any specific ontology or structure at this stage. Hence, the selector takes as input a superset of all informations that need to be shown to the end-user, and trim them to an input graph G containing exactly the information that needs to be rendered. This step is typically carried out by a data specialist. Figure 3 gives a simple example input graph corresponding to Figure 4.

```
@base <http://example.com/me/> .
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .

<corky> a vcard:Individual;
  vcard:fn "Corky Crystal";
  vcard:hasAddress <corky#address>;
  vcard:hasTelephone <corky#telephone>;
  vcard:nickname "Corks".

<corky#telephone> a vcard:Home, a vcard:Voice;
  vcard:hasValue "tel:+61755555555".

<corky#address> a vcard:Home;
  vcard:country-name "Australia";
  vcard:locality "WonderCity";
  vcard:postal-code "5555";
  vcard:street-address "111 Lake Drive".
```

Figure 3: G: an input RDF graph serialized using the Turtle format

4.3 Structure Matcher

The second component in our architecture, the Structure Matcher, holds a catalogue M of known structures (matchers) and tries to match parts of this catalogue onto the input graph G.

The catalogue contains on one hand custom matchers that describe dedicated processings for specific data patterns, and predefined matchers on the other hand that act as a fall-back solution in case no custom matcher can be matched to some node in the input graph. Figure 7 gives an example of a structure matcher.

The custom structures in the catalogue are typically provided by the UX specialist. They are serialized as JSON structures in our implementation (see below), and can be generated through a GUI or can be based on an RDF processor. In order to make the definitions in the catalogue as reusable as possible, they are all built hierarchically as tree structures. This gives the possibility to attach a template at every step and also to regroup already matched data structures into bigger structures, without touching the inner-working of the smaller structures.

The matcher takes as input an input graph G and one or several corresponding known structures from its catalogue and returns as output a tree structure (see Algorithm 1) with at least one pointer to a rendering structure from the Renderer (see below Section 4.4). The output is split into a set of multiple tree-structured sub-graphs with additional information about the template and a precedence number. All input data is eventually consumed through this process.

At this step, we like to mention that a pointer pointing back to the input graph is attached to the tree for every literal. This makes it possible to provide a simple form of data lineage, as well as to create applications which can write back to the input graph.

During this process, all information is treated equally except identical properties available in multiple languages. To provide language information in a transparent manner, the literals which are provided in multiple languages are grouped in the internal data structures. This enables the renderer component to select the correct language based on the context, without analyzing the graph again in more detail.

Catalogue of Matchers (Known Structures). We adopt a simple yet generic way to match RDF subgraphs and to generate trees. To initiate the matching process, providing an entry point in the RDF data is preferable. This *start resource* is a URI which denotes the node in the graph which will be used to begin the matching process. If no start resource is provided, the matcher falls back to try all nodes available to start the matching. Even though it is more flexible, this fall back approach needs more processing power compared to the former.

The matching process is implemented as a guided graph traversal. Based on the current start resource, all matching functions which are able to proceed through a predicate edge to the next node are listed. The set of functions which incorporates the highest amount of nodes is selected to be added to the set of trees. After this step, the matched nodes are deleted from the input graph and the same procedure starts over on the remaining nodes. The current solution is based on three types of matching functions:

Data: Graph G, Start Resource URIs S, Matcher Functions M_1
Result: Forest with References to Template Sets F
Function `Matcher()`
  foreach *Matcher Function* M_1 **do**
    Process G with Function;
    if *Result* **then** Add to Proposals P;
  end
  if *Proposals* P *available* **then**
    Find best P add to F;
    Delete Nodes covered by P from G;
    Call `Matcher()` with remaining G;
  else
    Add remaining Nodes each as Tree to F;
  end

