SA-REST and (S)mashups : Adding Semantics to RESTful Services

Jon Lathem
Department of Computer Science
University of Georgia
Athens, GA, USA
lathem@cs.uga.edu

Karthik Gomadam and Amit P. Sheth
kno.e.sis center
Department of Computer Science and Engineering
Wright State University, Dayton, OH
{gomadam-rajagopal.2, amit.sheth}@wright.edu

Abstract

The evolution of the Web 2.0 phenomenon has led to the increased adoption of the RESTful services paradigm. RESTful services often take the form of RSS/Atom feeds and AJAX based light weight services. The XML based messaging paradigm of RESTful services has made it possible to compose various services together. Such compositions of RESTful services is widely referred to as Mashups. In this paper, we outline the limitations in current approaches to creating mashups. We address these limitations by proposing a framework called as SA-REST. SA-REST adds semantics to RESTful services. Our proposed framework builds upon the original ideas in WSDL-S, our W3C submission, which was subsequently adapted for Semantic Annotation of WSDL (SAWSDL), now a W3C proposed recommendation. We demonstrate use of microformats for semantic annotation of RESTful services and then the use of such semantically enabled services with better support for interoperability for creating dynamic mashups called SMashups.

1. Introduction

The evolution of the Web 2.0 phenomenon has led to the increased adoption of the RESTful services paradigm. RESTful services often take the form of RSS/Atom feeds and AJAX based light weight services. The XML based messaging paradigm of RESTful services has made it possible to bring discrete data from services together and create more meaningful datasets. This is being referred to as building a mashup. A mashup is the creation of a new service from two or more existing services. In other words, a mashup can be described as a composition of RESTful services.

Mashups really embrace the idea of a customizable web. A user is most likely not going to want to browse Craigs list and look it the map for each location he finds on the list. On the other hand mashups do not embrace the idea of a read/write web. The reason for this is that it is difficult for the average user with technical training to create a mashup. Typically there is a lot of programming that goes into creating a mashup and the user would need to not only understand how to write code but understand the API of all the services that need to be included in the mashup. This is a time consuming task and impossible for the typical web user. To solve this problem, leading companies are now actively developing tools that can be used to create a mashup and require little to no programming knowledge form the user. These tools typically facilitate a way to select some number of RESTful web services or other web resource and chain them together by piping one services output into the next services input while filtering content and making slight format changes. Three of the leaders in this field are Yahoo! Pipes, Google Mashup Editor, and IBMs QEDWiki.

The drawback of these tools is that they are limited in the number of services that they can interact with. These tools normally deal with services that are internal to the company that the tool was developed from (Google Mashup Editor can use Google Maps) or to services that have standard types of outputs such as RSS or ATOM. This leaves out a vast number of services that cannot be utilized via these tools for the creation of mashups. This is an understandable limitation because it would be extremely difficult to work with a service whose inputs and outputs are in a nonstandard form. This makes it difficult to address issues related to data mediation. It is true that that if one of these companies wanted to add a new service that did not have a standard output or was not an internal service to their tool, it would be possible by making modifications to the existing tooling in order to incorporate the interface of the new service. However, this is not a scalable solution due to the rate at which new services are coming online. The need to change the tool itself also removes the ideal of a customizable web.

We propose to address the limitations in both complexity and scalability, by adding semantics to the descriptions of RESTful services. The proposed SA-REST framework is

a derivative of SAWSDL [11], the W3C PR for adding semantics to WSDL. SA-REST borrows the idea of grounding service descriptions to semantic meta-models using *model-reference* annotations. In this paper we discuss adding annotations using RDFa and GRDDL. SA-REST captures annotations for service inputs, outputs, operations and faults. In addition to these elements, SAREST also captures the type of the request. The SA-REST approach is discussed in detail in section 3.

We illustrate the role of SA-REST in the context of a user-friendly, customizable and scalable approach to creating mashups, called Semantic Mashups (Smashups). The idea behind creating Smashups is to exploit the explicit semantics modeled in SA-REST to demonstrate the use of semantics in the creation and the customization of mashups.

Rest of the paper is organized as follows. In section 2, we motivate the need for our approach by illustrating the limitations of the current mashups. We discuss the SA-REST framework in Section 3. Section 4 discusses creation of mashups using the SA-REST framework. We present our conclusions and future work in Section 5.

