Interacting with Linked Data via Semantically Annotated Widgets

Armin Haller[1], Tudor Groza[2], and Florian Rosenberg[3]

[1] CSIRO ICT Centre, armin.haller@csiro.au,
[2] University of Queensland, tudor.groza@uq.edu.au
[3] IBM T.J. Watson Research Center, rosenberg@us.ibm.com

Abstract. The continuous growth of the Linked Data Web brings us closer to the original vision of the Web as an interconnected network of machine-readable resources. There is, however, an essential aspect in principle still missing from this vision, i.e., the ability for the Web user to interact directly with the Linked Data in a read/write manner. In this paper we introduce a lifecycle and associated mechanism to enable a domain-agnostic read/write interaction with Linked Data in the context of a single data provider. Our solution uses an ontology to build a binding front-end for a given RDF model, in addition to RDFa to maintain the semantics of the resulting form/widget components. On the processing side, a RESTful Web service is provided to seamlessly manage semantic widgets and their associated data, and hence enable the read/write data interaction mechanism. The evaluation shows that the generation process presents no performance issues, while the content overhead required for the actual form-data binding is kept to a minimum.

1 Introduction

Over the course of the last five years, the progressive use of Semantic Web technologies in conjunction with the Linked Data principles [3] has led to an explosion of datasets being openly published on the Web. The emergence of this Linked Data Web [6] was foreseen in the very early conception of the World Wide Web itself. The original vision had, however, a second part that regarded the Web as a bi-directional communication channel between content producers and consumers [5]. In other terms, as opposed to today's read-only Web, which allows us only to act as simple viewers with respect to the published content or data, the read/write Web enables a direct interaction, as part of a common creative process.

From the pure textual created/consumed content perspective, social media environments such as Facebook, Twitter, blogs or wikis are close to fulfilling the vision. However, interacting with the large diversity of existing Linked Data in a read/write fashion over a typical Web application is still an open research topic. Some work has already been done in the area (e.g., [16,4,13]), in general by creating (HTML) forms as a medium between RDF data and humans. While this bridging concept represents probably the best solution, the underlying technical aspects of current approaches share a common drawback: they either require manual mappings between domain concepts and form components, or are specifically tailored for a particular domain (i.e., forms generated for a specific schema/ontology describing a dataset).

This paper presents a novel solution aimed to bring us a step closer to the original vision, by providing a lifecycle and associated mechanism to enable read/write interaction with Linked Data exposed by a single producer, via Web forms embedded in a widget. We use widgets as a manifestation paradigm because of their versatility and seamless integration possibility within diverse Web environments. The interaction takes place in a *local* dataset context (i.e., the context of a dataset exposed by the producer), however subject to the user's knowledge, it also takes advantage of the *global* Linked Data Web context. The proposed lifecycle is applicable to arbitrary RDF graphs (hence being domain-agnostic) and comprises three phases, described in the following via the direct contributions of this paper:

Widget generation. We propose a markup ontology that describes the structure of a Web form, the RDFa User Interface Language [12] (RaUL), for semantically defining Web widgets. We use RDFa to maintain the semantics of the widget in XHTML and for binding input data in Web forms/widgets to an RDF graph. A RaUL widget provides a binding to RDF data similar to what popular traditional web application frameworks (e.g. Ruby on Rails [19], GWT [10], Apache Struts [21] etc.) provide for a relational model. The data submitted through a semantically annotated widget is processed by our proposed ActiveRaUL client-side JavaScript (JS) API and send to the ActiveRaUL Web service where it is stored as RDF triples in the underlying database. If the same data is used by multiple Web widgets, it can be bound to the underlying widget elements by querying for the same uniquely defined resource. These relations (triples) can exists locally, or in the Linked Data Web.

Widget deployment. We provide ActiveRaUL, a RESTful Web service to seamlessly deploy and manage generated widgets. ActiveRaUL maps the four common HTTP methods, `POST`, `GET`, `PUT`, `DELETE` to corresponding CRUD operations (i.e., `CREATE`, `READ`, `UPDATE`, `DELETE`) on an RDF model in the backend. This approach relieves the Web developer from defining SPARQL queries to be executed when data is submitted in the Web widgets. Depending on the desired operation on the model, only the appropriate method in our Web service has to be called.

Widget usage. The Web user is able to transparently manipulate the underlying RDF data via traditional Web forms in a browser, while being unaware of the underlying data binding mechanism. Using a non-ambiguous RDF model as binding mechanism in RaUL forms allows the ActiveRaUL client-side JS API to retrieve data that already exists in the Linked Data Web (e.g., personal information about a user which is stored and published elsewhere) and pre-populated the form elements.

Following this lifecycle, a developer only needs to deploy the backend service[4] and to deal with the usual form styling elements, unless s/he explicitly opts for manually creating the widget model. In this case, knowledge about the RaUL ontology is required, however, even so, the widget model once created can be shared with and immediately adopted by others.

The remainder of the paper is structured as follows: Sect. 2 introduces a motivating/running example used in Sect. 3 to showcase the phases of the proposed lifecycle.

[4] The service is packaged as a Web archive and available for download at: http://w3c.org.au/raul/

Fig. 1. Examples of simplified forms used in a typical trading platform

Sect. 4 presents the ActiveRaUL service, its architecture and the client-side JS API. In Sect. 5 we benchmark the overhead introduced by RaUL annotations and the performance of the ActiveRaUL service, and before concluding in Sect. 7 we discuss existing related work in Sect. 6.

2 Running example

To illustrate our approach, we introduce a running example inspired from a typical e-Commerce interaction. Trading platforms are ideal candidates for exposing their product datasets on the Linked Data Web. Pioneering examples already exist, such as `http://www.bestbuy.com`, which uses RDFa and the Good Relations ontology [14] to publish its product information as Linked Data. In addition, due to the very nature of the domain, one can truly exploit the benefits of the Linked Data Web, as product instances in one site may have slightly different or extra information on other sites. Hence, being able to link the different instances brings an added value for the end-user.

