WEB SERVICE WRAPPING, DISCOVERY AND CONSUMPTION — MORE POWER TO THE END-USER

Ismael Rivera, Knud Hinnerk Möller
DERI, National University of Ireland, Galway
ismael.rivera@deri.org, knud.moeller@deri.org

Albert Zündorf
University of Kassel, Germany
zuendorf@cs.uni-kassel.de

Keywords: web services, end-users, discovery, consumption, linked data

Abstract: In B2B systems integration and web services, many companies see the advantage of increased operational efficiencies and a reduction of costs. In this scenario, highly qualified software developers are responsible for the integration of services with other systems. However, this model fails when targeting the long tail of enterprise software demand, the end-users. Discovery and consumption of web services are difficult tasks for end-users. This means that potential long tail of end-users creating task-specific applications from existing services is as of yet completely untapped. This paper presents an approach to facilitate the discovery and consumption of business web services by end-users, closing the gap between the two. The approach includes: (a) a catalogue which users can browse to search for web services fitting their needs, and (b) a method to generate ready-to-use web service wrappers to use in the catalogue.

1 Introduction

Business-to-business (B2B) integration is still a significant challenge, often requiring extensive efforts in terms of different aspects and technologies for protocols, architectures or security. While the adoption of open standards such as RosettaNet, ebXML, the Web Service Description Language (WSDL), Universal Description, Discovery and Integration (UDDI) or the Simple Object Access Protocol (SOAP) have reduced the complexity of integrating business applications between different companies and partners, and offered some advantages in business-to-consumer (B2C) integration as well, the interaction and consumption of web services still requires programming skills and a deep understanding of the technology, which poses an obstacle for an end-user with limited knowledge of the matter. Regarding the selection of the right service for the right task — a crucial requirement for the dynamic use of web services — most publishing platforms are syntax-based, making it difficult to navigate through a large number of web services (Pilioura and Tsalgatidou, 2009), preventing end-users from performing these tasks. Solutions such as Semantic Web Services (SWS) promised many advantages in this respect. However, as of yet they have not been widely adopted, possibly because the perceived potential benefits did not justify the additional investments (Shi, 2007).

The motivation of our work has been strongly influenced by the end-users' needs. We are targeting users which are non-skilled in programming and software development, empowering them with a platform to select and consume web services in a straight-forward manner. Rather than publishing services directly in our platform, we leave existing third-party services untouched and instead integrate them through service wrappers. The products resulting from the wrapping process are two artifacts: a specific definition of the web service to be used by this tool, and a ready-to-use piece of code with the proper functions to invoke the web service.

2 Related Work

Web services have been around for a long time. One of the most important claims about their benefits has been (syntactic) interoperability between third-party systems and applications based on different platforms / programming languages. System integration, within a company or between systems from different enterprises, became easier with the adoption of web service standard technologies such as WSDL. While many integrated development environments (IDEs) can deal with WSDL to facilitate the integration task, trained developers are still required for this. As a step forward, there are several tools which facilitate the interaction with data sources and services on the Web. Yahoo! Pipes, Apatar, JackBe Presto and NetVibes, among others provide a set of modules to access different kind of data sources, such as RSS feeds, a given web page (HTML code), Flickr images, databases, and even powerful enterprise systems such as Salesforce CRM or Goldmine CRM. However, none of these solutions facilitate end-users to build their own applications allowing the interaction with web services created by third-party providers. The solutions found are mainly data-oriented (RSS feeds, databases, raw text). Several tools permits some sort of (web) service integration, but are meant to be used within an enterprise level by savvy business users or developers. The solution presented in this paper leverages the possibility of integrating RESTful or SOAP-based web services inside browser-based applications (i.e., widgets or gadgets) by providing a platform to create, publish and select web services wrappers.

