End-user support for access to heterogeneous linked data

Hildebrand, M.

Citation for published version (APA):
Hildebrand, M. (2010). End-user support for access to heterogeneous linked data.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Configuring Semantic Web interfaces by data mapping

From the three case studies (Chapters 3, 4 and 5) it became clear that different types of user interaction are required and the search functionality and presentation methods of these need to be configured for the specific domain and task. In this chapter we propose a method to configure interface components for Semantic Web applications. We describe how the underlying data model of interface components can be formally defined, allowing Semantic Web application developers to configure a component using familiar RDF constructs. This chapter demonstrates how the search functionality and presentation methods of interface components can be configured for different domains.

This chapter was published as “Configuring semantic web interfaces by data mapping” in the Proceedings of the Workshop on Visual Interfaces for the Social Semantic Web (Hildebrand and van Ossenbruggen 2009) and was co-authored by Jacco van Ossenbruggen.

7.1 Introduction

Semantic Web data is typically rich in interconnections and highly heterogeneous. Designing user interfaces for applications that use this type of data is intrinsically hard. Designing interfaces for highly diverse data tends to lead to overly generic interfaces that do not communicate the richness of the data to the end user. On the other hand, interfaces that communicate this richness effectively tend to work well for only a set of fixed schemata and not for the entire dataset. Finding a sweet spot that balances these two forces is not trivial, especially if one takes into account that most Semantic Web developers are not UI specialists, and often have even little affinity with UI design. The problem is even deepened because
many Semantic Web developers tend to build UIs from scratch, as the fixed data model that is assumed by many off-the-shelf UI toolkits seems to conflict with the heterogeneity of their data.

In this chapter, we argue that for a wide range of applications, such a sweetspot can be identified and formally modelled in RDFS or OWL. By building standard interface components on top of this model, building an initial interface can be as easy as mapping the application’s data model to this interface model.

This approach has the advantage that it leverages the significant amount of design, implementation and testing effort already invested in today’s Web UI toolkits, and we believe that reuse of these commonly available toolkits will, in general, lead to better interfaces than interfaces that are designed and implemented from scratch by a (small) Semantic Web research team.

In addition, it replaces the traditional configuration and tailoring that is needed to adapt a generic interface to a local dataset by a straightforward RDF data mapping task, a skill most Semantic Web developers will have. By providing default mappings, it is even possible to provide a no-configuration, first crude version of a user interface, very early in the application development life cycle. This will give RDF data developers the “instant gratification” that has made many Web 1.0 and 2.0 applications so popular. It also encourages them to, during their data modelling and data development tasks, think about how their data will be used in the end-user applications, and how their modelling decisions may impact the interface.

Finally, while our approach provides pre-packaged solutions for common tasks, it does not prohibit application developers to go beyond those solutions in order to add more advanced or more application specific interface components. It is built to be extended or to build other layers on top of it.

This chapter is structured as follows. In the next section, we discuss related work and compare it to the approach proposed in this chapter. As an example user interface model and its binding to a Web UI toolkit, we discuss the interface model upon which the ClioPatria (Wielemaker et al. 2008) interface components are built, and how these components are implemented on top of the Yahoo! User Interface (YUI) library. We show how this model can be used to easily create two interfaces, one in the cultural heritage domain and one in the news domain. In the last section, we discuss the pros and cons of our approach.

### 7.2 Related Work

modelling part of the user interface in RDF is in itself not new. Fresnel (Pietriga et al. 2006) is a good example of an RDF vocabulary that can be used to specify the presentation details of the RDF data as they appear in the user interface. For interface widgets Fresnel forms a good solution to define the visualisation of individual data items within a widget. The Fresnel vocabulary can’t be applied
to for the configuration of widget specific properties as this requires a vocabulary specific for this widget.

Simile’s faceted browser Longwell supports Fresnel for the visualisation of the results (SIMILE 2005). In addition, the set of facets displayed in the interface can be defined in a configuration file in RDF. The individual facets are, however, not be configurable.