Algorithm 1: Matcher Algorithm

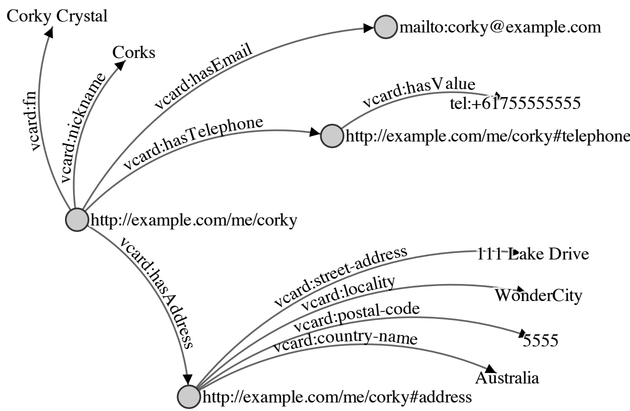

Figure 4: G: An example input graph.
(based on http://www.w3.org/TR/vcard-rdf/)

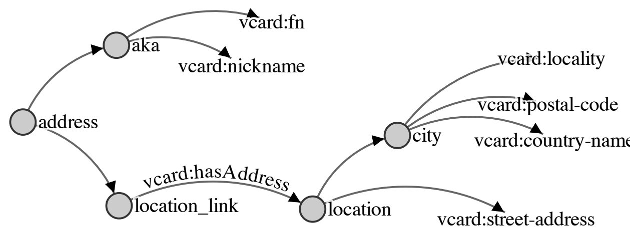

Figure 5: F: After the matching, the example tree structure which can be mapped on a template.

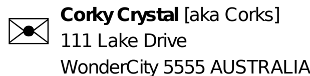

Figure 6: S: Final visualization after the rendering by a template T, including the context.

PredicateMatcher: this matching function only considers a predicate to match and a direction for the traversal; this can be illustrated through either one of these selecting SPARQL statements:
{<ex:startresource> <ex:predicate> ?O.} or
{S? <ex:predicate> <ex:startresource>.}

CombineMatcher: sets of triples are often regrouped in the datasets, e.g., for an address, which can consist of a locality, a zip code, a street, and so on. This matching function matches such sets by combining multiple functions (PredicateMatchers or other functions).

LinkMatcher: this matching function supports simple navigation inside an RDF graph. It allows to hop from a start resource to another URI, which is then defined as a new start resource. For this function also, it is possible, like for the PredicateMatcher, to specify if the resource URI is in the subject or object position.

```
list predicateMatchers:
  vcard_fn <vcard:fn>
  vcard_nickname <vcard:nickname>
  vcard_postal-code <vcard:postal-code>uu
  vcard_street-address <vcard:street-address>
  vcard_country-name <vcard:country-name>
  vcard_locality <vcard:locality>

list combineMatchers:
  vcard_address     as "address"
    ['vcard_locationLink','vcard_nameCombine']
  vcard_location    as "location"
    ['vcard_cityLine','vcard_street-address']
  vcard_cityLine    as "city"
    ['vcard_locality','vcard_postal-code',
     'vcard_country-name']
  vcard_nameCombine as "aka"
    ['vcard_fn','vcard_nickname']

list linkMatchers:
  vcard_locationLink as "location_link"
    on <vcard:hasAddress>
    ['vcard_location']
```

Figure 7: An example structure matcher with three different matching definitions: *vcard_*: ID of matcher, <*>: Predicate matched, *['*', ...]*: Referred IDs, *"*"*: Variable Name

New or more specific matcher type can be created or derived from the above matchers whenever necessary. As an example, we recently created a *SchemaMatcher*. It can incorporated RDF classes in the graph traversal in order to attach templates based on specific ontologies.

4.4 Adaptive Renderer

The adaptive renderer takes as input the tree structure given by the matcher (F, see for instance Figure 5), the available context variables (language, user, device) and the provided templates (T, see for instance Figure 8) to finally render the output. The templates are written in HTML and access the tree structure through escaped variable definitions (see Table 2 for details).

Before the input is combined with the template, the structure is prepared by a language subcomponent, which determines—based on the language context—which literals to use.

