JAWS: A Javascript API for the Efficient Testing and Integration of Semantic Web Services

David A. Ostrowski

System Analytics and Environmental Science
Research and Advanced Engineering
Ford Motor Company
dostrows@ford.com

Abstract. Semantic Web Services (SWS) hold a lot of potential to the future of the Semantic Web. In this area, a number of tools have been developed to facilitate their definition and deployment. Our goal is to support an efficient means of testing and integration within a browser-based solution. For this purpose we propose JAWS (Javascript, AJAX, Web Service) : A Javascript API to facilitate the testing and integration of SWS. This software decouples the process of SWS integration and development through facilitation of the AJAX/REST paradigm. By leveraging meta-programming and deep integration techniques we support Web 2.0 inspired applications in the context of complete browser-based development.

Keywords: Semantic Web Service, Javascript, AJAX, REST

1 Introduction

General applications of web services support enterprise development as a means to reduce costs and complexity of integration.[1] This is accomplished through machine-independent protocols and utilization of internet-based technologies for communication. These advantages and characteristics provide a strong motivation for integrating this technology with the Semantic Web, most commonly regarded as the next generation of the Web.[2] This work intends to support this goal by leveraging existing toolkits through an API to enable the automatic and semi-automatic utilization of SWS. [3][4][5] Specifically, we are interested in the rapid facilitation of activities linked to SWS including matchmaking, input/output types comparison and analysis of effects. [6][7] Through the combination of Javascript, AJAX and Web Services we intend to satisfy a number of goals:

- **Web 2.0** We are interested in providing a higher level of accessibility to web services – supporting work collaboration and data sharing among second generation web applications.
- **Integration** Given the ubiquitous nature of Javascript, developers can readily incorporate a Javascript API within any web-based frameworks.
- **Deployment (Compatibility)** All software functionality is developed without browser-specific capability**.**
- **Protection** Methods of access to corporate (secure) data stores are controlled through the application of complete Javascript APIs. This approach to controlled data sharing has been popularized by such companies as Amazon and Google .[8][9]

- **Simplicity** Applications can be developed solely in Javascript requiring knowledge of only one language. Complexity of software integration (data server, external software) can be handled separately from a client side developer.

In section Two, we present an overview of our SWS environment including an introduction to the API functionality. Section Three discusses the major constructs of our API. Chapter Four demonstrates its usage in a short example along with a use-case scenario. Chapter Five concludes with a summary including several issues highlighted for future expansion within our API.

2 SWS Environment

The design for our SWS environment relies on two separate knowledge bases.(Figure 1) The first OWL-based KB supports the description of the application domain, defining concepts and terms used for web services description. The second maintains semantic based-definition of web services through the employment of OWL-S [10][11].

Our API supports interaction with these two data sources as well as invocation of the established web services. The first activity supports development of AJAX-based requests to support discovery of SWS within an OWL-based taxonomy. This task ranges from simple keyword matching within a class taxonomy to interaction with server-side tools to provide advanced query capabilities and reasoning. [12][13][14][15] Here, the JAWS API relies upon REST-based services providing efficient support to clients. The second major support step is to utilize AJAX-based requests to identify and retrieve OWL-S files in order to dynamically generate Javascript objects for their representation. By referencing the predefined format of the OWL-S files, users are able to generate applications for the purpose of comparison, integration and testing. The last step is the utilization of the Javascript constructs to invoke our REST-based web services. Here, the services will be defined in an array of Javascript objects referenced by the web services name. Javascript methods as named in OWL-S representations will be used to reference the REST-defined web services.

2.1 Software Layers

The theme of the JAWS architecture is to provide a browser-based environment that separates the activity of SWS testing and maintenance. With this goal the top layer is presented as the application layer in which a complete SWS application is developed either completely in Javascript or in cooperation with another framework. In this layer the user interface is HTML (also supporting 3D markup via embedded object such as in our case study) controlled via Javascript. In the second layer, we have the actual API that exists as a Javascript library. The third layer is considered the mapping layer in which specific OWL and OWL-S data are mapped to Javascript objects and methods on-the-fly. This development relies heavily on the leveraging of XHR alone to reference data stores residing as OWL and OWL-S files as well as integrating with

REST-based web services. OWL-defined data stores are read by adapters as REST web services enabling the client to perform activities related to SWS discovery. Due to minimal size, OWL-S documents are completely referenced by XHR requests. Finally, the web service invocation is presented as Javascript objects and methods allowing the client programmer to access the data. Web service implementations not conforming to the REST approach or outside of the current domain are handled by a process designed to make secondary web service calls to bridge them to our SWS environment.