Our example simulates a second-hand goods trading platform at which registered users can sell and buy second-hand goods [5]. For demonstration purposes, and later for understanding the value provided by our approach, we consider a scenario in which a typical user performs three basic (yet common) operations: registration, selling a product and buying a product. Fig. 1 depicts these three operations by means of simplified widgets generated by our solution from the underlying RDF data supposedly used by the trading platform. Some additional details on the three operations are presented as follows:

User Registration. A new seller/buyer has to register a new user account on the e-Commerce website. The user requests the registration form, which offers him the standard fields to fill in his data (Fig. 1, part A). One option would be to directly use the FOAF [7] profile.

Product Advertisement. A user, once registered, may add products to be sold (Fig. 1, part B). The associated form / widget consists of examples of common fields used to describe a product. Similar to the first operation, the user could re-use an existing description of the product on the Linked Data Web, by entering the product's URI.

[5] The example can be tested online at `http://w3c.org.au/raul/demo.html`

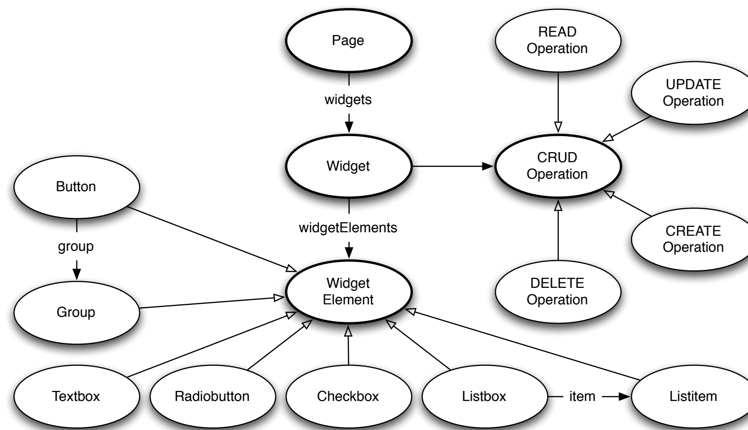

Fig. 2. The RaUL form model

Bid on a Product. Finally, the user may want to purchase an advertised product. The product can be easily found by filling in the form / widget depicted Fig. 1, part C.

In the remainder of the paper we show how by starting from the underlying RDF representation of the product and using RaUL, we are able to generate semantically annotated front-end widgets to uniformly identify the data input, and to link it to the Linked Data Web. At the same time, we show how to manipulate the data stored in the trading platform's triple store over the ActiveRaUL RESTful service interface.

3 A framework for data publishing in RDFa

In this section we detail the three phases of our proposed lifecycle, i.e., widget generation, deployment and usage, all using as back-end the ActiveRaUL service presented in Sect. 4.

3.1 Widget Generation

The first step of the lifecycle creates a widget model in RDF, using the RaUL ontology. The widget generation task is performed by the developer, who, subject to the chosen option (i.e., manual or semi-automatic), may need to be familiar with the RaUL ontology. However, this is done once for the lifetime of the site and can be compared with the creation of a form (widget) in HTML or a template in Web application frameworks like Ruby on Rails [19], GWT [10], Apache Struts [21] etc. However, in contrast to HTML forms that are usually custom-build for every website, RaUL-based widget models are reusable RDF graphs which are assigned a URI by ActiveRaUL. As such, they can be reused and can become standardised widget models for certain tasks themselves. Using the ActiveRaUL framework, there are two options to create a widget: (i) manually, by posting a handcrafted RaUL-based widget model (in RDF/XML, RDF/JSON or RDF/N3) to the ActiveRaUL service;

```
<!DOCTYPE rdf:RDF [
  <!ENTITY product
   "http://w3c.org.au/raul/service/public/forms/addproduct"> ]>
<rdf:Description rdf:about="@product#currency">
  <rdf:type rdf:resource="http://purl.org/NET/raul#Listbox"/>
  <id xmlns="http://purl.org/NET/raul#">transaction_type</id>
  <value xmlns="http://purl.org/NET/raul#">@product#value_currency
  </value>
  <list xmlns="http://purl.org/NET/raul#">@product#currency_list
</list>
</rdf:Description>

<rdf:Description rdf:about="@product#currency_list">
  <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns
          #Seq"/>
  <rdf:_1 xmlns="http://purl.org/NET/raul#"
   rdf:resource="@product#currency_1"/>
  <rdf:_2 xmlns="http://purl.org/NET/raul#"
   rdf:resource="@product#currency_2"/>
</rdf:Description>
<rdf:Description rdf:about="@product#currency_1">
  <rdf:type rdf:resource="http://purl.org/NET/raul#Listitem"/>
  <label xmlns="http://purl.org/NET/raul#">USD</label>
  <value xmlns="http://purl.org/NET/raul#">USD</value>
</rdf:Description>
<rdf:Description rdf:about="@product#currency_2">
  <rdf:type rdf:resource="http://purl.org/NET/raul#Listitem"/>
  <label xmlns="http://purl.org/NET/raul#">EUR</label>
  <value xmlns="http://purl.org/NET/raul#">EUR</value>
</rdf:Description>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0//EN" [
  <!ENTITY product
   "http://w3c.org.au/raul/service/public/forms/addproduct"> ]>
<div about="@product#currency" typeof="raul:Listbox">
  <span style="display:none;">
    <span property="raul:id" content="transaction_type"/>
    <span property="raul:value" content="@product#value_currency"/>
    <span property="raul:list" content="@product#currency_list"/>
  </span>
</div>
<select id="currency" name="currency">
  <option value="USD">USD</option>
  <option value="EUR">EUR</option>
</select>
<ol style="display:none;" about="@product#currency_options">
  <li rel="rdf:_1" resource="@product#currency_1"></li>
  <li rel="rdf:_2" resource="@product#currency_2"></li>
</ol>
<div about="@product#currency_1" typeof="raul:Listitem">
  <span style="display:none;">
    <span property="raul:label" content="USD"/>
    <span property="raul:value" content="USD"/>
  </span>
</div>
<div about="@product#currency_2" typeof="raul:Listitem">
  <span style="display:none;">
    <span property="raul:label" content="EUR"/>
    <span property="raul:value" content="EUR"/>
  </span>
</div>
```

Fig. 3. A RaUL Listbox in RDF/XML (left) and the generated XHTML+RDFa (right)

or (ii) (semi)-automatically, by posting an arbitrary RDF graph to the ActiveRaUL service which will then create the corresponding RaUL-based widget model.