For each template pointer, multiple templates can be available. When no specific template is specified, a combination of the deepest underlying templates available in the hierarchy will be combined together. The templates in the deeper levels of the hierarchy will be chained in place. As a set of simple fallback templates are defined for all literals, it is hence assured that all provided data will be rendered at least in its simplest form.

The different templates have different flags attached denoting the context in which they have been prepared. It is the duty of the renderer to decide which is the best suitable template for the context at hand. The different flags describing the contexts can be on the input / output (device), the usecase, the targeted user, or the language. Those flags allow to

```
<div>
    <span class="glyphicon glyphicon-envelope"
         style="font-size:48px; float:left;
         margin: 0 15px 0 10px;">
    </span>
    <div>
    <b><%-address.aka.fn.u%></b>
    [aka <%-address.aka.nickname.u%>]<br>
    <%-address.location_link.0.location.street-address.u%><br>
    <%-address.location_link.0.location.city.locality.u%>
    <%-address.location_link.0.location.city.postal-code.u%>
    <%print(address.location_link.0.location.
            city.country-name.u.toUpperCase())%>
    </div>
</div>
```

Figure 8: T_1: The template used to render our example. This example deliberately does not reuse deeper templates. The variables that are structured in a tree object, are accessed inside the variable blocks (denoted through `<%-` and `%>`.)

easily change the rendering based on the context provided.

Context information is not always guaranteed to be available. The output device and language are usually the most readily available types of context information, typically followed by the user at hand. The decision on which template to use is solved by giving the output device the highest priority and enforcing it if available. The remaining available context variables are used to find the best match within the available templates. The template with the most matching flags is chosen by the system. If there is still some ambiguity, the priority is set as follows: Use-case, user, and finally language.

4.5 Extensibility

We mentioned in the introduction that our architecture was designed to be highly extensible. In the short introduction of the architecture, we saw how the behavior of the pipeline can be extended by adding new matcher definitions and, accordingly, template definitions. For the template definitions, the system can be extended dynamically, as multiple templates can be defined for the context variables per matcher definition. Thus, by adding new templates for new kinds of devices or for new users or user groups, the behavior can also be dynamically adapted based on the context.

On the source code level, it is possible to extend the different types of matchers by providing new matcher factories. A matcher factory gets initialized according to the matcher definitions that in turn populate the tree which is handed to the renderer.

4.6 Limitations

Uduvudu does not enable laypersons to create stunning visualizations through predefined off-the-shelf templates. Rather, it provides a holistic framework for developers, which in turn have the possibility to delegate data domain, UX and design tasks to the respective experts.

Furthermore, the framework proposes no functionality to enhance or reason on the input data. With the creation of user interfaces for RDF centric applications in mind, the pre-processing of the input graph needs to be provided by the application logic.

5. EDITOR

Our framework also supports a simple yet intuitive and powerful editor. As it is possible to render an output at any time, the editor is simply based on the results incorporating the available templates. Matchers and their corresponding templates can be added iteratively to create templates for bigger structures.

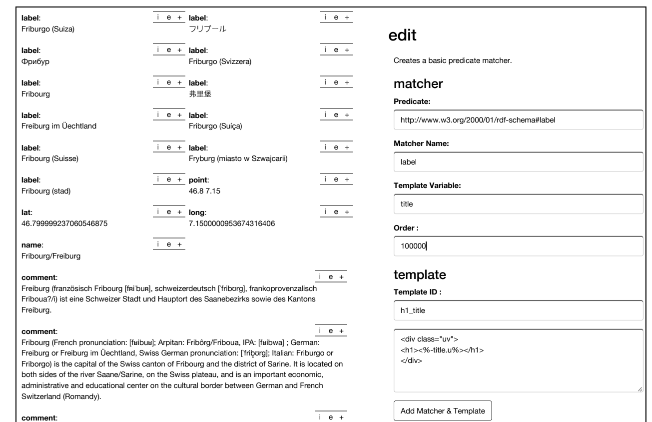

Figure 9: Rendering of *http://dbpedia.org/page/Fribourg* with no templates defined (left), in edit mode with *rdf:label* staged (right).