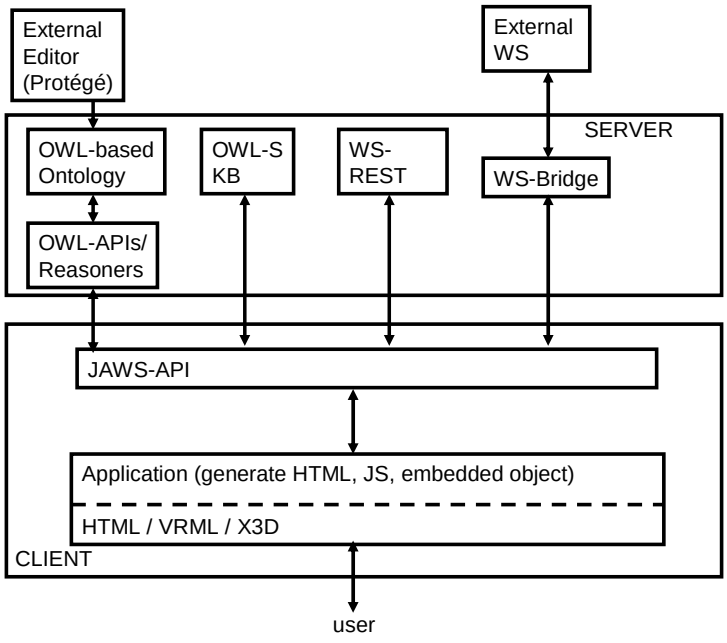

Figure 1 SWS Environment

2.2 Advantages of Javascript

Scripting languages (Ruby, Python, PHP) have gained traction in their application to map RDF-based resources to more programmer-friendly representations. [16][17][18] While reflection is supported in Java and attributes related to dynamic prototyping (interfaces) are supported in languages such as C++, they do not allow for the same level of flexibility in implementation. Maintaining the dynamic run-time capabilities of scripting languages, Javascript maintains similar advantages to Python or Ruby. Recently, with the utilization of the XMLHTTPRequest (XHR) object library, Javascript has demonstrated increased potential by matching capabilities held by traditional server-side scripting languages.[19][20] In utilization of the XHR request model, Javascript can supply a unique approach to the development of semantic web

applications due to its support of asynchronous activity. While any web service implementation can fit into our architecture it is the REST design philosophy that demonstrates highest efficiency. Through the employment of the basic HTTP constructs, the REST approach supports a low-level means of web service implementation. Through avoidance of higher level constructs, REST calls are easily employed within XHR requests thus avoiding extra software for browser-based invocation. Both paradigms together support an approach that brings the highest compromise between efficiency and (browser) compatibility.

2.3 Challenges in Implementation

As noted in earlier software projects mapping RDF based stores, a number of differences between RDF and Object Orientated design have been noted. [21] Among the problems identified include differences between class-based representation, structural inheritance and object conformance. In the first activity, we do not support a complete mapping but present a strictly controlled OWL-based representation generated in Protégé that allows for basic taxonomy definition and categorization with properties of the associate classes pointing to the data stores. Through maintenance of a data store closely conforming to O-O representation, we reduce potential problems with data mapping.

A second challenge is concerning the support of large data stores. In the case of referencing our OWL-based data store, client-heavy implementations can create scenarios involving very large data stores being constrained by memory limitations of the browser. To provide necessary error handling a method is provided to obtain the size of the data store (or subset) imported to the browser.

An additional issue is support of web services residing outside of the domain of the established server as well as applications that are not yet designed as REST-based services. These additional services are integrated through application of a CGI-based process. This implementation has been chosen over higher level security controls including use of automatic proxy generation which can add to complexities in implementation.

3 API

Our API is defined within three major categories in order to support the integration with the OWL data store, direct mapping of the selected OWL-S data stores and eventual invocation of the REST based web services.

3.1 Applied to the OWL Taxonomy

Javascript objects are allocated to support access to OWL-based data. setRes() allows the user to establish an OWL defined resource. This function returns an object providing methods to support the loading and subsequent discovery of SWS resources. The two main input arguments are the adapter and host. The assignment of individual objects provide support for multiple data sources.

```
var currentRes = setRes( adapter, host)
```

With currentRes assigned to a specified resource type, the data store access is loaded and enabled for SWS discovery activities including keyword match, query and reasoning statements.

```
currentRes.getRes(  Resourcename )
```

The size method defines the memory requirements of the data store to provide a means of error handing when loading very large data stores.

```
currentRes.size()
```

The find method provides keyword or basic expressions to be searched through the OWL class hierarchy.

```
currentRes.find( keyword_or_expression )
```

The query method allows for a data point format to be defined (such as SPARQL) and a query string to be applied.

```
currentRes.query( format, queryString )
```

Reasoning statements are performed by defining the format followed by a reasoning expression.

```
currentRes.reasoning( format, expression )
```