The RaUL ontology[6] (defining the widget model) consists of two parts: (i) **Form controls** describing the structure of the widget, and their associated operations (i.e., `READ`, `UPDATE`, `CREATE` or `DELETE`), and (ii) a **Data model** defining the structure of the exchanged data as RDF statements which are referenced from the *form model* via a data binding mechanism. The model gives meaning to the data used in the *form controls* by uniquely referencing standard ontologies on the Semantic Web.

Form Controls. A *form control* in RaUL is an element that acts as a direct point of user interaction and provides access to the triples describing the *data model*. Fig. 2 depicts a high-level overview of the RaUL form model defining a set of *form controls*. As most of the concepts have a self-explanatory name we refer the interested reader to our earlier publications [12,11] for more detail on the ontology itself. It is worth mentioning, however, that the controls have corresponding XHTML elements which are used when generating the widget for rendering and interaction purposes. Fig. 3 shows the RaUL representation of a *Listbox* and its corresponding XHTML created by the ActiveRaUL service, part of the *Product Advertisement* widget introduced in our running example in Sect. 2.

Data Model. In contrast to untyped key/value pairs used in traditional XHTML forms, data in RaUL widgets is submitted in a structured way, as RDF data according to a certain schema. Hence, the data model is de-coupled from the form controls. Handcrafting the form–data mapping provides the dataset developers with full flexibility in defining the structure of the model. Every form control in the widget maps to (a set of) triples that describe the input. For example, the form controls composing the *User Registration* widget in our running example (Fig. 1 part A in Sect. 2) could be mapped using the FOAF ontology and the W3C Time ontology [15] as follows: (i) *FOAF URI* – `foaf:Person` (ii) *First name* – `foaf:givenName` (iii) *Last name*

[6] RaUL can be found at `http://purl.org/NET/raul#`

```
/* Defined Reification Mapping */
<span about="#valuefirstname">
  <span property="rdf:subject" content="http://..." />
  <span property="rdf:predicate" content="foaf:givenName" />
  <span property="rdf:object" content="" />
</span>
```

Fig. 4. RDFa reified triple for a foaf:givenName object.

– `foaf:familyName` (iv) *Email* – `foaf:mbox` (v) *Birth day* – `time:day` (vi) *Birth month* – `time:month` (vii) *Birth year* – `time:year` . Similarly, the mappings behind the other two widgets could be handcrafted, using for example the GoodRelations ontology [14].

The actual binding of the *form controls* to the underlying data structure is realised via reification within the RDFa embedded in the resulting XHTML form control content. Fig. 4 shows, for example, how the *firstname Textbox* in the RaUL *User Registration* widget references the RDF triple representing the corresponding underlying data element (i.e., *<http://...><foaf:firstName>* <`''`>). The `rdf:subject` triple references the URI assigned by the ActiveRaUL service for the instance graph, the `rdf:predicate` triple is a reference to the URI of a standard Web ontology property, and the `rdf:object` triple is a reference to the value that can be edited by the respective form control. Empty `rdf:object` fields serve as place-holders and are filled at runtime by the ActiveRaUL client-side JS API with the user input.

Semi-automatic widget generation. To assist the developer/ontology engineer in the creation of a RaUL widget model, we propose a mapping framework to semi-automatically derive semiotics according to our RaUL user interface model from arbitrary RDF graphs. Although it can be foreseen that such user interface (widget) models become part of the Linked Data Web, there are no standard models available yet. Thus, the ActiveRaUL service includes a generation algorithm that creates a best-effort widget model on any deployed RDF graph. Figure 5 briefly outlines our algorithm. The *RaUL-generation* function takes as input the URI (U) created by the ActiveRaUL controller for the new widget model (see Sect. 4.1) and a ground RDF graph G with no blank nodes (if blank nodes exist, they are discarded). The algorithm iterates through all unique subject URIs and creates a corresponding RaUL *Widget* (line 6-8). The *trim-fragment()* function creates a fragment from any input URI and ensures that there are no duplicates. Then, for every predicate in G, where the subject is a URI reference, a RaUL *WidgetElement* is created. The actual type of the *WidgetElement* is determined as follows: if the predicate p_x exists only once in G and if the object is a `Literal` and the XSD datatype is not `boolean` then a RaUL *Textbox* is created (line 11-14). Also the associated reified triple referenced through the value property (line 12) is created with the *create-reified-triple()* function (line 13). The value of the object o_x in G is inserted as the value for the object of the reified triple. If the datatype of the object o_x is `boolean` then a RaUL *Group* (17-19) is created and two RaUL *Radiobuttons* are created (line 20-24). If a predicate p_x occurs more than once in G and if the object o_x is a `Literal`, a RaUL *Listbox* is created (line 26-31), a RDF sequence with the number of predicates p_x (line 32) is inserted and for each predicate a RaUL *Listitem* with the value of the object literal in G is created. If the object o_x is a `Literal` then a *raul:label* with the value of o_x is created (line 38-39), if the object is a URI reference and if the URI reference is in the local graph, we follow it and check for the existence of a triple with a predicate

rdfs:label. If it exists we use the object o_y of this triple for the *raul:label* property of the *Listitem* (line 42-43), otherwise, we use the URI reference (line 44-45). For every *WidgetContainer* we create a RaUL submit *Button* (line 51).

The resulting RaUL widget model is currently meant for assisting the developer, but in most cases he will need to refine the model which can be retrieved via its URI assigned at generation time by ActiveRaUL. However, additionally to the best-effort generation of *form controls*, a correct data binding between the *form controls* and *data model* is ensured, significantly easing the effort of the developer in defining/refining the widget model.