Web interface widgets have become a standard in web development. The choice among JavaScript libraries is numerous and all provide a convenient abstraction of low-level issues, such as cross-browser support. Interface widgets for semantic content are also available. Eetu Mäkelä et. al. provide several interface widgets that work on top of their ontology service infrastructure ONKI (Mäkelä et al. 2007). Example widgets are autocompletion and faceted navigation. They also so seem to strive for easy configuration of the widgets, but have not presented a clear model for this.

The approach of semantic widgets is also used by the Semantic Web company Mondeca.

7.3 Combining the Yahoo! User Interface Library with the ClioPatria Interface Model

In many domains there is a central role for persons, locations, times and artefacts. Sometimes these are modelled together as an event. For example, in the cultural heritage domain works of art are created by persons at a specific time and location. In a figurative art it may also be important to know which persons, times and locations are depicted. News images are also made on a specific time and place by a specific photographer and depict an event often involving persons, times and locations. Persons, locations and times are thus good candidates for a central model which is sufficiently generic, while sufficiently specific to answer the classic who, where and when questions to the end user. Man-made artefacts also play a central role in many domains and there specific properties can often be abstracted from by using, for example, a general vocabulary such as Dublin Core. In addition, different domains often deploy their own set of specific thesaurus concepts that describe the properties of events. We found that SKOS provides a sufficiently rich and abstract model to describe these concepts and their relations.

Within the semantic search and annotation framework ClioPatria (Wielemaker et al. 2008) we have developed several interface widgets. When deploying ClioPatria in a specific application domain we use persons, locations, times, artefacts and thesaurus concepts as an intermediate model between the interface model and the domain specific details of the underlying RDF data. In the following paragraphs
we explain the configuration dimensions of two of these interface widgets, auto-completion and faceted navigation, and show how this can be captured in an RDF interface model. In the next section we show how the intermediate model sketched above can be mapped to these widget’s interface models.

7.3.1 Example 1: Autocompletion

Autocompletion is an interface feature that allows users to type only a few characters instead of a full word or phrase. After the user has entered the first characters, the system responds by completing the word or phrase. If the characters typed in so far can be completed in more than one way, most interfaces present a list of multiple options. The user can then either select one of the options from the list, or continue typing to narrow down the number of options.

Figure 7.1: Autocompletion suggestions of historical persons. Underneath the name a short biography is displayed. This contains the nationality, role/profession and birth/death date. Note that for the first person listed, only the profession is available in the data. The abbreviation RMA, shown to the right, indicates the thesaurus source.

In context of the Semantic Web autocompletion is useful to quickly find a vocabulary term by one of its labels². In Chapter 3 (Hildebrand et al. 2009) we argued that for different tasks and data sets autocompletion widgets require a different configuration. The screenshots in Figure 7.1 and Figure 7.2 show autocompletion suggestions of historical persons and thesauri concepts. In the next section we discuss the configurations of these two widgets, here, we focus on the main differences between these two widgets. First, the widgets suggest a different type of term (e.g. persons and concepts), thus, requiring a different selection of the right RDF resources. Second, the persons are organised in an alphabetically ordered list, while the concepts are grouped by different thesauri and ranked

---

²For sake of simplicity we do not consider finding terms by a label of a related term.
Figure 7.2: Autocompletion suggestions of thesauri concepts. Results from both IconClass and WordNet are shown, each presented in a separate group. A secondary panel shows more information for the highlighted term (“[45K] siege, position war”), including the hierarchical structure the term is part of. The hierarchy contains the term itself in bold, its ancestors and the direct children. Images of the prints used with permission, courtesy of the Rijksmuseum Amsterdam.

according to popularity. Finally, the individual suggestions are visualised differently in each widget. The suggested persons are shown with extra biographical information, whereas, the concepts are shown within their hierarchy.

The two examples are built on top of the YUI autocompletion widget. The YUI widget contains several client side configuration parameters, it supports custom functions for result formatting, construction of remote data requests and it provides many event handlers. Although this is sufficient to configure the widget for an RDF data source, we experienced that it required extensive JavaScript programming to obtain the appropriate configurations. For example, visualising different types of information requires the configuration of the server request as well as the client side JavaScript formatting functions.
An interface model for an autocompletion widget provides a single focal point for the configuration of a widget and only requires Semantic Web modelling skills. Note, we do not claim that this is a complete model for autocompletion, we merely want to illustrate that it is possible to define an interface model for an autocompletion widget in RDF. The model we present is for an extended version of the YUI autocompletion widget (Hildebrand et al. 2007). We added support for clustered presentation of search results, a secondary display that is shown when the user hovers over a suggestion, a single configurable result formatting function and support for easy configuration of the server side search algorithm.