3.2 OWL-S Data Stores

Complete mapping of all four OWL-S based class definitions are performed. The function setOWLS accepts a URI for the services description class file and returns a Javascript object representing the data. The object , referenced by hashing in one of the class names (service, profile, process, grounding) allows for access to the defined data types.

var s = setOWLS(“uri")

3.3 Web Service Invocation

Using appropriate information from the OWL-S based definitions, web services are accessed from the WS object array by using the service name provided by an OWL-S description as a hash reference.

```
WS(“SWS_name”).refMethods()
```

Asynchronous callback functionality is provided in the context of the Javascript implementation through the means of the required naming conventions assumed by the wrapper. The naming convention requires that every callback method is defined as the method name followed directly by "_CB" as in example code below. In situations where HTML and 3D markup is shared via embedded object on the browser widows, asynchronous activity can be implemented without callbacks. In this case, efficiencies are provided by the use of return values from the REST WS calls thus eliminating unnecessary overhead.

```
WS("SWS_name").refMethods_CB()
```

3.4 Case Study

We demonstrate use of some of the functionality of the JAWS API through the development of an application to support numerical matchmaking of mathematical based services. This application involves the discovery, dynamic testing and invocation of mathematically based web services. [22][23] For this application we utilize a proprietary algorithm for the find method in which keyword searches are performed against an OWL-defined taxonomy of algorithms. In the first sequence of code we instantiate our object defining an OWL-based algorithm taxonomy.

```
var r =
setres("adapter",http://srl1xpm9q7h41.srl.ford.com)
r.getRes("mathOnto.owl")
r.size()
var arr = r.find("optimization")
```

The next segment of code presents the results of the find allowing for variables to be referenced from the OWL-S data stores to invoke a possible web service

```
for(I = 0;I = arr.length();i++){
      document.write( arr[i].className ,arr[i].childName,
      arr[i].URIproperty)
}
```

When a class is identified of interest, then its properties can be located via a method call (note the usage of 0 as subscript is arbitrary).

```
var s = setOWLs(arr[0].URIproperty)
```

Now call the web service by means of referencing the Javascript object with WS as the established naming convention for the data structure to contain references to the OWL-S defined web services. In our specific example there are two structures passed to the services.

```
WS["className"].optimization(obj_function, var_string)
```

Callback functionality is implemented by a user defining the callback function according to the set naming standard so it can be referenced by the JAWS API.

```
WS["className"].optimization_CB()
```

3.5 Use Case Scenario

A use case scenario demonstrating a subset of the functionality is illustrated in figure 2. In this case we utilize our API in a manufacturing simulation application.[24] This software allows for the design of workstation layouts in a manufacturing (assembly line) setting. The operator paths are dynamically generated via the definition of vehicle and operator velocities along with estimated task times, container location, zone location and associated synchronization activities. In this application we are interested in the incorporation of a web service based means of optimization to generate a suggested optimal design.
In this simulation software we incorporated our API to support a semi-automatic process of SWS discovery and integration. The controls of this portion of our integrated application are implemented via a pop-up window as shown in the lower right portion of figure 2. The first step in this application is to enter a keyword to be applied to the ontology search. This activity will in turn allow for a drop down menu to be populated with possible web service alternatives to be explored. Once a web service is selected, a user can display the input/output definition. Test data scenarios can also be executed to allow the user to examine the input/output requirements in order to trouble-shoot any integration issues.

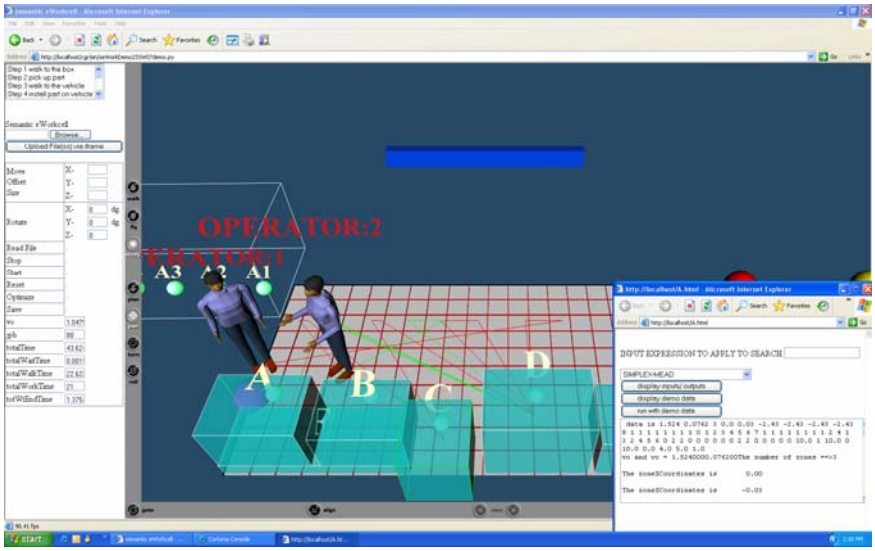

Figure 2 Manufacturing Simulation / Optimization incorporating JAWS API.