In the context of publishing and discovering Web services, service providers have well-known and widely used technologies to accomplish the task of publication, such as the Universal Description, Discovery and Integration (UDDI) (Clement et al., 2004). UDDI serves as a centralised repository of WSDL documents. A similar concept is iServe (Pedrinaci et al., 2010). This platform aims to publish web services as what they called Linked Services — linked data describing services —, storing web service definitions as semantic annotations, so that other semantically aware applications may take advantage of it. However, the platform does not handle the step from definition to consumption of the services.

3 Context: The FAST Platform

The intention of this paper is to demonstrate the advances made regarding web service discovery and consumption through two artifacts developed as part of our research: the *publishing and discovery platform* and the *service wrapper tool*. While these artifacts may be deployed and used separately by third-party applications, they were originally developed to form the backbone of the FAST platform (Hoyer et al., 2009). FAST constitutes a novel approach to application composition from a user-centric perspective. It is aimed at allowing users without previous programming experience to create their own situational applications by visually combining different *building blocks*, such as graphical forms and back-end services, based on their inputs and outputs (or *pre-* and *post-conditions*). The work covered in this paper are the components highlighted in Fig. 1 by a dashed line. Communication within the platform is mostly done via a RESTful API, using JSON as an exchange syntax.

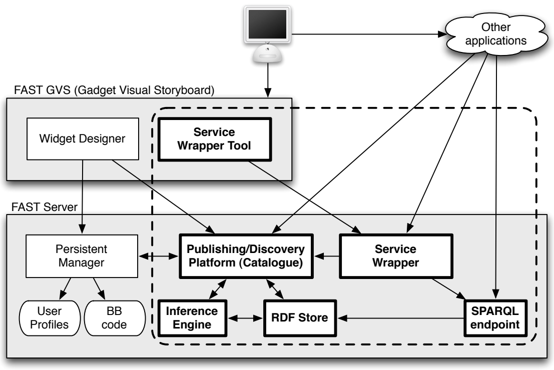


Figure 1: Overview of the FAST architecture

In order to provide a better understanding of these components, we describe a number of concepts related to FAST in this section. A *gadget* is the end product of the platform, ready to be run in any ordinary web browser, usually through a mashup platform. An undeployed gadget is also called *screenflow*, which comprises a set of *screens* connected through their *pre-* and *post-conditions*. A screen is the most complex building block fully functional by itself, visually similar to a tab in a tabbed application. It is composed by a *form* conveying the graphical interface, and a set of *operators* and *back-end services*, wrapped into so-called *service resources*.

The publishing platform, also called *catalogue*, covers several important purposes, such as storage, indexing, publication and search of gadgets, gadget building blocks and user profiles. The service wrapper tool is used to create wrappers for third-party web services, transforming them into building blocks to be stored and reused within the FAST platform.

4 Publishing and Discovery Platform

As explained in previous sections, current solutions and strategies for web service publication and discovery suffer from limited syntax-based descriptions and simple keyword-based search, while other more complex approach failed because the added complexity did not offer sufficient benefits. This paper therefore presents a novel publishing platform (the *catalogue*), permitting any enterprise or individual to publish their public web services, providing enhanced semantic search service wrappers for easy consumption in web applications.

4.1 Overview

One of the main difference of this platform with regards to the state of the art is that it is targeting a different kind of user. As a brief overview, the platform being presented:

- allows functional discovery through web service pre- and post-conditions;
- serves web services wrappers ready to consume in web applications;
- provides advanced search capabilities, based on the formal service definition and inferences extracted from it;
- supports managing its resources via its RESTful API;
- offers a SPARQL endpoint, giving direct access to the data through complex queries;
- offers web service descriptions as linked data;
- follows well-known best practices for publishing data on the web;
- supports content negotiation so that different clients may retrieve information in their preferred format, choosing from JSON, RDF/XML, RDF/N3 or a human-readable HTML version.

The enriched search capabilities are supported by the definition of pre- and post-conditions. They allow to define the inputs and outputs of the services and other building blocks using concepts from any ontology, and in this way to find web services or other building blocks which can be integrated. The concepts of pre-/post-conditions were strongly influenced by WSMO (Roman et al., 2005), simplified and implemented in RDFS for better live performance.