Figure 7.3: Interface model for ClioPatria’s thesaurus concept autocompletion widget. All organisation and visualisation properties are optional.

An interface model for the RDF concept autocompletion widget is shown in Figure 7.3. The widget contains three main configuration properties that correspond with the three phases of the search process: selection, organisation and visualisation. For the selection of the appropriate term it should be known what type of terms should be selected and which literals should be used to find these terms? The first is configured by providing an rdfs:Class for :target. The second requires the definition of a collection of RDF label properties for :match. The order

\[\text{AutocompletionWidget} \quad :\text{sort} \quad (\text{exact} || \text{inlink} || \text{rdf:Property}) \quad :\text{group} \quad \text{rdf:Property} \quad :\text{smush} \quad \text{true} || \text{false} \quad :\text{max} \quad \text{integer}\]

The properties and classes used in the interface models are contained in our own namespace http://e-culture.multimedian.nl/ns/interface/. In this chapter we omit this namespace and simply write a colon.
of the properties indicates which property has preferences in case the same term is found by multiple labels. The selected terms can then be organised in a list or in groups of different lists. The grouping is performed on the values of the RDF properties provided for :group. The terms in the list can be ordered according to several criteria and are defined in a collection as a value of :sort. The built in constant, exact puts all terms with an exact matching label before terms with partial matching labels. Another built in constant is inlink that sorts the terms by the number of incoming links they have in the graph. Further sorting criteria are the display labels (explained in the next paragraph) or any RDF Property. The number of results that are returned can be limited by defining :max. In a grouped organisation the maximum applies to the number of items within a group. Finally, terms that are defined as equivalent (owl:sameAs or skos:exactMatch) are shown as a single suggestion when :smush is set to true.

Figure 7.4: Layout of an autocompletion result. Primary display with a preLabel, the label itself, an alternative label between brackets, a postLabel aligned on the right side and a subLabel on a second line. The secondary panel provides additional space for larger content, such as images, descriptions and trees.

The results are visualised in a primary display panel. Besides the matching label itself the formatting function can print four additional labels: :preLabel, :altLabel, :postLabel and :subLabel. Figure 7.4 shows the skeleton of the primary display and an additional secondary result display. The secondary display, shown when the user hovers over a suggestion, provides place markers for an image, a longer piece of text and a tree containing the result.
7.3.2 Example 2: Faceted browsing

Facet browser interfaces provide a convenient way to navigate through a wide range of data collections. Originally demonstrated in the Flamenco system (Yee et al. 2003), facet browsing has also become popular in the Semantic Web community thanks to MUSEUMFINLAND (Hyvönen et al. 2005) and other systems (Huynh et al. 2007; m.c. schraefel et al. 2005; SIMILE 2005). An individual facet highlights one dimension of the underlying data. Often, the values of this dimension are hierarchically structured. By visualising and navigating this hierarchy in the user interface, the user is able to specify constraints on the items selected from the repository.

The facet browser we developed within ClioPatria, /facet (Chapter 4) (Hildebrand et al. 2006), can be applied to any RDF dataset. By considering the Class and Property hierarchy as special facets the user could configure the interface to her needs. In the Class facet the user selects the target objects (e.g. documents or persons) and from the Property facet she selects the facets she wants to navigate (e.g. creator and subject for documents or birth place and birth date for persons). This approach provides an instant interface for Semantic Web engineers. Presenting the raw data is, however, not suited for end user applications. In the project HealthFinland it was demonstrated that through careful user studies a more user-friendly configuration of the facets can be achieved (Suominen et al. 2007b).