2. Limitations in Current Mashups

In this section we outline the limitations in current mashups. We consider the housing maps mashup as an example [5]. The housing maps mashup integrates real estate data from Craigslist.com [1] service and the location information from the Google map [3] web service. Now, it is conceivable that for a given geography, there can be other alternate real estate service providers along with other map service providers. In the current housing maps application, it is not possible for users to change either the real estate service (Craigslist) or the map service (Google maps). Further it is also not possible for an user to to integrate a service that provides information about the traffic in a given locality, in this application. This example illustrates the limitations in the current mashups. The service providers are bound to the application, thereby making it very difficult to switch between different service providers. It is also very difficult to add or remove services from a given mashup. Even though efforts such as Yahoo Pipes [13] and Google Mashup Editor [4], have tried to alleviate the latter problem, they are only partially successful. In these above mentioned applications, the choice of services that a user can add is limited to a set of few services. In this paper, we identify the issues that lead to these limitations and propose a framework to address them.

In the next section, we discuss the SA-REST approach to adding semantics to RESTful services.

3. SA-REST

In this section we present the SA-REST approach to add semantics to RESTful Web services. Adding semantic annotations to a RESTful web service yields many benefits and alleviates many of problems associated with RESTful web services. In this section, we discuss why, what and how of adding semantic annotation to RESTful services. We call this idea Semantic Annotation of RESTful web services or SA-REST.

There have been a number of efforts to add formal semantics to traditional Web services including OWL-S [8], WSMO [12] and WSDL-S [9]. Driven primarily by the W3C member input of our METEOR-S research group at LSDIS lab and Kno.e.sis Center, and in association with IBM [9], a W3C work group has recently released a candidate recommendation called Semantic Annotation of WSDL or SAWSDL. We have developed SA-REST from many of the ideas that were first presented in WSDL-S and then adapted in SAWSDL. The idea behind SAWSDL is to add semantic annotations to the WSDL that describes service. The basic annotations that SAWSDL adds are inputs, outputs, operation, interfaces, and faults. In terms of SAWSDL, semantic annotations are simply bits of xml that are embedded as properties in the WSDL. These properties are URIs of ontology objects. This means that the annotation of a concept in SAWSDL or SA-REST is a way to tie together the concept out of the SAWSDL or SA-REST to a class that exists in ontology. An ontology is a conceptualization of a domain represented in terms of concepts and the relationships between those concepts. Furthermore, it embodies an agreement among its users and provides a common nomenclature to improve interoperability. Since the adoptions of OWL (the Web Ontology Language,[6]) and RDF (Resource Description Framework, [7]) with the associated language of RDFS for RDF schemas as languages for ontology representation and semantic Web data, ontologies are most frequently represented in OWL or RDF. As with SAWSDL, SA-REST does not enforce the choice of a language for representing ontology or a conceptual model, but does allow use of OWL or RDF for representing ontologies or conceptual models. Since SAWSDL and SA-REST are more concerned with the data structure than with the relationships between objects and reasoning, RDF, as the simpler of the two languages, is likely to be used more frequently in the near future. For example, the output message in a WSDL may be annotated with a URI to an ontology class that logically represents that output. Taking the lead of SAWSDL, SA-REST annotates outputs, inputs, operations, and faults, along with the type of request that it needed to invoke the service. The latter is not required in SAWSDL because it deals with traditional SOAP-based services that primarily transmit messages via an HTTP Post because of

the size of the payload, whereas RESTful Web services, being much lighter weight, can use either an HTTP POST request or an HTTP GET request. Later we will see that since SA-REST is in once sense a directive of SAWSDL, we can translate a SAWSDL service into a SA-REST service.

