MACHINE READABLE WEB SERVICE DESCRIPTIONS FOR AUTOMATED DISCOVERY AND COMPOSITION OF RESTFUL WEB SERVICES

Dhananjay Balan*, Abil N George*, Akhil P M*, Deepak Krishanan*, and Dr. Abdul Nizar M †
* Department of Computer Science and Engineering, College of Engineering, Trivandrum
†Professor, Department of Computer Science and Engineering, College of Engineering, Trivandrum

‘

***Abstract*—Restful web services are getting popular today. Most of the service providers are moving to REST based services due to its simplicity and popularity. Even with this much adoption rate, creation of mashups is a himalayan task for an average web developer. Many solutions had been proposed for adding semantics to RESTful web services to automate discovery and composition of services. However they suffer from many problems like, biased towards SOAP based web services,need a separate file for machine parsable descriptions and more. The existing automated mahsup creation service faces scalability issues. The need of the hour is an highly scalable system to reduce as much human intervention as possible for the discovery and composition of the RESTful web services. In this paper, we proposed a microformat like grammar added to the existing service documentation to double them as machine readable. It also uses RDF descriptions at the backend in order to provide strong interlinking among resources, which the end users are not aware of.**

***Keywords*—Semantic Web, intelligent agents, web service, semantic web service, REST, RESTful architecture, service discovery, service composition, Microformats, Poshformats, RDF**

I. Introduction

REST defines a set of architectural principles for designing web services that uses a Resource Oriented approach compared to SOAP based services, which uses RPC semantics. REST is light-weight compared to SOAP which packs a lot of complexity with the implementation of SOAP envelopes and Header. SOAP messages are formatted in XML while we can use XML or JSON with RESTful services for sending and receiving data.

REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. REST uses Uniform Resource Indicators(URIs) to locate resources.Web developers are shifting to REST based services due to its simplicity and wide adoption rate as a standard for creating web services. Most of the companies now provides RESTful services like google,yahoo,amazon,facebook etc.

Mashups are formed by integrating different services. They can combine data or functionality from two or more sources to create new services. Early mashups were developed manually by enthusiastic programmers. More sophisticated approaches are in need as the use of mashuped services are increased in day to day life. This led to the thought of automating mashup creation so that machines can intelligently create mashups with minimal human intervention.

Earlier approaches of automated mashup creation sticks to the use of an external file for service description which only machine can understand. This gives additional overhead to developers for maintaining machine readable descriptions parallel to human readable documentations. The microformat [6] like approach is simple and easy to use,hence provides low entry barriers for web developers. It uses existing service descriptions in HTML to double them as machine readable descriptions by adding annotations to it such that it is not visible in markups.

There exists so many mashup editors which can automate mashup creation like Yahoo pipes [9], Google Mashup Editors(GME), Microsoft Popfly etc.Yahoo pipes is a free online service with a drag and drop interface to aggregate,manipulate and mashup content from around the web.You can create your own custom RSS feeds with Yahoo Pipe that pull in content from a variety of sources and filter it so that you only see the most relevant news stories.However Yahoo Pipes faces many drawbacks. It cannot create a mashup between any pair of services.Also Yahoo Pipes is not scalable since lack processing power. It fails to generate results while processing complex regex from far multiple locations, with hundreds of posts every minutes.

GME is the mashup editor service from Google.Compared to Yahoo Pipes Google Mashup Editor wins in terms of power and flexibility. It allows to use coding in javascript,HTML,CSS etc along with the interactive interface. Microsoft Popfly is the simplest mashup editor which can be used as entry point for web developers. However both of these services were discontinued.

II. Related Works

Many solutions had been proposed for formally describing RESTful web services. These proposals approach the problem from different directions each providing a novel way of addressing the issue at hand. Most of these solutions were member submissions to the W3C but there is hardly any consensus on one global standard.

A. Web Service Description Language (WSDL) 2.0

WSDL 2.0 [1] is an extension of the Web Service Description Language (WSDL)that was used to describe traditional SOAP based services. WSDL 2.0 supports describing RESTful services by mapping the HTTP methods into explicit services available at the given URLs. So every resource will translate into 4 or fewer different services: GET, POST, PUT and DELETE.

The advantage of WSDL 2.0 is that it provides a unified syntax for describing both SOAP based and RESTful services. It also has very expressive descriptions where you can define the specific data type, the cardinality and other advanced parameters for each input type.