3.2 Widget Deployment

The ActiveRaUL service offers an endpoint to deploy a widget with a `POST` request to its `/public/forms` resource. The developer has two options for the payload when deploying a widget depending on the chosen generation path: (i) a handcrafted **RaUL-based widget model** that can be deployed within the public namespace `/public/forms` of the ActiveRaUL service backend (to support the re-use of generic form models in other Semantic Web applications and enable the emergence of standard widget models – e.g. a user registration form shared between many sites), and (ii) an **Arbitrary RDF graph**, which sent to the same endpoint will trigger the ActiveRaUL service to perform a best-effort generation of a widget according to the process described above.

The ActiveRaUL service supports different data representations, such as RDF/XML, RDF/JSON or RDF/N3. If the `POST` request is successful, the service will return the URL of the newly created widget in the HTTP `Location` header of the response, for example, `/public/forms/addproduct`. The resource name `addproduct` is automatically generated from the URI of the widget class in RDF.

3.3 Widget Usage

Once a widget has been deployed with ActiveRaUL by the developer, a Web user can access and use it through a browser. The browser issues an HTTP `GET` request to the URL that was created in the widget deployment phase to retrieve the widget. It depends on the HTTP `Accept` header in the `GET` request to determine what content type is sent back to the client. In the case of accessing the URL through a browser the response content type is XHTML.

Whenever the Web user fills in the form or changes already existing data in a form, the ActiveRaUL client-side JS API processes the form (details in Section 4.2) at submission time and sends the appropriate HTTP message (derived by interpreting the type of *CRUDOperation* in the RaUL widget model) to the ActiveRaUL service. The client-side JS API distinguishes several cases when updating the `rdf:object` in the reified value triple depending on the type of the form control. For example, the values provided in *Textboxes* are directly written to the `rdf:object` in the value triple. *Listboxes*, on the other hand, are more complex. After the submission of a *Listbox* form control the client-side JS API creates a *checked* relation for all selected *Listitems*. Additionally, it writes the reference to the object describing the *Listitem* into the `rdf:object` of the value triple. If the *Listbox* is a multi select one, defined by its *multiple* property, the referenced reified triple in the value property must be

```
------------------------------------------------------------------------------------------
 0: RaUL-generation(U, G )
    Where U is the URI assigned to the new widget model by the ActiveRaUL controller
    Where G is a ground RDF graph (no blank nodes) containing a set of triples (s,p,o) with
    s ∈ URIs , p ∈ URIs, o ∈ URIs ∪ Literals, where
    s is the subject, p the property and o the object of the triple and, where
    G(s_u) is the set of unique subjects in G.
 1: Create empty widget model W
    W(G) = I
 2: OPEN.enqueue( (s_l, p_l, o_l), ..., (s_m, p_m, o_m) )
 3:   WHILE OPEN ≠ ∅
 4:     (s_x, p_x, o_x) = OPEN.dequeue()
 5:     IF (s_x, p_x, o_x) == END, report success and return W
 6:     FOR each s_x in G(s_u)
 7:        U_wc = U . trim-fragment(s_x)
 8:       W(G) = W(G) ∪ (U_wc,rdf:type,raul:WidgetContainer)
 9:       IF | p_x | in G == 1 AND if o_x ∪ Literals AND literal-type(o_x) is not boolean
10:         U_tb = U . trim-fragment(p_x)
11:         W(G) = W(G) ∪ (U_tb,rdf:type,raul:Textbox)
12:         W(G) = W(G) ∪ (U_tb,raul:value,value-reference(o_x))
13:         W(G) = W(G) ∪ (create-reified-triple(value-reference(o_x)),p_x,o_x)
14:         W(G) = W(G) ∪ (U_tb,raul:label,trim-fragment(p_x))
15:       ELSEIF | p_x | in G == 1 AND if o_x ∪ Literals AND literal-type(o_x) is boolean
16:         U_gr = U . trim-fragment(p_x)
17:         W(G) = W(G) ∪ (U_gr,rdf:type,raul:Group)
18:         W(G) = W(G) ∪ (U_gr,raul:value,value-reference(p_x))
19:         W(G) = W(G) ∪ (create-reified-triple(value-reference(p_x)),p_x,o_x)
20:         U_rbt1 = U . trim-fragment(U_gr)."1"
21:         U_rbt2 = U . trim-fragment(U_gr)."2"
22:         W(G) = W(G) ∪ (U_rbt1,rdf:type,raul:Radiobutton) ... same for U_rbt2
23:         W(G) = W(G) ∪ (U_rbt1,raul:group,U_gr ) ... same for U_rbt2
24:         W(G) = W(G) ∪ (U_rbt1,raul:label,trim-fragment(p_x))... same for U_rbt2
25:       ELSEIF | p_x | in G > 1 AND if o_x ∪ Literals
26:         U_lb = U . trim-fragment(p_x)
27:         W(G) = W(G) ∪ (U_lb,rdf:type,raul:Listbox)
28:         W(G) = W(G) ∪ (U_lb,raul:value,value-reference(p_x))
29:         U_ll = U . create-listuri(U_lb)
30:         W(G) = W(G) ∪ (U_lb,raul:list,U_ll))
31:         W(G) = W(G) ∪ (create-reified-triple(value-reference(p_x)),p_x,o_x)
32:         FOR each p_x
33:           i++
34:           U_i = U . trim-fragment(U_lb).i
35:           W(G) = W(G) ∪ (U_ll,create-rdf-seq-element(p_x),U_i)
36:           W(G) = W(G) ∪ (U_i,rdf:type,raul:Listitem)
37:           W(G) = W(G) ∪ (U_i,raul:value,o_x)
38:           IF o_x ∈ Literals
39:             W(G) = W(G) ∪ (U_i,raul:label,o_x)
40:           ELSE
41:             IF o_x in G(s_u)
42:               IF exists  p_y  in G where s == s_u AND p_y == "rdfs:label"
43:                 W(G) = W(G) ∪ (U_i,raul:label,o_y)
44:               ELSE
45:                 W(G) = W(G) ∪ (U_i,raul:label,o_x)
46:               ENDIF
47:             ENDIF
48:           ENDIF
49:         ENDFOR
50:       ENDIF
51:       W(G) = W(G) ∪ (U,rdf:type,raul:Button)
52:     ENDFOR
53:   ENDWHILE
54:   RETURN Failure
------------------------------------------------------------------------------------------
```

Fig. 5. RaUL widget model generation algorithm

an RDF collection, whereby all selected *Listitem* references are written to the value triple.