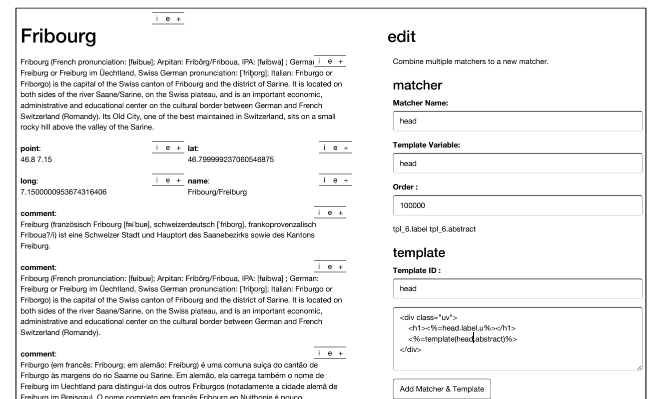

Figure 10: The same interface as in Figure 9 after the definition of a matcher and a template respectively for *rdf:label* and *rdf:abstract*. Further a new combine matcher merging the mentioned is staged.

Figure 9 shows on the left-hand side one resource (*http://dbpedia.org/resource/Fribourg*) rendered using the simple fallback template rendering. On the right-hand side, a first predicate matcher (*rdf:label*) is already staged; by simply clicking on the + icon (shown in edit mode only), the template was changed to render the literal as an HTML title.

Through the same mechanism (using the + icon), templates can be combined together to create more expressive tem-

Usage	Description	Example
Delimiters		
`<%- %>`	Output variable HTML-escaped.	`<%- label.u %>`
`<%= %>`	Output variable non-escaped.	`<%= html_desc.u %>`
`<% %>`	Execute JavaScript: Use print() for output.	`<% print(label.u.toUpperCase()) %>`
Variables		
`label.u`	Literal in context language.	`<%- label.u %>`
`abel.l.en`	Literal in specific language (lang tag).	`<%- label.u.ja %>`
`city.label.u`	Literal deeper in matched structure.	`<%- city.label.u %>`
`template(city.label)`	Rendered template for sub element.	`<%- template(city.label) %>`

Table 2: Overview of the most important template commands available.

plates. This is shown in Figure 10, where the *rdf:label* gets combined with a pre-existing *rdf:abstract* template to form a new *head* template.

In the staged example, the template of the abstract is reused (`<%-template(head.abstract)%>`), while the variable of the *rdf:label* is accessed directly (`<%-head.label.u%>`).

A more advanced editor allows users to define multiple templates for different contexts like different devices or languages. The matchers and templates created in the editor can be persisted in a public or private triple store. It is also possible to mix-in multiple matchers and template sources, which allows distinct users to create or adapt the templates to their own needs.

6. CURRENT DEPLOYMENTS

We now turn to the description of two current deployments of our system. We start below by describing how we used Uduvudu to build a user-friendly DBpedia front-end. Then, we review how we used it in the context of a large-scale project to render heterogeneous pieces of RDF data.

6.1 DBpedia Frontend

As a proof of concept, we used our framework to build a user-friendly DBpedia front-end. A live deployment of our front-end is available at `http://dbpedia.exascale.info`. The front-end takes as input an address describing a Wikipedia article. Typically, the rendering of DBpedia articles are optimized for engineers who need to explore the available data. In Figure 12, we provide as an example a rather technical rendering of all information about the City of Fribourg. WikiData provides a similar interface for their graph structure (see Figure 11).

Both interfaces are optimized for the exploration or maintenance of the underlying data itself.

The result obtained using our framework is shown in Figure 13. The main two differences with the other visualizations are i) the more user-friendly (and less technical) rendering and ii) the reuse of the rendering definitions (matcher and templates) for other datasets.