Consider a faceted interface on a collection of documents. Each individual facet contains the values within one dimension. For example, one facet might display all the creators, whereas, another might display the subject categories. On an RDF data source this type of value selection corresponds to the values of a particular RDF property (e.g. dc:creator and dc:subject). Other selection criteria are also possible, such as all instances of a particular Class. In a similar fashion as the autocompletion suggestions, different types of facet values require different methods of organisation. The creators might be best organised in an alphabetically ordered list, while the hierarchical structure is important for the subject categories. Also the visualisation of facet values shows similarities with the autocompletion widget. Adding extra information in the display may help to disambiguate similar values. In addition, specific types of values (e.g. geographical locations and dates) are suited for alternative visualisation (geographical map and timeline).

When the number of defined facets is too large to be shown in the interface, it has to be defined which facets are shown. In Longwell a facet view can be defined as a collection of facets for a particular target. The facets defined in this view are shown, while all other facets are collapsed and available on requested. An alternative method is to allow multiple views and allow the user to select the most appropriate view. For example, the creation view contains all facets that cover the creation of a document, whereas, the content view contains the facets about the topic. In either solution, a view defines which facets are selected for display.
As an example we describe a possible model for a faceted navigation widget. A screenshot of this widget, used for a collection of news items, is shown in Figure 7.9. The widget displays multiple facets that are defined in a facet view and allows the selection of alternative views. The individual facets are built on top of the autocompletion widget, which allows re-use of its organisation and visualisation methods. In addition, it allows autocompletion within each facet. Again, we make the disclaimer that our purpose is not to provide a complete model for faceted navigation, but merely to illustrate that it is possible to define an interface model for a faceted navigation widget in RDF.

Figure 7.5: Interface model for ClioPatria’s faceted navigation widget.

An interface model for a faceted navigation widget is shown in Figure 7.5. The widget can contain multiple facet views, that each apply to the objects of a particular Class. Each facet view contains an RDF list of facets. A facet is configurable in the selection of facet values, the organisation and visualisation of these values. At the moment our widget only supports the selection of facet values from an RDF Property. The configuration of the organisation and visualisation is similar to that of the autocompletion widget. In addition, the facet values can be
organised hierarchically, meaning that initially only the root values of the hierarchy are shown and after selection of one of these its children become available.

7.4 Configuring interface widgets: a mapping task

Given an interface model the configuration of a Semantic Web application becomes a task of mapping the right properties and classes to this model. In practice, this often means first finding a suited intermediate model for the domain. For example, in terms of persons, locations and times. Second mapping this intermediate model to the widget’s interface model. We illustrate the mapping task with two use cases: configuring autocompletion components for an annotation application and configuring faceted navigation for a news application.

7.4.1 Use case 1: Rijksmuseum annotation user experiment

In Chapter 3 (Hildebrand et al. 2009) we developed a prototype interface for the subject annotation performed at the Rijksmuseum in Amsterdam, the Netherlands. The professional annotators of the Rijksmuseum describe thousands of artworks a year by assigning terms from controlled vocabularies. Finding the right term is complicated because the vocabularies used are large, very detailed, contain similar terms (or even duplicates) and often the annotator does not know exactly how to spell a term. We experienced that autocompletion helps professional annotators to find the right terms, but only when the widget is properly configured.

In an extensive study with these professionals we gathered the requirements for term search from multiple thesauri. During an iterative process of prototyping and discussion we tested several configurations of autocompletion widgets. A screenshot of the interface of the final prototype is shown in Figure 7.6. On the right side the interface contains three autocompletion fields to look-up terms from thesauri and a free text field to input dates. One of the results of the study is that the three autocompletion fields all required a different configuration.

The interface model for the autocompletion widget, as described in the previous section, is based on our findings at the Rijksmuseum. We acknowledge that a single study might not be sufficient to determine a complete interface model that applies to other domains. On the other hand, all three autocompletion fields required different features and configurations. Furthermore, the three fields cover generic types of terms (persons, thesaurus concepts and locations) that are very likely to be used in other domains as well.

We first introduce the vocabularies used in the annotation interface, before describing the configuration of the person and concept autocompletion fields. We used three thesauri with persons: Getty’s United List of Artist Names\(^4\) (ULAN),

\(^4\)http://www.getty.edu/research/conducting_research/thesauri/ulan/
Figure 7.6: Interface of the Rijksmuseum subject annotation interface. The four annotation fields in the right column are configured to support effective search in different thesauri. Image of the print used with permission, courtesy of the Rijksmuseum Amsterdam.