In our running example, when a Web user fills in the *Product Advertisement* widget and submits it, the ActiveRaUL client-side JS API issues a `POST` request to the resource `/public/forms/addproduct/`, with a payload consisting of the RDF triples that are parsed from the RDFa annotations and the user input data. The ActiveRaUL service processes the request, inserts the data in the RDF triple store, and sends the URI of the newly created resource for the submitted data, e.g., `/public/forms/addproduct/101` in the HTTP `Location` header back to the client. This uniquely identified data can then be accessed with different widgets, as long as the data binding uses similar URI references for the `predicates` in the value triple. For example, when a user posts a certain product for sale, this data can also be retrieved in the widget which displays the product for sale.

3.4 Data Reuse

An important feature in this lifecycle is the reuse of existing data. Often, the data to be provided in a widget is already present on the Linked Data Web, e.g., a FOAF file describing a person, which usually includes many of the properties required by a registration form. The relation-based data binding implemented by RaUL form controls, via standardised ontologies (e.g., *foaf:givenName*), enables a direct re-use of such Linked Data. The user can hence point, for example, to an existing FOAF file and ActiveRaUL will automatically fill in the corresponding widget controls.

All widget controls designed to reference an *owl:sameAs* relation are treated as reference to external data. From the XHTML rendering perspective, these widgets correspond to *Textboxes* with an associated `update` *button*. At runtime, the Web user can directly provide a URL to an existing Linked Data resource in the *Textbox* which is used by the client-side JS API to retrieve the RDF graph and pre-fill the form controls in the widget if the data exists in the resource graph. Alternatively, the user can type in a search term which the client-side JS API uses for a call to the Sindice API [18] which in turn returns an RDF graph that is again used to pre-fill the remaining form controls. In either case, if a URI to an external resource is provided the underlying *owl:sameAs* relations for this form control asserts that the graph representing the user's input data its URI assigned by ActiveRaUL is the same individual having a different URI in the Linked Data Web.

An illustrative example of the use of form controls that reference external data is shown in the *User Registration* widget (Fig. 1 part A in Sect. 2), where the **FOAF URI** control was mapped to a `foaf:Person`, which enables the form to auto-fill the rest of the controls according to the given schema. Similarly, in the *Product Advertisement* widget (Fig. 1 part B in Sect. 2) the **Product URI** could be mapped to a GoodRelations `gr:Offering` object.

4 The ActiveRaUL system

The lifecycle introduced in Sect. 3 relies, particularly for the deployment and usage phases, on the ActiveRaUL backend[7], described in this section. The backend consists

[7] ActiveRaUL is available at `http://w3c.org.au/raul/service`

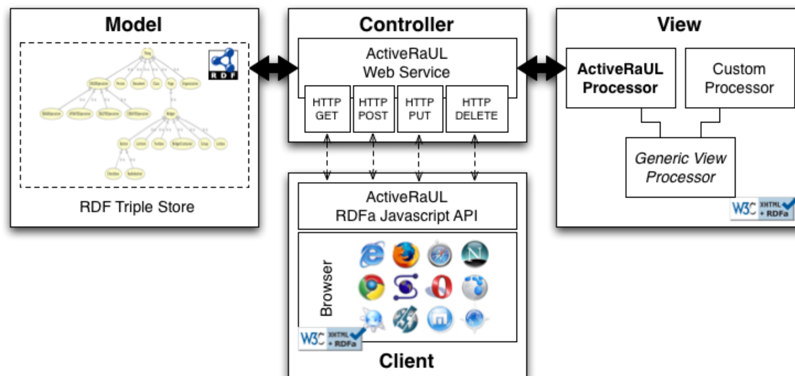

Fig. 6. Architecture of ActiveRaUL

of two main parts, as depicted in Fig. 6: (i) a RESTful Web service, and (ii) a client-side JS RDFa API. In the following, we detail the technical details of both the RESTful Web service, as well as the client-side JS API.

4.1 ActiveRaUL RESTful Web service

The RESTful Web service provides a uniform way to manage widgets and the data that is processed by the widgets. It abstracts from specific low-level details, such as storing and querying RDF data by interacting with an RDF triple store. This enables Web developers to use and integrate the ActiveRaUL framework without requiring a deep understanding of the core Semantic Web technologies (e.g., SPARQL, triple stores, etc.). Additionally, the service provides a mechanism to produce and render widgets via RDFa annotated XHTML forms. As shown in Fig. 6, the ActiveRaUL service architecture implements a Model-View-Controller (MVC) pattern [20].

Model. The *form controls* and their associated *data model* constitute the *model* part of the ActiveRaUL system. The service uses the RaUL ontology for defining widgets. However, its implementation is generic and can accommodate diverse models, as only the plugable rendering part is depending on the RaUL ontology. ActiveRaUL currently uses OpenRDF Sesame[8] as a triple store to persist all RDF data. However, any other triple store can be plugged into the backend with little modifications.

View. The view in MVC is the part that the user sees. In ActiveRaUL, the model can be queried using different representations, e.g., RDF/XML, RDF/JSON, RDF/N3 and XHTML+RDFa. The first three representations are not meant for human consumption and as such, do not require the generation of a view. ActiveRaUL provides serialisations according to the content type requested in the HTTP header.