3.1 Data Mediation

The point of adding annotations to the inputs and outputs of a service is to facilitate data mediation. Data mediation is effected in SAWSDL and in SA-REST not only by specifying the ontology concept that its message represents but also by specifying a lifting and lowering schema. What a lifting and lowering schema does is to translate the data structure that represents the input and output to the data structure of the ontology, which we call the grounding schema. This grounding schema is needed is because it would be impossible to get everyone on the Web to agree on a single data structure for a concept. Furthermore, to force a data structure for a concept would hurt the flexibility of the service and the ideology of a read/write web. Lifting and lowering schemas are XSLTs or XQueries that can take an instance of an implementation-level data structure and turn it into an ontology-compliant data structure. By implementation-level data structure we mean the data structure that the service expects in the format that the service expects. This is a scalable data mediation solution because the service provider need provide only one lifting schema, which translates the output of the server to the grounding schema that logically represents the object, and one lowering schema that translates the grounding schema to the input of the service. This solution contrasts to having to provide a translation or mapping from one service to every other service that plans to utilize that service. In terms of an object-oriented style language, the lifting schema should be thought of as an up cast and the lower schema should be considered a down cast, where the ontology class plays the role of the parent object and the service input and output plays the role of child object.

The first significant benefit of adding semantic annotation to RESTful Web services is data mediation. This is a key benefit since RESTful Web services have no way to specify the format of the inputs or outputs of the service. The following example illustrates the problem faced by standard (non-annotated) RESTful services: a user wants to plot the output of a service that tells the address of homes for rent on a map. Logically the real estate service returns a list of locations but the output is represented in XML. Assume that address constitutes an address line 1 field and an address line 2 field. The map service logically takes in a list of locations but the input needs to be represented in JSON and only has one address line. Logically these services should be able to work together with little effort, but they cannot; some work must go mediating the data to be used between these two services. We propose to address this problem by annotating all inputs and outputs of the service with a URI to concepts related to address described in ontology. It is important to note that the annotation of the services inputs and outputs does not change the input or output of the service, nor does it place a restriction on the format. The annotation of a service is simply a way of attaching metadata to the service to be used later to glean information about the service. These metadata act as a loose standardization or normalization for data in the messages of the service.

Now assuming that the output of the real estate service points to a location object in an ontology and that the map service points to the same location object, data mediation could be done as follows: The first schema would not only point to the ontology for the output of the service but would also specify a lifting schema that could translate the output of the service into a form compliant to the ontology. It is important to note that the actual fields of the data structures are not annotated. The high-level concepts that the data structures represent are annotated. Likewise the mapping service will specify a lowering schema that would take data in the form of the ontology item and translate it into the form the service requires for input. In this way, the code for that mashup would invoke the real estate service, lift the output to the ontology form, lower the output the form for the mapping service, and invoke the mapping service. This bit of code would be general for all mashups. The solution is scalable, because for each service that is created the author need supply only a single lifting and lowing schema and not a mapping to all other services that may want to use this service in the future. By contrast the tools currently available require that all data mediation go on within the tool itself. The key role that ontologies play in our approach will be described later. Figure 1 demonstrates the process outlined above.

3.2 Service Invocation

The second problem that adding semantic annotations to RESTful services solves is to automatically determine how services are invoked. Traditional SOAP bases services are invoked via an HTTP Post request. This is because the HTTP Get requests are restricted in size and the amount of data that is present in an SOAP envelope can easily exceed this limit. Since RESTful web services do not suffer for the bloating that SOAP services suffer from, the style of invocation reflect more traditional view of HTTP Get and HTTP Post. Traditionally a HTTP Get request should be used when the request will not cause any changes to the serve where at a HTTP Post request could cause some change on the server. The issue for a tool to invoke a service would

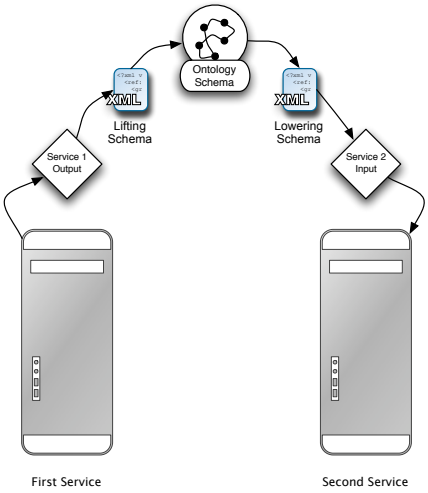

Figure 1. A demonstration of Lifting and Lowering Schemas

be which type of request to choose. This is why in SA-REST we annotate the service with the type of request that should be used. The current tools that are being used to create mashups circumvent this problem by always using a HTTP Get on external services. This is possible because the external services that are allowed are normally feeds such as RSS and the retrieval of a feed does not cause a change on the server.