4 Conclusion

To support leveraging existing WS technologies, we have presented JAWS, a Javascript API for the purpose of SWS application development. This project breaks down the task of SWS development by creating a means to integrate existing open source implementations, mapping resultant data to Javascript objects thus decoupling integration from the rest of the processes in SWS. A prototype has been implemented allowing for keyword matching and application of SPARQL queries against an OWL ontology, mapping against OWL-S data stores and invocation of REST-based web services. A use-case scenario is presented in which a user is allowed to perform a semi-automatic process of switching in and out web services as necessary. Future work includes addition of the API to support multiple query languages and reasoners thus allowing server functionality to be further configured from a browser-based application.

Acknowledgements

Dr. Anton Eliens, Vrije Univeriteit, Amsterdam, The Netherlands for insight provided in developing this work.

References

1 Deitel, Harvey M., Deitel, Paul J., DuWaldt B., Trees L.K., Web Services: A Technical Introduction, Prentice Hall, 2002
2 Geromenko, Vladimir, Chen , Chaomei, Visualizing the Semantic Web: XML-based Internet and Information Visualization, Springer, 2005
3 J.Scicluna, C.Abela, M.Montebello,*Visual Modelling of OWL-S Services*, submitted at the IADIS International Conference WWW/Internet, Madrid Spain, October 2004
4 http://www.daml.org/services/owl-s/1.1B/owl-s.pdf
5 http://protege.stanford.edu
6. Chaiyakul, Sukasom, Limapichat, Kati, Dixit, Avani, Nantajeewarawat, Ekawit, "A Framework for Semantic Web Service Discovery and Planning, IEEE 2006
7 Srinvasan, Naveen, Paolucci, Massimo, Sycara, Katia, CODE: A Development Environment for OWL-S Web Services, 3rd International Semantic Web Conference ISWC2004
8 http://aws.amazon.com
9 http://code.google.com
10 http://www.daml.org/services/owl-s/1.1B/owl-s.pdf
11 Chase, Nicholas, "The Ultimate Mashup: Web Services and the Semantic Web", http://www-128.ibm.com/developerworks/edu/x-dw-x-ultimashup1.html
12. Zhou, Jiehan, Koivisto, Juha-Pekka, Niemela, Eila, A Survey on Semantic Web Services and a Case Study, IEEE Proceedings of the 10th International Coniference on Computer Supported Cooperative Work in Design, 2006
13. Dodds Leighh, "Introducing SPARQL: Querying the Semantic Web",http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-

14. Carroll, Jeremy J., Reynolds, Dave, Dickinson, Ian, Seaborne, Andy, Dollin, Chris, Wilkinson, Keven, Jena:Implemetning the Semantic Web Recommendations, The 13th International World Wide Web Conference, 2004
15.Evren, Sirin, Bijan,Parsia, Bernardo,Cuenca Grau,Aditya, Kalyanpur and Yarden Katz, Pellet: A Practical OWL-DL resoner, Journal of Web Semantics, 2006
16..Babik, Marian, Hlucky, Ladislav, Deep Integration of Python with the Web Ontology Language, Scripting for the Semantic Web 2006
17 Oren, Eyal, Delbru, Renaud, Gerke, Sebastian, Haller, Armin, Decker, Stefan, "ActiveRDF: Object-Orientated Semantic Web Programming", WWW 2007, May-8-12, Banoff, Alberta Canada
18 D. Vrandecic, Deep Integration of the Scripting language and Semantic Web Technologies, Scripting for the Semantic Web 2005
19. Gross, Christian, "Ajax and REST Recipies, Apress 2006
20. Gehtland Justin, Galbraith Ben, Almaer, Dion, "Pragmatic Ajax: A Web 2.0 Primer", Pragmatic Bookshelf, 2006
21. Oren, Eyal, Delbru, ActiveRDF: object-orientated RDF in Ruby. Scripting for the Semantic Web, 2006
22. The MONET Consortium. MONET Architecture Overview, Technical Report Deliverable DO4, The Monet Consortium, March 2003 Available from http://monet.nag.co.uk
23. The MONET Consortium. The MONET Mathematical Query Ontology. Technical Report Deliverable D13, The MONET Consortium , March 2003, http://monet.nag.co.uk
24. Ostrowski, David A., A lightweight Framework for Web-Based, 3D, Information Visualization, Intnl. Conf. on Enterprise Inf. Systems and Technologies, EIWST 2007