Only the XHTML+RDFa representation is meant for human consumption. Its rendering as a widget is dependent on the underlying widget model, in our case RaUL. However, to keep this view generic, the `GenericViewProcessor` component is provided as a Java interface that defines method signatures for the rendering functionality specific to a particular widget model. For RaUL, we provide an `ActiveRaULProcessor` that implements the view generation based on the RaUL

[8] `http://www.openrdf.org/`

model. The view generation is triggered once a client requests a view via the controller (using an HTTP `GET` request with `Accept` header MIME type set to `application-/xhtml+xml`). The view generation then traverses all the form controls and generates the necessary XHTML+RDFa (see Section 3). The widget layout is built via CSS references.

Controller. The *controller* is responsible for creating, updating and deleting widget definitions and associated data required to be processed and stored as part of a form submission. The *controller* also assigns URIs to the submitted resources and returns the URL in the HTTP `Location` header of the response, for example, `/public/forms/addproduct`. To deal with duplicate names, unique numbers are appended at deployment time, e.g., `addproduct1`.For the instance data submitted for a widget, the ActiveRaUL service dynamically assigns a URI, such as the `/public/forms/addproduct/101` we mentioned above.

Errors are handled by returning the status code 500 (Internal Server Error) if an unexpected error happens. For errors related to wrong or incorrect input (payload), the service returns a status code 400 (Bad Request). Incorrect URL parameters will result in a 404 (Not Found).

User Authentication. In the URL scheme of the ActiveRaUL service the `/public/` part of the resource identifier is essentially a reference to a user id. As the name implies, `public` represents an open access space to upload forms and data. Any other {`userid`} parameter in the /{`userid`}`/forms` resource represents the user id that has access to the forms under this specific `forms` resource. Access to any {`userid`}, but the `/public/` identifier, is only granted if the necessary HTTP authentication credentials are present in the corresponding HTTP requests.

4.2 ActiveRaUL client-side JS API processing

The current implementation of the ActiveRaUL client-side JS API uses the rdfquery[9] library as an RDFa parser and jQuery for handling and querying the XHTML DOM tree. The actual processing consists of two main steps: (i) *data binding*, and (ii) *RDFa parsing and server communication.*

The data binding is performed immediately prior to parsing the RDFa. If data is retrieved from the Linked Data Web as described in Section 3.4, the data fetched from the external RDF Graph is treated as if it was provided by the user through the form. The update of the form controls is done by querying the received RDF for the object values of the predicates defined in the reified triple of the widget, which are then replaced with each successful query result. The processing of the data binding is done directly over the XHTML DOM tree using the jQuery library. For each form element the reified triple is identified by its URI and the respective object is replaced with the user input value.

After the data binding operations, the documents is parsed to extract the RDF triples from the XHTML+RDFa representation and the full RDF graph is sent to the ActiveRaUL service. To determine which operation to invoke in the ActiveRaUL service, the client extracts the invocation URL and the HTTP method (from the type of the *CRUDOperation* defined in the RDFa annotations).

[9] `http://code.google.com/p/rdfquery/`

Form Element	XHTML		XHTML+RDFa		# triples		Overhead in %	
	min	max	min	max	min	max	min	max
Widget	72	115	251	376	3	6	248%	226%
Textbox	41	100	88	377	2	7	114%	277%
Listbox	49	125	228	459	5	9	365%	367%
Button	48	81	98	415	2	8	104%	412%

Table 1. Added overhead by RDFa markup for a HTTP GET response for a form request.

Form Element	XHTML		RDF/XML		# triples		Overhead in %	
	min	max	min	max	min	max	min	max
Widget	278	321	525	617	3	6	88.84%	92.21%
Textbox	239	298	454	613	2	7	89.95%	105.7%
Listbox	274	323	656	758	5	9	139.41%	134.67%
Button	245	278	459	720	2	8	87.34%	158.99%

Table 2. Added overhead by RDFa markup in HTTP POST/PUT requests for a form submission/update.

5 Evaluation

Since Web forms/widgets are the de facto data interaction mechanism on the Web and their superiority over direct RDF editing is indisputable, we chose to perform a quantitative evaluation to analyse possible performance issues of our novel data binding mechanism. The performance of the ActiveRaUL framework is influenced by two factors: (i) the overhead introduced by the RDFa annotations, and (ii) the performance of the widget/model generation.

RDFa overhead. The first aspect, i.e., the RDFa overhead, can be investigated by looking into two factors: (i) the time required for upload/download of the generated widgets, and (ii) the efficiency of the client-side JS API.

For an accurate measuring of the overhead, we need to distinguish between the four different HTTP methods (`GET`, `PUT`, `POST`, `DELETE`) and their associated payloads. From the evaluation perspective, only the `GET` response and the `POST`/`PUT` requests are interesting because these operations are responsible for the data transfer. The rest have no payload attached. We have measured the additional data transfer for HTTP requests and responses for each individual annotated form control and compared it to the size of plain XHTML form controls.

Table 1 shows the overhead of the HTTP `GET` requests grouped by RaUL widget components. The first column (XHTML) shows the size of the equivalent XHTML element rendered without annotations (in bytes), while the second column (XHTML+RDFa) shows the size of the XHTML+RDFa element (in bytes). The minimal (min) and maximal (max) values denote the size of the RaUL model required to generate the simplest or the richest (i.e., using all possible properties) widget. The third column shows the number of triples required in the backend and encoded in the resulting widget as annotations. Finally, the last column shows the overhead in percentages.

The evaluation results show that the *Widget* element adds around 2.5 times the overhead to a pure form container in XHTML. Since only one *Widget* is required for every form, the bytes presented in the table are in most cases only added once per page. An annotated *Textbox* takes about twice the size of the pure XHTML form and minimally requires two triples in the form model. Adding all properties (in total 7 RDF statements) to a *Textbox* causes an overhead of about 277%. The *Listbox* form control rendered in RDFa adds more than 3.5 times the size of the pure XHTML form. This is due to the fact that there is at least one *Listitem* associated with a *Listbox* which includes the reference to the value triple. As such, a Listbox

needs at least five statements. Similar to the *Textbox* an annotated *Button* takes about twice the size of the pure XHTML control element and minimally requires two triples in the widget model. Again, adding all properties to a *Button* adds a considerable amount of space (more than four times the pure XHTML element) to the page due to the *Group* class which can be used to associate multiple buttons together (see Section 3.1). However, it also includes 8 triples in the annotation.