The catalogue architecture comprises an RDF store used for persistence, a business layer dealing with the model and reasoning, and a public facade providing the core functionality as a RESTful API, as well as a SPARQL endpoint accessing the RDF store directly. This presentation layer is aimed to interact with the FAST Gadget Visual Storyboard (see Fig. 1), or any other third-party application.

4.2 Conceptual Model

The conceptual model used to define the web service wrappers within this application has been influenced by both WSDL and semantic approaches such as WSMO and OWL-S. It is part of a more complex conceptual model for FAST (Möller et al., 2010), which can be grouped into three levels: the gadget and screenflow level at the top, the level of individual screens in the middle, and the level of web services at the bottom. All these building blocks and their subparts share the same structure: a set of actions (like operations in WSDL), each of which needs a set of pre-conditions (inputs) to be fulfilled in order to be executed, and provides a set of post-conditions (outputs) to other building blocks. These pre- and post-conditions are defined as RDF graph patterns. E.g., the post-condition of a login service such as "there exists a user" will be expressed as a simple pattern such as "`?user a sioc:User`"[1]. Using this mechanism, extended by RDFS entailment rules, services and other building blocks can be matched automatically. The publishing and discovery platform can employ this functionality to support the internal discovery of web services based on the current user needs (expressed in the same way as pre-conditions).

4.3 Discovery Mechanisms

The main goal of the discovery process is to aid the user in finding suitable building blocks to complement the ones they are already using. E.g., on the screen-flow level, this would mean to suggest additional screens to make existing screens within a screen-flow reachable, and the screen-flow therefore executable. The platform offers two mechanisms to find and recommend screens (or other building blocks) stored in the catalogue: a simple discovery based on pre- and postconditions, and a multi-step discovery or planning algorithm. Before being presented to the user, the results are being ranked, as discussed in Sect. 4.3.3.

4.3.1 Simple Discovery Based on Pre- and Post-Conditions

In this simple approach, the platform will assist the process by recommending all building blocks which will satisfy currently unfulfilled pre-conditions. The pre-conditions of all the unreachable building blocks

[1]We use Trig (`http://www4.wiwiss.fu-berlin.de/bizer/TriG/`) and SPARQL (`http://www.w3.org/TR/rdf-sparql-query/`) notation for RDF graphs throughout this paper.

are collected as a graph pattern, which is then matched against the post-conditions of all available building blocks. In the following scenario, there are two screens: *s1* and *s2*. *s1* has as a pre-condition: *"there exists a search criteria"*, and as a post-condition: *"there exists a item"*. *s2* has just a pre-condition stating: *"there exists a search criteria"*.

```
:G1 { :s1 a fgo:Screen .
      :s1 fgo:hasPrecondition c1 .
      :s1 fgo:hasPostcondition c2 .
      :c1 fgo:hasPattern GC1 .
      :c2 fgo:hasPattern GC2 .
      :s2 a fgo:Screen .
      :s2 fgo:hasPrecondition c3 .
      :c3 fgo:hasPattern GC3 }
:GC1 { _:x a amazon:SearchCriteria }
:GC2 { _:x a amazon:Item }
:GC3 { _:x a amazon:SearchCriteria }
```

The algorithm will construct a SPARQL query to retrieve building blocks satisfying the pre-conditions *c1* and *c3*. The query, although simplified for the sake of clarity, would look something like:

```
SELECT DISTINCT ?bb
WHERE {
  ?bb a fgo:Screen .
  { { ?bb fgo:hasPostCondition ?c .
      ?c fgo:hasPattern ?p .
      GRAPH ?p { ?x a amazon:SearchCriteria }
    } UNION {
      ?bb fgo:hasPostCondition ?c .
      ?c fgo:hasPattern ?p .
      GRAPH ?p { ?x a amazon:Item } } }
  FILTER (?bb != <http://fast.org/screens/S1>)
  FILTER (?bb != <http://fast.org/screens/S2>) }
```

4.3.2 Enhanced Discovery: Search Tree Planning

In artificial intelligence, the term *planning* originally meant to search for a sequence of logical operators or actions that transform an initial state into a desired goal state.