However, WSDL 2.0 requires RESTful services to be viewed and described from a different architectural platform: that of traditional RPC-like services. This forceful conversion negates many of the advantages of the RESTful philosophy. In addition, the expressiveness of the format comes at the price of losing the simplicity achieved by moving to the RESTful paradigm. These verbose files are not the easiest to be written by hand and also impose a maintenance headache. Hence, WSDL files are typically generated with the help of some tools. Further, WSDL descriptions are external files based on XML syntax that the developer has to create and maintain separately.

B. Web Application Description Language (WADL)

Web Application Description Language (WADL) [3] is another XML based description language proposed by Sun Microsystems. Unlike WSDL, WADL is created specifically to address the description of web applications, which are usually RESTful services. WADL documents describe resources instead of services but maintain the expressive nature of WSDL. WADL still has some of the concerns associated with WSDL in that they still requires an external XML file to be created and maintained by the developer. It also results in boilerplate code.

C. hRESTS

hRESTS [6] is a Microformat that attempts to fortify the already existing service documentations with semantic annotations in an effort to reuse them as formal resource descriptions. The Microformat uses special purpose class names in the HTML code to annotate different aspects of the associated services. It defines the annotations service, operation, address, method, input and output. A parser can examine the DOM tree of an hRESTS fortified web page to extract the information and produce a formal description.

hRESTS is a format specifically designed for RESTful services and hence avoids a lot of unnecessary complexities associated with other solutions. It also reduces the efforts required from the developer since he no longer needs to maintain a separate description of the service.

One downside with hRESTS is that, despite being specifically designed for REST, it still adheres to an RPC-like service semantics. It is still required to explicitly mention the HTTP methods as the operations involved. Moreover, instead of representing the attributes of a resource, it attempts to represent them as input-output as in traditional services. This results in a lot of unnecessary markup.

D. SA-REST

Similar to hRESTS, SA-REST [2] is also a semantic annotation based technique. Unlike hRESTS, it uses the RDFa syntax to describe services, operations and associated properties. The biggest difference between them is that SA-REST has some built in support for semantic annotations whereas hRESTS provides nothing more than a label for the inputs and outputs. SA-REST uses the concept of lifting and lowering schema mappings to translate the data structures in the inputs and outputs to the data structure of an ontology, the grounding schema, to facilitate data integration. It shares much the same advantages and disadvantages as the hRESTS format. In addition, since SA- REST is strictly based on RDF concepts and notations, the developer needs to be well aware of the full spectrum of concepts in RDF.

E. SEREDASj

SEREDASj [7], a novel description method addresses the problem of resource description from a different perspective. While the other methods resort to the RPC-like semantics of input-operation-output, SEREDASj attempts to describe resources in their native format: as resources with attributes that could be created, retrieved, updated and deleted. This helps to reduce the inherent difference between operation oriented and resource oriented systems. The method also emphasizes on a simple approach that provides a low entry barrier for developers.

One interesting aspect about SEREDASj is that it uses JSON (JavaScript Object Notation)[15] to describe resources. JSON is an easy and popular markup technique used on the web - especially with RESTful service developers. The advantage is that, the target audience is already familiar with the notation used for markup and can reduce friction when it comes to adoption.

SEREDASj, however, addresses the documentation and description in the reverse order. You can create the description in JSON first and then generate the documentation from this. This can increase upgrade effort required for existing services and is not very flexible. It is still possible to embed these JSON descriptions into existing documentation but it floats as a data island, separated from the rest of the HTML page. This, again, causes some duplication of data between the documentation and the description and causes a maintenance hurdle.

III. Service Description

Services are described using special purpose annotations in the HTML code. These annotations are specified in the class attribute of the associated tags. This attribute is usually used for classification of HTML elements and also as selectors for JavaScript access and to style elements via CSS. Using special purpose annotations as class names help us to reuse whatever is already provided by HTML and JavaScript. Following annotations are proposed for describing resources in a RESTful API.

A. General Annotations

1) **hresource**: This is the root annotation that marks the resource description. All other annotations are contained within an element marked with *class="hresource"*. A client parsing a page could treat the presence of this annotation as an indication of the existence of a resource description on the page. Unless all other annotations are encapsulated in an hresource, they will not be parsed.
2) **name** :Annotates the name of the resource. This can be any human readable name and need not have any programming significance.
3) **url/uri**: Annotates the URL at which the resource is accessible.