Table 2 shows the results of measuring the overhead of a `POST/PUT` request for each RaUL form control. We compare a standard form submission using the `application/x-www-form-urlencoded` MIME type (first column) to a RDF/XML submission (second column) via an Ajax request from the client-side JS API. Due to its stateless behaviour, in the case of a `POST` or `PUT` request, the client-side JS API always sends the entire form, as it is not aware of which elements need to be updated on the server. Since the representation of the payload can vary (as discussed in Sect. 3.1), the time required for the transaction will also vary. Among the three representations (RDF/XML, RDF/JSON and N3), we have evaluated only the RDF/XML representation as it is the most verbose one.

The results show that the serialisation of form elements in RDF/XML causes an overhead of minimal 87% and maximal 159%, depending on the element. The RDF/XML sent to the ActiveRaUL Web service is generated from the RDFa annotations by the client-side JS API. Although the overhead seems to be significant in size when all properties of a form control element are used, for the minimally required annotations the size of the `POST` or `PUT` request is less than double (except in the case of the *Listbox*) to the size of a pure `application/x-www-form-urlencoded` request.

The second factor we have investigated in the RDFa overhead was the efficiency of the ActiveRaUL client-side JS API. For evaluation purposes we have measured the time required to parse the RDFa out of the XHTML, via the Blackbird[10] JavaScript library for profiling. Table 3 lists the average parse time from 10 runs for the form elements and our demo form pages. We can see that it takes less than 20 ms to parse the RDFa out of the XHTML for each type of form control. Evaluation the parsing time of our motivating example documents, the most expensive parsing is the user registration form due to the long listboxes containing items for the birthdate input.

Widget/model generation performance. In order to analyse the performance of the server-side widget/model generation we have measured the time required to generate an increasing number of widgets and widget elements. Fig. 4 shows the behaviour (time in ms. as average over 10 runs) of the service for 1, 10 and 100 generated widgets and widget elements (i.e., *Textbox*, *Button*, *Radiobutton*, *Checkbutton*, *Listbox* and *Listitem*). In the case of the *Listbox*, the number of generated *Listitems* per *Listbox* were 1, 10 and 100. The result shows, as expected, a linear scalability of all widget elements except for two cases: (i) the widget itself, which has a constant behaviour (as it does not have any variable elements in its construction), and (ii) the *Listbox* and *Listitems*, which also have a linear scalability, but with a much higher factor, due to the increased number of triples required in the model. In reality, reaching such numbers is highly improbable because they would produce an unusable user interface. Hence, from the usability perspective, the interesting range of values is between 10 and 30 widget elements. Here, the service performs very well

[10] `http://www.gscottolson.com/blackbirdjs/`

RaUL Element	Parsing (ms)	
	min	max
Widget	16	19
Textbox	15	18
Listbox	16	20
Button	16	19
Demo Page	**Parsing (ms)**	
Add User	465	
Add Product	97	
Buy Product	90	

Table 3. ActiveRaUL client-side JS API performance.

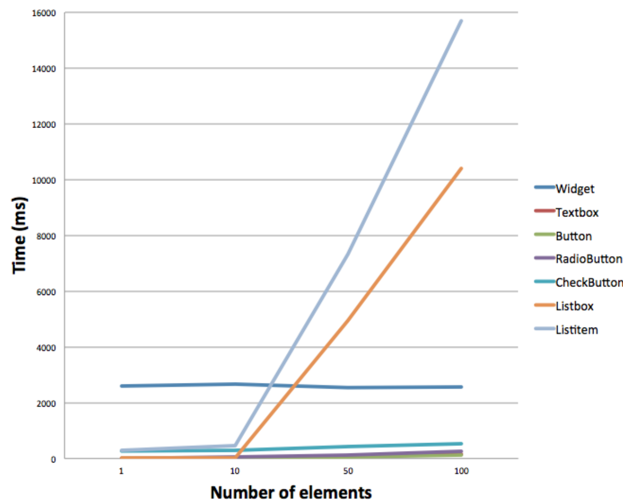

Table 4. ActiveRaULProcessor performance.

(under 0.1 sec. to generate any element), with Listitems being the only element that pose some performance issues.

Discussion. We have shown that the content overhead required for the actual form–data binding is kept to a minimum and is, in principle, a result of using RDFa for the annotation of the semantic forms/widgets. As there are potentially many form controls that could be used in a widget, the user has the trade-off between the depth of the annotations and the size and bandwidth they consume. The more triples are used for a form control the richer its annotations. It also has to be noted that while the ActiveRaUL processor automatically adds RDFa annotations to XHTML pages, the practice of manually adding RDFa or Microformat annotations to XHTML pages is already a common and accepted practice (despite its additional size). The additional size of RDFa annotations, as shown in our benchmarks, is acceptable in most cases. Further, adding the structure of the widget as semantic relations (i.e., RDF statements) yields the following benefits: (a) support for full machine understandable structured form data; (b) structured form data is encoded directly in the Web page and usable to any Semantic Web application; (c) the reuse of existing schemas in the modelling of the form data; (d) the automatic retrieval of form data from the Linked Data Web; and (e) the approach is fully browser agnostic via its rendering in XHTML + RDFa.

6 Related Work

Automatically generating forms from RDF ontologies, with the goal of interacting with Linked Data is a relatively new research topic. We are aware of some earlier attempts concerning form-based editing of RDF data [2], as well as mapping between RDF and forms [16]. None of these approaches propose a generic RESTful Web service to seamlessly combine data binding with the processing and generation of semantic annotations in Web applications.