In contrast to the simple approach, the planning approach finds sets of building blocks to fulfil the pre-conditions of a given building block (e.g., a screen). For a certain state, i.e., the initial state which contains the pre-condition to fulfil, a large search tree of possible continuations is considered. Those building blocks cannot satisfy the unsatisfied pre-conditions are discarded, reducing the branches of the tree. A branch stops growing when a building block is reachable (i.e., it has no unfulfilled pre-conditions). Once there are no screens added in a certain step, the algorithm stops and discards all incomplete branches.

It should be pointed out that some of the tree structure is pre-computed to speed up the querying process at runtime. Any time a building block is inserted into the catalogue, the algorithm is executed following two approaches: *forward search* and *backward search*. The forward search approach finds the building blocks whose pre-conditions will be satisfied by the post-conditions of the new building block while the backward search finds the building blocks whose post-conditions will satisfy the pre-conditions of the recently created building block.

4.3.3 Results ranking

This section explains the ranking techniques applied for the different discovery mechanisms.

The ranking algorithm for the simple approach applies the following rules: (1) it gives a higher position to those building blocks which satisfy the highest number of pre-conditions; (2) it prioritises building blocks created by the same user who is querying; (3) it adjusts the rank by using the ratings given to the building blocks, and their popularity in terms of usage statistics; (4) it weights the results according to non-functional features such as availability. This is only applied for what we call "web service wrapper", and it is calculated periodically by invoking the wrapped web services.

For the planning case, the objective is not only to produce a plan but also to satisfy user-specified preferences, or what is known as *preference-based planning*. The ranking algorithm: (1) minimises the size of the plans, after removing the elements of the plan which are already in the canvas, so it gives priority to the elements the user has already inserted, (2) adjust the rank by using the rules 2, 3 and 4 used from the ranking algorithm of simple discovery based on pre- and post-conditions.

4.4 Serving Linked Data

The idea of a Web of data has recently seen a remarkable uptake, e.g. highlighted by large players such as the New York Times, the BBC or an increasing number of national governments (most notably the US and UK governments). We apply this concept in order to provide metadata about the web services as linked data, following the principles as defined in (Bizer et al., 2009), in order to make them available to arbitrary third-party applications. Each web service is identified by an HTTP URI and hosted in the publishing platform so that it can be dereferenced through the same URI. For each building block, data is available in representations in different standard formats such as JSON (for communication with the applications such as the widget designer shown in Fig. 1), RDF/XML, Turtle, or even HTML+RDFa as a human-readable version. To do this, we employ content-negotiation, such as proposed as a best practice in (Sauermann et al., 2008).

5 Wrapping Web Services

Before third-party services can be used by our platform, they need to be provided with a service wrapper. In our approach, this is done in two steps: (i) constructing an exemplary service request, and (ii) analysing the response received from the execution of the service, allowing the mapping of the response data to domain-specific concepts to be used as pre- and post-conditions. From this input, the tool then generates the actual wrapper — JavaScript code to be embedded into web gadgets.