DBPedia’s RDF version of person data⁵ from Wikipedia (WP) and the Rijksmuseum’s own people thesaurus. All three thesauri were mapped to the generic “Person” scheme scheme of ULAN. For places we also used, Getty’s Thesaurus of Geographic Names⁶ (TGN) and aligned it with the Rijksmuseum’s place thesaurus. We used SKOS for the geographical containment relations in combination with location-specific properties from TGN. The concepts used in this domain were also modelled or mapped to SKOS. In addition to the Rijksmuseum’s events thesaurus we added the RKD IconClass⁷ thesaurus and, as a source for more general terms, W3C’s RDF version of Princeton’s WordNet⁸.

⁵http://dbpedia.org/
⁶http://www.getty.edu/research/conducting_research/thesauri/tgn/
⁷http://www.iconclass.nl/
⁸http://www.w3.org/2006/03/wn/wn20/
7.4.1.1 Autocompletion on persons

```turtle
:PersonAutocomplete
  a :Autocomplete ;
  :label "search person"@en;
  :label "zoek persoon"@nl ;
  :selection [ 
    :target ulan:Person ;
    :match (skos:prefLabel rdfs:label)
  ] ;
  :organization [ 
    :sort ("exact" :matchLabel);
    :smushing "true"
  ] ;
  :primaryDisplay [ 
    :subLabel ( 
      ulan:role 
      ulan:nationality 
      ulan:birthDate 
      ulan:deathDate 
    ) ;
    :altLabel skos:prefLabel ;
    :postLabel skos:inScheme
  ] ;
  :secondaryDisplay [ 
    :description ulan:biography ;
    :image vra:subject
  ] .
```

Figure 7.7: Person autocompletion allows autocompletion on instances of `ulan:Person`. The results are sorted first on exact matches and then alphabetically on the matching label. Results that are defined as equivalent (skos:exactMatch or owl:sameAs) are smushed. Each result is displayed with extra information. The primary display contains a short biography composed out of the values different properties and it contains the thesaurus source. The secondary display contains a full description and an artwork that depicts the person.

Figure 7.7 shows the configuration of the autocompletion widget in the `Who` field. The selection is restricted to terms of type `ulan:Person`. Note, the class of persons in the Rijksmuseum thesaurus and DBPedia people are subclasses of `ulan:Person`. We only consider literal values of skos:prefLabel and rdfs:label, where preference is given to the skos:prefLabel as this is first in the list. The results are organised alphabetically on the label and first showing all terms with an exactly matching label. The professionals at the Rijksmuseum explicitly indicated that they expect alphabetical ordering for a list of person names. As the autocompletion field gives
access to the terms from different overlapping vocabularies it turned out essential to smush equivalent results to a single suggestion.

The primary display contains three labels in addition to the matching label. The :altLabel is only shown in case the match was not found an a skos:prefLabel. Thus, when a hit is found by a skos:altLabel it's skos:prefLabel is also shown. The :endLabel contains the value of the skos:inScheme property. Thus indicating the thesaurus the term comes from. The professional annotators requested this information as terms are suggested from their own as well as other thesauri. The :subLabel shown beneath the main label is composed out of the values of four properties. Together these compose a short biography of the person. The annotators use this information to disambiguate similar persons from one another. The secondary display contains an image depicting the person and a longer biography.

```plaintext
:ConceptAutocomplete
  a :Autocomplete ;
  :label "search concept"@en;
  :label "zoek concept"@nl ;
  :selection [
    :target [ 
      owl:unionOf ( 
        ic:Concept ; 
        wn:Synset ; 
        rma:Event 
      )
    ];
    :matchLabel (skos:prefLabel rdfs:label) ;
  ];
  :organization [ 
    :sort ("exact" "inlink") ;
    :group skos:inScheme 
  ];
  :primaryDisplay [ 
    :subLabel skos:broader 
  ];
  :secondaryDisplay [ 
    :description skos:note ;
    :image vra:subject 
  ] .
```

Figure 7.8: Concept autocompletion allows autocompletion on instances of skos:Concept. The results are sorted first on exact matches and then on the number of in-links. The suggestions from the same thesaurus are grouped together. In the secondary display a tree is shown with the all ancestors and direct children of the result.
7.4.1.2 Autocompletion on thesaurus concepts