This use-case deployment shows how rendering information about a complex instance, like the city of Fribourg, can be made in an expressive and yet inexpensive process. For this

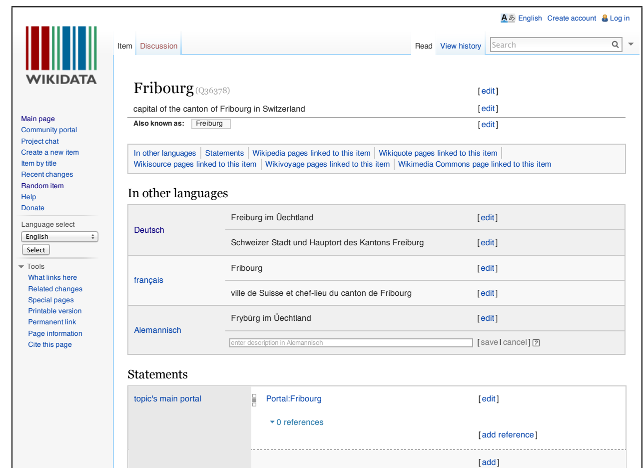

Figure 11: Wikidata interface regarding Fribourg. (fetched from https://www.wikidata.org/wiki/Q36378)

Figure 12: DBpedia interface regarding Fribourg. (fetched from http://dbpedia.org/page/Fribourg)

rendering, we prepared a set of matchers and templates in the context of cities, or populated places in general. The templates provided show different kinds of renderings and visualizations. Let us discuss the different template types built based on our generic architecture shown in Figure 13.

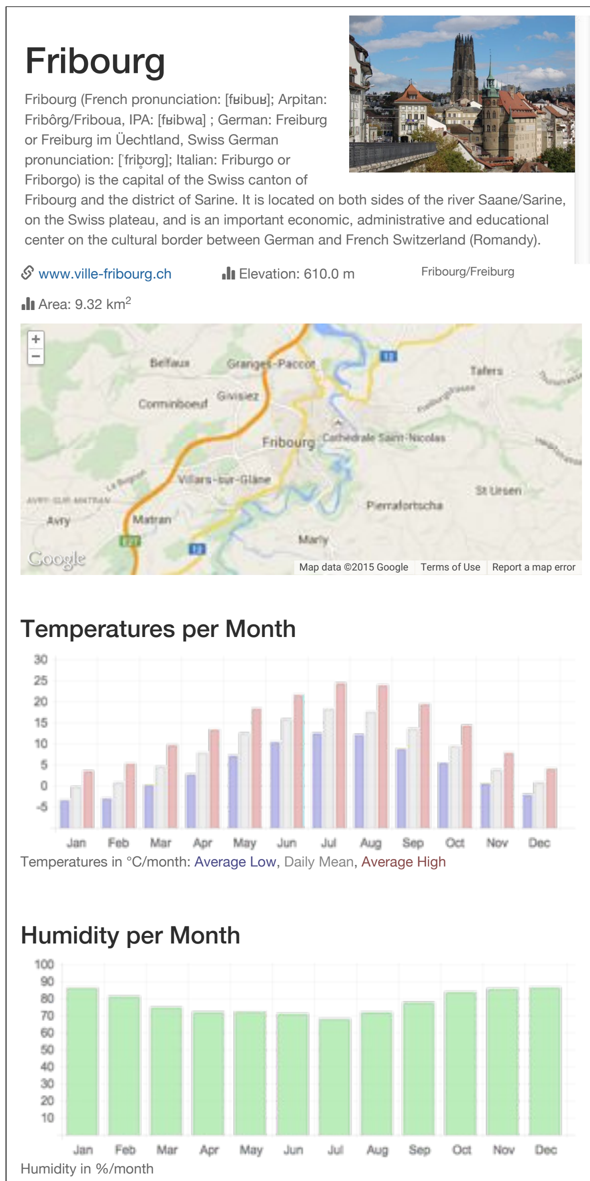

Figure 13: Rendering of *http://dbpedia.org/page/Fribourg* with more elaborate templates defined. Available at *http://dbpedia.exascale.info*

The bigger part of the rendering about Fribourg is made by mapping the information structure about a city. There are four generic templates used to generate the figure:

Picture with Title and Abstract: The first template extracts the picture and the labels of the entity, and creates a rendering with both elements integrated (with the title overlapping the picture). Furthermore, either a comment or abstract is shown to the right of the picture.