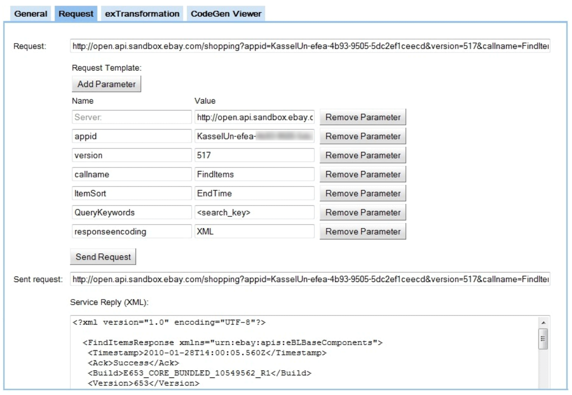
5.1 Constructing Service Requests

We now illustrate the interaction with RESTful services on simple GET requests[2]. In this case, a service request is assembled using its URL and a set of parameters. As an example, we will look at the eBay Shopping web service. To define the desired pre-condition and the type of post-condition of a new service wrapper, the service wrapper tool provides a form (not shown here). For the current example, we assume that the user has defined a pre-condition *search_key*. We are invoking the service to retrieve a list of items corresponding to the search keyword "USB":

```
http://open.api.sandbox.ebay.com/shopping?
  appid=KasselUn-efea-xxx&version=517&
  callname=FindItems&ItemSort=EndTime&
  QueryKeywords=USB&ResponseEncoding=XML
```

The various parameters in the example are defined by the service provider. However, the most relevant one for our example is *QueryKeywords*, which communicates to the services what we are looking for. An example of service request construction is shown in the Fig. 2. In the "Request" input field the user enters an example HTTP request, e.g. the one above. The tool analyses the request and provides a form for editing the request parameters. In our example, the user has connected the service's *QueryKeywords* parameter with the wrapper's *search_key* pre-condition. Requests constructed in this way can then be sent off to the service, whose response is shown in the bottom text area.

[2]Other methods such as POST or SOAP-based services are also supported.



Request: http://open.api.sandbox.ebay.com/shopping?appid=KasselUn-efea-4b93-9505-5dc2ef1ceecd&version=517&callname=FindIte

Request Template:

Name	Value
Server:	http://open.api.sandbox.ebay.c
appid	KasselUn-efea-[redacted]
version	517
callname	FindItems
ItemSort	EndTime
QueryKeywords	<search_key>
responseencoding	XML

Sent request: http://open.api.sandbox.ebay.com/shopping?appid=KasselUn-efea-4b93-9505-5dc2ef1ceecd&version=517&callname=FindIte

Service Reply (XML):

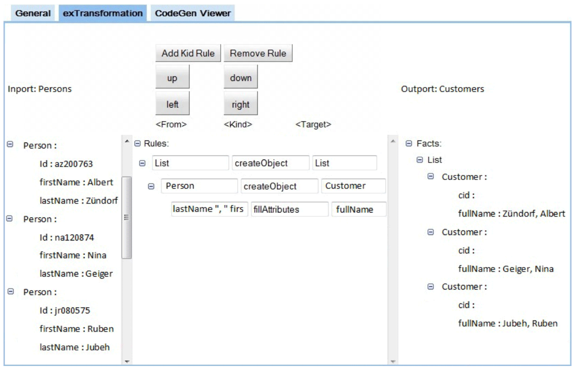
```
<?xml version="1.0" encoding="UTF-8"?>

  <FindItemsResponse xmlns="urn:ebay:apis:eBLBaseComponents">
   <Timestamp>2010-01-28T14:00:05.560Z</Timestamp>
   <Ack>Success</Ack>
   <Build>E653_CORE_BUNDLED_10549562_R1</Build>
   <Version>653</Version>
```

Figure 2: Constructing the service request

5.2 Interpreting Service Responses

Once the service response has been retrieved, the transformation tab of the wrapper tool shows it as an interactive object tree, as seen in Fig. 3.



```
⊟ Person :
    Id : az200763
    firstName : Albert
    lastName : Zündorf
⊟ Person :
    Id : na120874
    firstName : Nina
    lastName : Geiger
⊟ Person :
    Id : jr080575
    firstName : Ruben
    lastName : Jubeh
```

```
⊟ Rules:
  ⊟ List              createObject     List
    ⊟ Person          createObject     Customer
        lastName ", " firs  fillAttributes  fullName
```

```
⊟ Facts:
  ⊟ List
    ⊟ Customer :
        cid :
        fullName : Zündorf, Albert
    ⊟ Customer :
        cid :
        fullName : Geiger, Nina
    ⊟ Customer :
        cid :
        fullName : Jubeh, Ruben
```