B. Annotation Of Attributes

1) **attribute**: Annotates an attribute/property of the resource. All attributes of a resource should be annotated with this annotation. Specific characteristics of the attribute could be further specified by more annotations that are used together with the attribute annotation.
2) **required**: Indicates a required attribute. This annotation is always used along with the attribute annotation.
3) **queryable**: Indicates an attribute that may be provided in the HTTP querystring during a GET operation to filter the results. This annotation is always used along with the attribute annotation.
4) **read-only**: Indicates a read-only attribute. A read-only attribute may be retrieved during a GET operation but may not be included in a POST or a PUT. This annotation is always used along with the attribute annotation.write-once: Indicates a write-once attribute that can be specified only during the create operation (POST) but not during update (PUT). This annotation is always used along with the attribute annotation.
5) **guid**: Indicates if an attribute is a globally unique identifier for the resource that could be used across multiple services.
6) **Comment**: Provides a human-readable description of the attribute. This should be descendant of parent of attribute node
7) **hresource-datatype**: Annotates the datatype of the attribute.This should be descendant of parent of attribute node.For permissible types see table: I

Data Type	Description
Integer or Int	32 bit integer
float	floating point number
Int64	64 bit integer.
Range	Boolean or Bool
Date or Time	should specify date formatting
Timestamp	Timestamp of entity

TABLE I. Data types

eg: Range(0.0,1.0) specifies floating point number between 0 and 1 and Range(0,1) specifies integer between 0 and 1.

C. Annotation of Methods

Attribute can be input or output or both. Originally REST resource can be considers as bundle of attributes, each of which may have restriction on their access. But most of existing REST resources is specified as input/output of HTTP Methods (GET,POST,PUT,DELETE).

1) **method**: This is the root annotation that marks the permissible method. It contains following sub-annotations
   - *type*: HTTP request type. GET,POST,PUT,DELETE
   - *input*: (optional) input attribute name
   - *output*: (optional) output attribute name
   - *header*: (optional) attributes that should be passed as header to HTTP request

D. Annotation of Errors

1) **hresource-error**: This is the root annotation that marks the errors.It should contain two sub-annotation for each error:
   - *error-code*:Specify the error code.
   - *comment*: Description of error.

   eg:

```
<li class="hresource-error">
    <code class="error-code">201</
        code>:
    <span class="comment">test failed
        </span>
</li>
```

E. Annotations For Linking Resources

1) *Link to Super class*: When a resource is a subclass of another resource, this link is indicated by the rel attribute hresource-is-a. This implies that wherever the super class is accepted, the subclass is also accepted. For e.g., if a publisher defines a Book resource to provide a search of their catalog, they could annotate the resource to be a subclass of a more generic Book resource.

```
<a rel="-is-a" href="http://
    dublincore.org/book/"> Book </a>
```

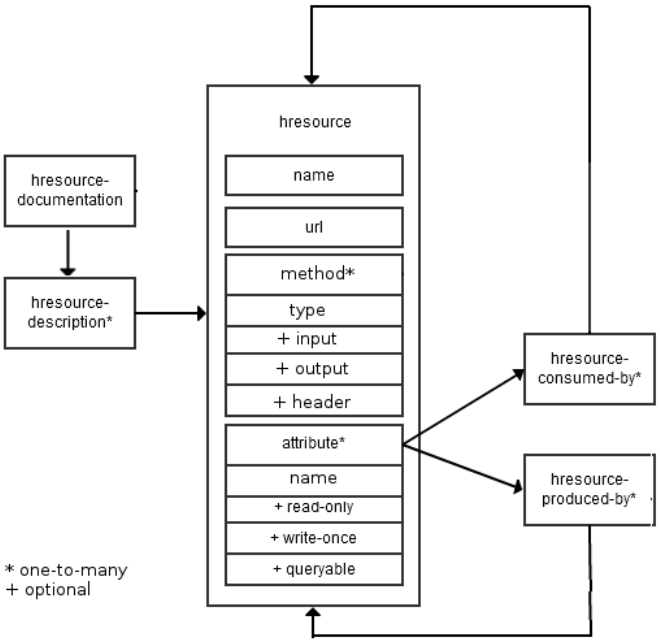

Fig. 1. Relationship between the semantic annotations

If there is another service from a bookshop that is known to accept a generic book resource for a purchase process, the client could infer that the specific book resource from the catalog would also be accepted there and use it.
For this linking to work properly, we need a core set of resources that can be extended by others. Fortunately, there is already a project named Dublin Core running that has defined many commonly used resources. We could reuse these resources for our purpose and use them as the root resources.