3.3 Annotation techniques and languages

So far we have only shown what kind of annotations need to be added to make a RESTful web service in to a SA-REST service but have not stated how or where to attach the annotations. In SAWSDL the semantic annotations that describe the service were added in the WSDL for that service. This was a very logical place to add the annotation because typically there is a one to one correlation between a WSDL and a traditional SOAP based web service. Most if RESTful web services to not have WSDLs. This is because one of the main objectives behind REST is simplicity and since WSDLs are complex along with creating another artifact of the service that must be kept up-to-date as the service changes. Also the simplicity of RESTful web services is sufficient to not require a WSDL, whose primary use currently is to facilitate tooling support. Most RESTful web services have HTML pages that describe to the users what the service does and how to invoke the service. This is in a way the equivalent of a WSDL for RESTful web services so would be an ideal place to add our semantic annotations. The problem with treating a HTML as a WSDL is that HTML is meant to be human readable where a WSDL was designed to be machine readable. This is where micro formats come in. Micro formats are a way to add semantic metadata to human readable text is such a way that machines can glean the semantics. Micro formats come in many different competing forms. Recently the W3C has worked on the standardization of two different technologies called GRDDL[2] and RDFa[10]. GRDDL (Gleaning Resource Descriptions from Dialects of Languages) is a way for the author of the human readable text to choose any micro format and also specify a translation, normally an XSLT, that translates the human readable text into machine readable text. RDFa is a way to embed RDF triples in to an XML, HTML, or XHTML document. As a preferred implementation of SA-REST we recommend the use of RDFa as a micro format because it is standardized by the W3C and as it is a subset of RDFa, it has built in support URIs and namespaces. We will first discuss how annotations will be added to an HTML page using RDFa and then we will discuss the more general case of using GRRDL.

We embed our semantic annotations in RDFa into the HTML page that describes the service making the page both a human readable and machine readable description of the service while also creating a single place to do an update if the service ever changes. This in contrast to attaching a separate document that contains the annotations so that when the service is updated the HTML(the human readable document) must be updated and the formal description (the machine readable document) also updated. The RDF triples that can be extracted from the XHTML via parsers or XSLTs. SA-REST leaves it up to the user on how and where to embed the triples– they could be intermingled into the HTML or clustered all together and not rendered by the web browser. The subject of the triple should be the URL at which you would invoke the service; the predicate of the triple should be either sarest:input, sarest:output, sarest:operation, sarest:lifting, sarest:lowering, or sarest:fault where sarest is the alias to SA-REST namespace. The object of the triple should be either a URI or a URL to a resource depending on the predicate of the triple. Figure 2 and 3 give a detailed example of a SA-REST document for a Web service to search for houses on Craigs List.

To allow the author more flexibility and to have a lower barrier of entry we allow the user to use GRDDL to attach annotations. To annotate a HTML page with GRDDL the author fist needs to embed the annotations in any micro format. To the head tag in the HTML document a profile attribute must be added that is the URL of the GRDDL profile. This tells agents that come to this HTML page that it has been annotated using GRDDL. The final step in adding annotations is to inside the head element add a link tag that contains the URL of the translation document. Though any format may be used to add annotations to this page the re-

```
<html xmlns:sarest="http://lsdis.cs.uga.edu/SAREST#">
  …
        <meta about=" http://craigslist.org/search/">
              <meta property="sarest:input"
                    content=
                    "http://lsdis.cs.uga.edu/
                    ont.owl#Location_Query"/>

              <meta property="sarest:output"
                    content=
                    "http://lsdis.cs.uga.edu/ont.owl#Location"/>

              <meta property="sarest:action" content="HTTP GET"/>

              <meta property="sarest:lifting" content=
                    "http://craigslist.org/api/lifting.xsl"/>

              <meta property="sarest:lowering" content=
                    "http://craigslist.org/api/lowering.xsl"/>

              <meta property="sarest:operation" content=
                    "http://lsdis.cs.uga.edu/
                    ont.owl#Location_Search"/>
        </meta>
  …
```