Figure 3: Rule-based transformation of service response

A transformation rule is used to analyse the response data and generate the wrapper's post-conditions. Such a rule is composed of three elements, as seen in the middle part of Fig. 3. The *from* field indicates the source elements to be translated by the rule. The second part defines the type of rule (see below). Thirdly, the target of the rule specifies a certain concept or attribute, to be created or filled. Rules can be chained together in a rule tree. A detailed explanation of the different rule types is as follows:

createObject specifies the creation of a new post-condition of the type specified in the third part of the rule. In the example in Fig. 2, the root rule could search for source elements with name *FindItemsResponse* and create a post-condition of type *List* for each such element. The resulting objects are shown in a facts tree in the right of the figure.

fillAttributes only ever appear as sub-rules in the rule tree. They match source elements defined in the first part of the rule. The third part defines the attribute of the super-rule that will be filled (e.g., *fullName* in Fig. 3).

dummy does not create or modify any objects but instead narrows the search space for their sub-rules, by selecting certain elements in the source tree and ignoring others.

Our tool follows an interactive paradigm. Any time a change to a transformation rule is done, the transformation process is triggered and the resulting facts tree is shown. This aids the user to deal with the complexity of the transformation rules, preventing errors or mistakes. In addition, the tool is ontology-driven, meaning that possible post-condition types and their attributes are selected from domain ontologies loaded in the system.

5.3 Generating the Service Wrapper

Once the wrapping of a service has been defined and tested in the tool, we generate an implementation of the desired specification in XML, HTML, and JavaScript, ready to be deployed and executed inside a web gadget.

6 Conclusions and future work

It has been argued that adopting web services standards helps enterprises to increase operational efficiency, reduce costs and strengthen the relations with partners. A range of WS standards help in this regard. However, when dealing with end-users, the process for publication and consumption is not well supported. In a move towards improving this situation, the work presented in this paper empowers the end-user with a platform (the *catalogue*) to easily select services based on functional behaviour (pre-/post-conditions) and other metadata, being able to download a so-called resource adapter allowing the consumption of web services using standard languages to execute within a web browser, and a tool to transform, in an interactive manner, formal definitions of web services into these resource adapters, ready for being published into the catalogue.

The current version of the wrapping tool permits to create resource adapters for RESTful web services. SOAP-based services are also experimentally supported. As potential future work, we consider to include semantically enriched WSDL documents using SAWSDL, and to support other SWS approaches such as WSMO-lite services — however, real-life examples of these are still hard to find.

ACKNOWLEDGEMENTS

This work is supported in part by the European Commission under the first call of its Seventh Framework Program (FAST STREP Project, ICT-216048) and in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Líon-2).

REFERENCES

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS).*

Clement, L., Hately, A., von Riegen, C., and Rogers, T. (2004). UDDI version 3.0.2 specification. `http://uddi.org/pubs/uddi_v3.htm`.

Hoyer, V., Janner, T., Delchev, I., López, J., Ortega, S., Fernández, R., Möller, K. H., Rivera, I., Reyes, M., and Fradinho, M. (2009). The FAST platform: An open and semantically-enriched platform for designing multi-channel and enterprise-class gadgets. In *The 7th International Joint Conference on Service Oriented Computing (ICSOC2009), Stockholm, Sweden.*

Möller, K., Rivera, I., Ureña, M. R., and Palaghita, C. A. (2010). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable 2.2.2, FAST Project (FP7-ICT-2007-1-216048).

Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecký, J., and Domingue, J. (2010). iServe: a linked services publishing platform. In *Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference*.

Pilioura, T. and Tsalgatidou, A. (2009). Unified publication and discovery of semantic web services. *ACM Trans. Web*, 3(3):1–44.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., and Fensel, D. (2005). Web service modeling ontology. *Applied Ontology*, 1(1):77–106.

Sauermann, L., Cyganiak, R., Ayers, D., and Völkel, M. (2008). Cool URIs for the Semantic Web. Interest group note, W3C. `http://www.w3.org/TR/cooluris/` 05/05/2009.

Shi, X. (2007). Semantic web services: An unfulfilled promise. *IT Professional*, 9:42–45.