2) *Link to Consumers*: When an attribute of a service is consumed by another known service, this is annotated using a `rel` attribute `hresource-consumed-by`. This enables a software agent to find out what all can be done with the resource that it has already retrieved.

```
<code class="attribute">ISBN</code>
Consumers:
<ul>
    <li rel="hresource-consumed-by">
     http://abc.com/buybook\#isbn
    </li>
    <li rel="hresource-consumed-by">
     http://xyz.com/rentbook\#isbn
    </li>
</ul>
```

3) *Link to Producers*: Similar to the link to consumers, services can annotate a link to a producer of one of its attributes. This helps reverse traversal of resources and also makes the system more peer-to-peer. This way, a link needs to be provided in either at one of the consumers or at the provider and an agent can identify this with link traversal. The annotation is made with the rel attribute `hresource-produced-by`.
The relationship between these semantic annotation is shown in figure 1.

IV. Example REST*ful* web service

Consider a RESTful API with a ...
TODO

The Listing.1 depicts an annotation(according to Section.III[Service Description]) across an HTML description of above API.

When above HTML page (Listing.1) is parsed,required information of API is generated as JSON,which is shown in Listing.2.

Moreover we can also generate HTML documentary page from a JSON in specified format.

```
<div class="hresource">
 <h1 class="name">User</h1>
 URL:<span class="url">http://example.com/
    api/user</span>
 <p class="comment">
  It is a <a rel="hresource-is-a" href="
     http://dublincore.org/user/">user</a
     >
 </p>
 <div class="method">
  It use HTTP <span class="type">GET</span
     > request with:<br>
  Inputs:
  <ul>
   <li class="input">username</li>
   <li class="input">password</li>
   <li class="input">country</li>
  </ul>
  Produces:
  <ul>
   <li class="output">token</li>
  </ul>
 </div>
 <h6>Attribute Descriptions</h6>
 <ol>
  <li>
   <code class="attribute required write-
      once queryable">username</code>:
```

```
   <span class="comment">required, must be
       unique.</span>
  </li>
  <li>
   <code class="attribute required">
      password</code>:
   <span class="comment">required.</span>
  </li>
  <li>
   <code class="attribute queryable">
      country</code>:
   <span class="comment">country code</
      span>(<span class="hresource-
      datatype">string</span>)
   <div>Producers:
        <ul>
         <li><a rel="hresource-produced-by
            " href="http://xyz.com/cn#
            country">cn</a></li>
         <li><a rel="hresource-produced-by
            " href="http://abc.com/
            getcountry#country">
            getcountry</a></li>
        </ul>
   </div>
  </li>

  <li>
   <code class="attribute read-only">token
      </code>:
   <span class="comment">userID returned
      as result</span>
  </li>
 </ol>

 <h6>Error Number and Description</h6>
 <ol>
  <li class="hresource-error">
   <code class="error-code">550</code> :-
      <span class="comment">no such user
      </span>
  </li>
  <li class="hresource-error">
   <code class="error-code">504</code> :-
      <span class="comment">Authorization
       failed</span>
  </li>
 </ol>
</div>
```