Figure 2. An annotated Web page to search for houses on Craigs List. Annotations not mixed with content

```
<html xmlns:sarest="http://lsdis.cs.uga.edu/SAREST#">
  …
        <p about=" http://craigslist.org/search/">
              The logical input of this service is an
              <span property="sarest:input">
                    http://lsdis.cs.uga.edu/ont.owl#Location_Query
              </span>
              object. The logical output of this service is a list
        of
              <span property="sarest:output">
                    http://lsdis.cs.uga.edu/ont.owl#Location
              </span>
              objects.  This service should be invoked using an
              <span property="sarest:action">
                    HTTP GET
              </span>
              <meta property="sarest:lifting" content=
                    "http://craigslist.org/api/lifting.xsl"/>

              <meta property="sarest:lowering" content=
                    "http://craigslist.org/api/lowering.xsl"/>

              <meta property="sarest:operation" content=
                    "http://lsdis.cs.uga.edu/
                    ont.owl#Location_Search"/>
        </p>
```

Figure 3. An annotated Web page to search for houses on Craigs List. Annotations mixed with content

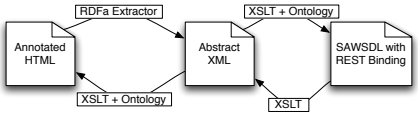

Figure 4. Translating from SA-REST to SAWDL and back

sulting data that is extracted after the translation is applied to the document must result in RDF triples that are identical to the ones that would be generated via RDFa embedding. That is to say a page that is annotated with GRDDL still needs to produce triples whose subject is the URL to which is used to invoke the service, whose predicated is the type of SA-REST annotation that is , and whose object is the URI or URL that is the resource that the predicate refers to.

GRDDL has the advantage that it is less intrusive then RDFa. GRDDL allows the user to embed annotations in any way that is convenient; this could be a pre-existing micro format or a new micro format only known by the user. It would also be possible using GRDDL for the user to embed no extra data in the HTML page which is not required by GRDDL and have all the metadata that is specific to SA-REST be contained in the translation. The advantage of RDFa is that the annotations are self contained in the HTML page. The user only needs to create and maintain one document. In contrast, GRDDL forces the user to create two documents, the HTML page and translation document. RDFa also has the advantage that it is a standardized micro format. The standardization makes it simpler for a developer to maintain and understand a page that has been created by someone else.

3.4 From SA-REST To SAWSDL and Back

Due to the fact that SA-REST is a derivative of SAWSDL it follows that a SAWSDL can be generated from an SA-REST annotated page. This idea would be useful because there are many tools that have been designed for SASWDL. These tools include programs for workflow automation, service discovery, and service publication. All there tools rely heavily on the annotated WSDL. This means that these tools could not cover REST services since they lack a WSDL. Given the above mentioned annotation scheme we can create a SAWSDL from a SA-REST page. The ability to create this mapping between SAWSDL and SA-REST shows that SA-REST has similar semantic support as that of SAWSDL.

The main part of a SA-REST HTML page that is used for computation is the RDF triples. These triples can easily be translated in to a simple XML by the application of an XSLT, the use of an RDFa extractor, or a GRDDL agent.

These RDF triples can be converted into an abstract XML that describes the SA-REST service. The WSDL 2.0 standard defines a REST binding which for purposes of this manuscript is a normal WSDL binding but does not include the SOAP envelope. We can create a simple XSLT that is a template for a SAWSDL that has a WSDL 2.0 REST binding. If this XSLT is applied to the abstract XML document, the input, output, schema mappings, and operations would be filled in to create a complete SAWSDL document. The input and output messages in the SAWSDL describe the data structures and since the abstract XML does not include data structure information, only the concepts about the messages, we must pull this information from the grounding schema of the ontology class that represents the concept.

In a similar way we can move from a SAWSDL to an annotated HTML page for a SA-REST service. We could use a simple XSLT to translate from the SAWSDL to the abstract XML. With this XML we could use another XSLT to translate to HTML. It would be difficult to write a useful HTML page only knowing the RDF triples. For this reason we add more data to the XSLT from the ontology. If the ontology contains comments about the concept, they are added. If the concept is a subclass of a different concept we add this to the XSLT along with any comments it may have.