Selected Facts: Elevation, Area and the Website of the selected city are put together in a small facts box. Each of the facts could also be rendered independently. This means that the maximum of all available elements are rendered in all cases.

Map of Location: Coordinates that are attached to an instance are shown using a map widget. To initialize the widget, some JavaScript code gets injected on the fly.

Graphs of Humidity and Temperatures: The average mean, high and low temperature as well as the humidity per month are attached to the main instance as a set of 12 properties. They get combined by matcher and rendered by a template which loads a charting library.

Below the templates, we then see the list of non-matched properties. Each of them is internally represented as a tree with only one leaf. The rendering of such simple trees is defined by the built-in templates. By overriding this template, it is also possible to customize the fall-back representations. Furthermore, it is possible—on the base of those fall-back templates—to create different implementations of user interfaces which help in defining new matcher and template definitions.

6.2 Fusepool

Within the EU research project Fusepool[1], a Uduvudu component is used to render particular data views selected by the user in a dashboard. Fusepool integrates multiple, potentially large data sources using RDF as their data model. Example data sources used in this context are patents, Pubmed articles or user-contributed datasets like logs of parliamentary discussions or collections of beers and breweries. Among other things, Fusepool applies natural language processing on plain text content to extract entities, which can then be used for more specific search queries.

Fusepool provides a basic user interface, which makes it possible to support faceted search on top of the data. Based on the search results, the user might want to read the detailed content of a particular patent, pubmed article or specific

[1] *http://www.fusepool.eu/*

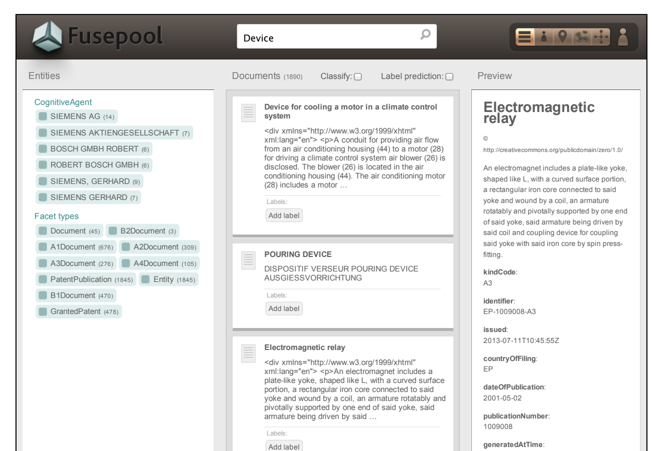

Figure 14: Fusepool dashboard where Uduvudu is used in the right-hand panel.

beer. This particular selection is then shown using Uduvudu. In the example shown in Figure 14, the title, abstract, and license properties have a matcher and template provided. The rest of the data properties are still presented in a generic way. If new data types are added to Fusepool, one just needs to provide specific matchers and templates to make sure Fusepool can then render the new data in a useful way. Besides providing the matcher and template, no interaction with a programmer who is familiar with the Fusepool code base is necessary. Both the matcher and the template can be stored in RDF as well as in the Fusepool graph store.

7. CONCLUSIONS

In this paper, we presented Uduvudu, a graph-aware, flexible and user-friendly UI Engine for Linked Data. Based on user input, Uduvudu captures and transforms inherent features of graph-structured data, which are then leveraged in the data-driven UI renderings. We described the uniquer features of our system, its architecture, and two of its recent deployments. By offering multiple ways of defining extensions based on a few basic templates, our solution is kept both user-friendly and generic enough to solve a wide variety of use-cases when rendering Linked Data. As the matchers and their accompanying template definitions are tailored to data structures and not to a particular UI use-case, the probability of reusing parts of the rendering definitions can be increased.