Figure 7.8 shows the configuration of the autocompletion widget in the What field. We only describe the configurations that are different from the Who field. The target is defined as an owl:union of three classes, ICONCLASS and WORDNET terms and the events from the Rijksmuseum thesaurus. The terms from the three thesauri are each shown in a separate group. The Rijksmuseum wanted to give preference to terms from ICONCLASS and only use WORDNET as a backup. Organising the results in different groups allowed the annotators to easily compare terms from the different thesauri to one and other. Within each group the results are ordered by the number of links that are pointing to the term. Intuitively, this means that the popular terms are shown first.

7.4.2 Use case 2: K-Space Semantic News Browser

ClioPatria is used to support search and browsing of news items (Troncy 2008). These news items are described with multimedia standards, news codes from the IPTC standard and additional metadata from various thesauri modelled (mapped) to SKOS. The additional metadata is acquired through extraction of named entities such as persons, organisations and locations, from the textual stories. The extracted named entities are mapped to existing resources available on the Web, such as locations from Geonames, and persons from DBPedia. The data set in this use case consists of news items from 2006, including the World Cup football.

A screenshot of the faceted interface from ClioPatria is shown in Figure 7.9. The top part contains four facets: document type, creation site, event and person. The result viewer, visible below the facets, contains news items related to the keyword “zidane”. The current query is shown in the header of the result viewer. The user can extend the query by selecting values from the facets. In this case the value “photo” is selected from the document type facet. The other facets only contain values that correspond with the current result set. Note, this prevents the user from constructing queries that lead to an empty result set.

Figure 7.10 shows an excerpt of the facet and facet view configuration for a news demonstrator. The creationSite facet applies to instances of the class NewsItem, as indicated by the value of the :facetTarget property. This facet will display values from the newsml:locCreated property. As the values are part of a geographical containment hierarchy, this is used for the organisation. In the screenshot of Figure 7.9 the creationSite facet it is visible that initially only the children of the hierarchy root (e.g. World) are shown (e.g. Europe, Africa and Asia). When one of these values is selected, the children available through the hierarchical relation, geo:parentFeature, become available. Four facets are grouped into a facet view that covers the content of news items. In a similar fashion other facets can be grouped into views on the production and document characteristics. The facet view menu shown in the screenshot on the top left allows the user to
Figure 7.9: Faceted interface of the NewsML demonstrator. Four facets are active: document type, creation site, event and person. The value “photo” is selected from the document type facet. The full query also contains the keyword “Zidane”, as is visible in the header above the results. Images used with permission, courtesy of INA.

select one of these facet views.

7.5 Conclusion and Future Work

We have shown how we can use RDF to model the interface widgets of a specific Web application, an abstract intermediate data model, and the mapping between these two models. We argue that this approach can provide developers with an interface early in the development cycle of a Semantic Web application. As long as the chosen widgets, associated interface model and intermediate model prove to be sufficiently rich, all the developer needs to do is to provide the mappings (in RDF) between his own data model and the intermediate model, using skills that Semantic Web developers can be safely assumed to possess. This approach also allows Semantic Web UIs to be built on top of existing Web tool kits, without sacrificing the heterogeneity and semantic richness of the underlying data.

A first drawback of our approach is that our interface models are typically specific for a given interface widget or toolkit. If the same RDF data needs to be displayed the same way in multiple interfaces, a vocabulary such as Fresnel, that
abstracts from the interface technology used, might be a better alternative. In our applications, we have aimed to fully exploit the functionality of the interface widgets, and have traded the advantages of extra functionality against generality. Other developers might make a different trade off.

A second drawback surfaces when a given set of widgets and the associated interface model provides insufficient functionality. Then, extensions will require traditional Web scripting skills to develop extensions to widget set, typically involving a mix of HTML, CSS and JavaScript. But it also requires skills to be able to model these extensions in RDF or OWL, and this combination of skills might
be hard to find.

For future work, we would improve upon our current interface model and its implementation. The current implementation is realised as an integral part of the ClioPatria server framework, and we are investing ways to be able to apply the same approach to create interfaces on top of arbitrary SPARQL endpoints.