The newly created SAWSDL contains the generic data structures for input and output messages instead of the data structures that the service expects. This could cause validation errors when the service is invoked. For this reason the client invoking the service must know that data mediation via the lowering schema mapping must be done before invoking the service. As an alternative solution, a SA-REST service could validate the incoming message to see if it is in the service specific form or in format that is compliant to the ontology. If the data is in the generic form the service itself could do the data mediation. This would be a more flexible approach because only the server code would have to be changed instead of changing the code for every client. The reason we propose both solutions is because the user may not always have access to the server code or the client code.

4. Creating (S)Mashups using SA-REST

One of the most popular applications of RESTful web services is mashups, which are basically a web site that aggregates content from different providers. The popularity of mashups has arisen partly due to the simplicity of RESTful web services since the languages used for programming web pages is typically lightweight or scripting languages. A mashup uses RESTful web services to query the providers to get content in results typically in XML format. Due to difference in data definitions and representations of different providers, a semantic approach is needed

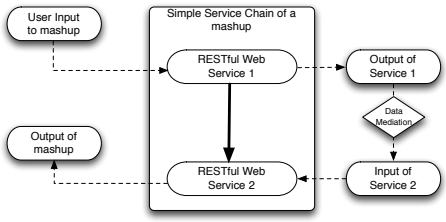

Figure 5. The typical architecture of a mashup that uses two services

for seamless integration of the data. Using semantics to integrate and coordinate mashups gives us SMashups (Semantic Mashups) [18]. As mentioned earlier, a key difficulty in creating mashups is the amount of coding and difficulty in automating its creation. Much of the coding needed to create a mashup is to do mediation between the different data definitions and representations. The key differences between a mashup and a SMashup are the underlying services that are used. A traditional mashup would use RESTful web services where as SMashups use RESTful web services that are annotated with SA-REST annotations. This gives SMashups the ability to know more about what the service is going and what the inputs and outputs are so that data mediation can be done automatically with out human intervention. In this section we describe a system that we have created to aid a user to create mashups without having to know any programming language or doing any programming. In these SMashups, we use semantics to do automatic data mediation.

In Figure 5 we show the typical architecture of a mashup that uses two RESTful web services. The key component of the system that is not the scaleable or flexible is the data mediation. The data mediation is the part of in traditional mashups that is time consuming and requires human intervention. Our system attempts to automate that component of mashup architecture. In our system we allow the user to first specify the chain of services that need to be invoked. That is to say the user specifies the URLs of the annotated HTML pages that describe the SA-REST services that are to be invoked in the SMashup in the order they are to be invoked. The system then sends a request per SA-REST service to the proxy server which in turns downloads the requested annotated webpage and forwards it back to the client. If three SA-REST services were to be used in a particular SMashup then three URLs to annotated HTML pages would have to supplied by the user and three requests to the proxy server would be submitted. The reason for the proxy server is to circumvent the javascript security policy. The javascript security policy states that a webpage can not programmatically download a resource from a domain different from the domain from which the code was

downloaded from. This means that if a user downloads code to create SMashups from one server they cannot download a webpage from a different server. This is why we funnel all requests through the server that we downloaded the smashup code from. This means that the proxy server is only used to host the smashup code and relay messages from the client and that for the creation of a SMashup only one proxy server can be used.

When the annotated HTML page is back on the client, an XSLT is applied to extract the RDF triples. These triples are then used to create a description of the service and display the description to the user. At this time the user can go though and specify where all the inputs should be gathered from. The user can specify that the inputs can be an input to the service or the inputs can be an output of any service that is higher up in the chain. That is to say that if a the first service returns a location object as an output the input to the second service can either be obtained as an input to the smashup or be the location object from the first service. When we say that two objects match we are referring to the objects having being annotated with the same ontology concept or have the service earlier in the chain use a child concept of the object that is later in the chain. That is to say that one object is extended from another. Also at this point the user is allowed to specify which objects are to the outputs of the SMashup.