In [22,4] the authors proposed a read/write-enabled Web of Data through utilising RDForms [13]. It provides a way for a Web browser to communicate structured

updates to a SPARQL endpoint. RDForms consists of an XHTML form, annotated with the RDForms vocabulary in RDFa [1], and an RDForms processor that gleans the triples from the form to create a SPARQL Update [23] statement, which is then sent to a SPARQL endpoint. The difference to our approach is that RDForms does not propose an ontology for form controls and it is bound to a domain-agnostic model – that is, it describes the fields as key/value pairs – requiring a mapping from the domain ontology (FOAF, DC, SIOC, etc.). Dietzold [8] propose a JavaScript library, which provides a way for viewing and editing RDFa semantic content independently from the rest of the application. Further, they propose update and synchronisation methods based on automatic client requests. Their model is restricted to a fixed environment (the Wiki), and they only present the client in-memory modification of the model, but the execution of these atomic add/delete actions as performed in our case by ActiveRaUL is not discussed. Furthermore, there are other approaches such as SWEET [17], which deals with semantic annotations of Web APIs. Fresnel [9] provides a vocabulary to customise the rendering of RDF data in specific browser. At time of writing, there are implementations for five browsers available.

Finally, the FAST gadget ontology (FGO) [11] could provide an alternative to, or could be complemented by the RaUL ontology for modelling widgets and their underlying components. Currently it offers a high-level description of the organisation and information flow of gadgets in respect to screens and resources, the finest granularity mentioned being a `Form element`. In practice the `FGO:Form` and `Form element` can be specialised to the classes introduced by the RaUL ontology, thus providing a more comprehensive model of the domain.

7 Conclusion and Future Work

In this paper we have proposed a novel approach for interacting with Linked Open Data in a read/write manner. The approach uses the RaUL ontology for creating and managing semantic widgets and provides a RESTful Web service, ActiveRaUL, that is used to deploy, generate and retrieve RDFa annotated Web forms. The data – expressed as RDFa triples – is referenced from the RaUL *form model* via a data binding mechanism. The *form model* and *data model* parts make RaUL widgets more tractable and give meaning to the input values in form controls by referencing relations defined in standard Web ontologies. It also eases reuse of forms, since the underlying essential part of a form is no longer irretrievably bound to the page it is used in. We have developed a client-side JS API that parses RaUL annotated XHTML forms and performs a data binding based on the user input or data referenced in the Linked Data Web. The client-side JS API interacts with the ActiveRaUL service, a generic RESTful Web service enabling developers to deploy and manage their Web forms and the data model associated with these forms.

For future work we will focus on extending our approach to allow the modeling of complex page flows including validation and navigation and their automatic rendering in ActiveRaUL. Currently this process is defined in widget specific JavaScript, however we intend to include it explicitly in the RDF model. In addition, we aim to improve the model generation algorithm, and evaluate it against a gold standard defined by ontology experts. Finally, we plan to develop algorithms to determine the

[11] `http://kantenwerk.org/ontology/fast_gadget_content/fgo2011-02-11.html`

type of form field to be used with object properties. Currently, object properties are handled by manually typing their URI in *Textbox* form controls which is arguably not intuitive enough.

References

1. Adida, B., et al.: RDFa in XHTML: Syntax and Processing. W3C Rec. 14 Oct. 2008, W3C Semantic Web Deployment WG (2008)
2. Baker, M.: RDF Forms. http://www.markbaker.ca/2003/05/RDF-Forms/ (2003)
3. Berners-Lee, T.: Linked Data. Design issues for the World Wide Web, W3C (2006), http://www.w3.org/DesignIssues/LinkedData.XHTML
4. Berners-Lee, T., et al.: On Integration Issues of Site-specific APIs into the Web Of Data. Tech. rep., DERI (2009)
5. Berners-Lee, T., Fischetti, M.: Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor. Texere (2000)
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – The Story So Far. International Journal on Semantic Web and Information Systems (IJSWIS) 5(3) (2009)
7. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.91. Namespace document (Nov 2007), http://xmlns.com/foaf/spec/
8. Dietzold, S., Hellmann, S., Peklo, M.: Using JavaScript RDFa Widgets for Model/View Separation inside Read/Write Websites. In: Proceedings of the Scripting and Development for the Semantic Web Workshop (SFSW) (2008)
9. Fresnel, Display Vocabulary for RDF. http://www.w3.org/2005/04/fresnel-info/ (2005)
10. GWT – Google Web Toolkit. http://code.google.com/webtoolkit/ (2010)
11. Haller, A., Rosenberg, F.: A Semantic Web Enabled form model and restful service implementation. In: Proceedings of the Service-Oriented Computing and Applications Conference (SOCA) (2010)
12. Haller, A., Umbrich, J., Hausenblas, M.: RaUL: RDFa User Interface Language – A data processing model for web applications. In: Proceedings of the Web Information System Engineering Conference (WISE) (2010)
13. Hausenblas, M.: RDForms Vocabulary. http://rdfs.org/ns/rdforms/XHTML (2010)
14. Hepp, M.: Web Ontology for e-Commerce. http://purl.org/goodrelations/ (2010)
15. Hobbs, J.R., Pan, F.: Time Ontology in OWL. W3C Working Draft (2006), http://www.w3.org/2006/timezone
16. de hOra, B.: Automated mapping between RDF and forms. http://www.dehora.net/journal/2005/08/automated_mapping_between_rdf_and_forms_part_i.XHTML (2005)
17. Maleshkova, M., Pedrinaci, C., Domingue, J.: Semantic Annotation of Web APIs with SWEET. In: Proceedings of the Scripting and Development for the Semantic Web Workshop (SFSW) (2010)
18. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Tummarello, G.: Sindice.com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies 3(1), 37–52 (2008)
19. Ruby on Rails. http://rubyonrails.org/ (2010)
20. Reenskaug, T.: The original mvc reports. Tech. rep. (February 2007)
21. Apache Struts. http://struts.apache.org/ (2010)
22. Ureche, O., Iqbal, A., Cyganiak, R., Hausenblas, M.: On Integration Issues of Site-Specific APIs into the Web of Data. In: Proceedings of the Semantics for the Rest of Us Workshop (SemRUs). Washington DC, USA (2009)
23. W3C: SPARQL 1.1 Update. http://www.w3.org/TR/sparql11-update/ (2010), Working Draft