Furthermore, our architecture takes into account the fact that such rendering projects often involve multiple parties. The clear separation of tasks during the UI creation process and the possibility to work on the sub-tasks independently allows more agile workflows.

One of the main features of the architecture we propose is the ability to easily create context-aware renderings. Inherent support is given in order to optimize UIs for different devices, users, or languages. This functionality is mostly transparent to the developers and can also be leveraged after a context-insensitive solution is provided.

We provide an open-source implementation of our architecture as a complete framework on *http://www.uduvudu.org/*. The resulting system runs in a browser directly. For further implementation details, please refer to the provided website. We discussed two applications leveraging our implementation. The DBpedia front-end shows how our framework can be used to provide all kinds of rendering modalities. This was done either by leveraging the templates themselves or by injecting rendering libraries through the template system. In the context of the Fusepool application, our system was used to create the rendering of various heterogeneous pieces of data. The main requirement for this application was the ability to adapt the presentation layer without extending the application code.

Finally, we note that for certain applications it might be necessary to dynamically update the user interface based on changes made to the underlying Linked Data itself. This dynamic update of the presentation layer can be realized on top of the pipeline approach of Uduvudu. We can apply the paradigm known as reactive programming in that sense, where the implementation is built on data flow architectures. Reactivity can be directly achieved in our framework by re-submitting the input graph to Uduvudu on every change. Using networked-clients and the ability of Uduvudu to propagate changes made through the visual rendering of the data directly back to the data itself, one could build exciting new applications leveraging reactivity, where the graph representation on the client-side propagates updates to the data server, which in turn automatically updates all other user interfaces.

8. ACKNOWLEDGMENTS

We thank the teams in Fribourg and Biel for many fruitful discussions; special mentions go to Bart van Leeuwen, Pascal Mainini and Thomas Bergwinkl for their contributions to the architecture and development of Uduvudu. Uduvudu was partly funded by the EU Framework Program for Innovation under grant 296192.

9. REFERENCES

[1] Auer, S., Doehring, R., and Dietzold, S. LESS - template-based syndication and presentation of linked data. In *The Semantic Web: Research and Applications*, L. Aroyo, G. Antoniou, E. Hyvönen, A. t. Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds., no. 6089 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, pp. 211–224.

[2] Battle, S., Wood, D., Leigh, J., and Ruth, L. The Callimachus Project: RDFa as a Web Template Language. In *COLD* (2012).

[3] Berners-Lee, T., Hendler, J., and Lassila, O. The semantic web. *Scientific american 284*, 5 (2001), 28–37.

[4] Brunetti, J. M., Auer, S., García, R., Klímek, J., and Nečaský, M. Formal linked data visualization model. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services* (New York, NY, USA, 2013), IIWAS '13, ACM, p. 309:309–309:318.

[5] Hannes Gassert, and Andreas Harth. From graph to GUI: displaying RDF data from the web with arago. Tech. rep., Digital Enterprise Research Institute, 2005.

[6] Huynh, D. F., Karger, D. R., and Miller, R. C. Exhibit: Lightweight structured data publishing. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, p. 737–746.

[7] Khalili, A., and Auer, S. User interfaces for semantic authoring of textual content: A systematic literature review. *Web Semantics: Science, Services and Agents on the World Wide Web 22* (Oct. 2013), 1–18.

[8] Pietriga, E., Bizer, C., Karger, D., and Lee, R. Fresnel: A browser-independent presentation vocabulary for RDF. In *The Semantic Web - ISWC 2006*, I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. M. Aroyo, Eds., no. 4273 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2006, pp. 158–171.

[9] Schlegel, K., Weissgerber, T., Stegmaier, F., Granitzer, M., and Kosch, H. Balloon Synopsis: A jQuery plugin to easily integrate the Semantic Web in a website. *CEUR Workshop Proceedings 1268* (Oct. 2014).