After all inputs and outputs have been specified the user can prepare the service to be executed. At this step the system takes what objects that are to be used as input to the service and creates an HTML form to prompt the user with. The way the system knows what fields to create is by going to the ontology and finding the attributes of the concept and recursively go through the attributes until you only primitive attributes are left. We are calling this the grounding schema. The name of these primitive attributes is what is presented to the user for inputs to the smashup. It is important to note that these fields may not match up with any of the input fields for a service that will be invoked. After the user has entered all the information the user can start the execution of the service. At this point the HTML fields that were created will be turned into a XML that will closely match what is in the descriptions of objects in the ontology. These XML fragments, there is one for each input, are placed in an intermediate object hash map and saved for later use.

After inputs are entered by the user and execution of the smashup starts, each service in the execution chain will be executed. This means that the engine will look up the inputs that the service needs in the intermediate object hash map. The objects in the hash map are in the form of the grounding schema that reflects that ontology. The lowering schema is then used to convert the data from the grounding schema to the concrete level format this is needed to invoke the service. This format may be XML, JSON, or a query string.

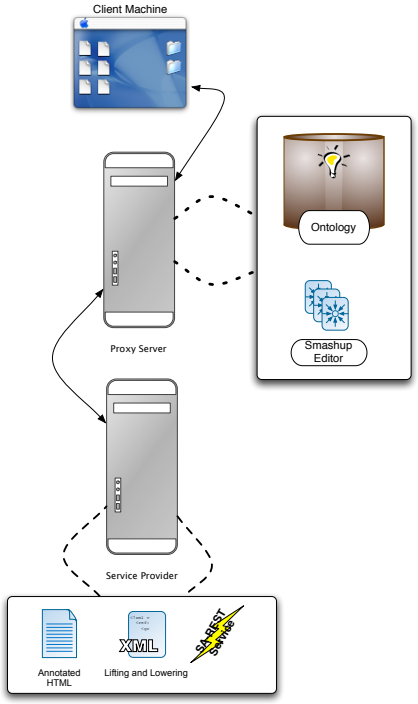

Figure 6. High level preferred architecture of a SMashup

If the service is to be invoked via a HTTP Get request the out put from the translation from grounding schema to implementation schema is appended to the URL for which the service should be invoked at. This string is then passed to the proxy sever who in turn downloads the resource from the URL. The result is then passed back to the client where the lifting schema is applied to convert the output to the grounding schema. This grounding schema XML is then placed in the intermediate objects hash map. Now this process is repeated for the next service in the execution chain.

After all services have been executed the engine loops through all possible outputs of the service and finds all objects that user requested to be returned as outputs and displays them to the user. Note that at this time the output objects that are being displayed to the user are in the grounding schema format and not implementation level format. Possible output formats could be HTML, XML, JSON or even javascript.

5. Conclusions and Future Work

In this paper we have presented an technique to add semantics to RESTful services. SA-REST, presented in the paper adds semantics to RESTful services by adding RDFa annotations. The approach to adding annotations is de-

rived from our earlier work on adding semantics to WSDL (WSDL-S). We also discussed a system we designed to create semantic mashups or SMashups using SA-REST. Our approach seeks to alleviate the two main drawbacks in terms of customizability and scalability of the current approaches to creating mashups.

References

[1] Craigs List. http://www.craigslist.com.

[2] Gleaning Resource Descriptions from Dialects of Languages (GRDDL). http://www.w3.org/tr/grddl/.

[3] Google Maps. http://www.map.google.com.

[4] Google Mashup Editor. http://editor.googlemashups.com/.

[5] Housing Maps. http://www.housingmaps.com.

[6] I.Horrocks, P. Patel Schneider, and F. van Harmelen. From shiq and rdf to owl: The making of a web ontology language. In *Journal of Web Semantics*, 2003.

[7] Ora Lassila and Ralph R. Swick. Resource description framework (rdf) model and syntax specification.

[8] OWL-S. http://www.daml.org/services/owl-s/.

[9] R.Akkiraju, J.Farrell, J.Miller, M.Nagarajan, M.Schmidt, A.Sheth, and K. Verma. Web service semantics – wsdl-s, a w3c member submission, nov. 7, 2005, http://www.w3.org/submission/wsdl-s/.

[10] RDFa. http://www.w3.org/2006/07/swd/rdfa/syntax/.

[11] Semantic Annotations for WSDL working group. Semantic annotations for wsdl and xml schema.

[12] Web Services Modelling Ontology. http://www.wsmo.org.

[13] Yahoo Pipes. http://pipes.yahoo.com.