Listing 1. annotated HTML page

```
{
  "hresource":"1.0",
  "baseurl":"",
  "doc-url":"http://localhost/api_desc/
     user.html",
  "apis":[
    {
      "name":"User",
      "uri":"http://example.com/api/user",
      "is-a":{
      "name":"user",
      "doc-url":"http://dublincore.org/user/
         "
      },
      "description":"It is a user",
      "output-format":"json",
      "methods":[
      {
        "type":"GET",
        "inputs":["username","password","
           country"],
        "outputs":["token"]
      }
      ],
      "attributes":[
      {
        "name":"username",
        "read-only":false,
        "write-once":true,
        "required":true,
        "queryable":true,
        "guid":false,
        "type":null,
        "description":"required,must be
           unique.",
        "consumed-by":[],
        "produced-by":[]
      },
      {
        "name":"password",
        "read-only":false,
        "write-once":false,
        "required":true,
        "queryable":false,
        "guid":false,
        "type":null,
        "description":"required.",
        "consumed-by":[],
        "produced-by":[]
      },
      {
        "name":"country",
        "read-only":false,
        "write-once":false,
        "required":false,
        "queryable":true,
        "guid":false,
        "type":"string",
        "description":"country code",
        "consumed-by":[],
        "produced-by":[
        {
          "attribute":"country",
          "doc-url":"http://xyz.com/cn",
```

```
        "api":"cn"
      },
      {
        "attribute":"country",
        "doc-url":"http://abc.com/
           getcountry",
        "api":"getcountry"
      }
      ]
    },
    {
      "name":"token",
      "read-only":true,
      "write-once":false,
      "required":false,
      "queryable":false,
      "guid":false,
      "type":null,
      "description":"userID returned as
         result",
      "consumed-by":[],
      "produced-by":[]
    }
    ],
    "errors":[
    {
      "code":"550",
      "comment":"no such user"
    },
    {
      "code":"504",
      "comment":"Authorization failed"
    }
    ]
  }
  ]
}
```

Listing 2. JSON description generated from above HTML page

V. System implementation

The proposed system currently addresses the composition of RESTful web services that represent resources using the JavaScript Object Notation (JSON). The system expects the services to return results to API queries as JSON objects and composes them as per the user specification. Extension of the same idea can enable the composition of XML based RESTful services. The system also includes an RDF conversion module that performs the automatic conversion process from Microformat annotations to RDF [8].

The system uses a web UI at the client side for reading user input. The requests are handled by a server program developed in node.js that accepts requests from multiple client machines and handles them asynchronously. The server program acts as a proxy and is developed to enable the system to handle high volumes of client traffic. The server does the bulk of processing and also allows multiple cross domain HTTP calls with ease, which would otherwise be not possible with a client side implementation because of the same origin policy enforced by the modern web browsers. The basic architecture of the system is as illustrated in 2.

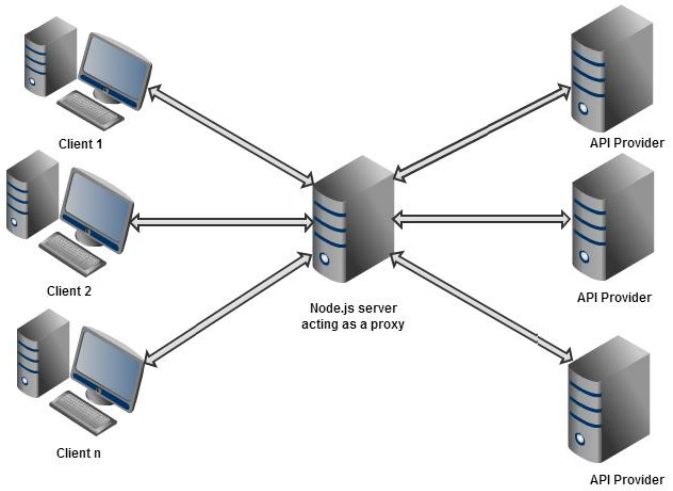

Fig. 2. System architecture

The system uses a parser module to parse the DOM tree of the annotated API Documentation page and extract the information embedded in it. Based on the information extracted from DOM tree, the web UI presented to the user is populated with a set of API operations that the system identifies and that can be composed. A workbench is presented to the user and a drag-drop based user interface enables the him/her to graphically describe the required composition of web services. Once the graphical design of the mash-up is complete and user submits it, information from the design is converted into an abstract internal representation that is passed to the server. The server now invokes the required API calls asynchronously and composes them as required and produces an HTML formatted output which is the required mash-up. The HTML output is then served back to the client system for the user.

The interaction flow in the system is illustrated in 3.

A. Server Side Implementation

The architecture of the server is as illustrated in 4.Server primarily deals with providing three services:

1) *Parsing API Documentation HTML*: Client side provides the URL of an annotated API Documentation HTML page. The server fetches the required documentation page and passes the HTML DOM to the parser module (hrestsparser). The parser generates a JSON description of the annotated resources which is then passed to the client to populate the client UI. The parser uses XPath for the traversal of the DOM tree and hence platform neutral.
2) *RDF Generation*: Client passes a URL to the server. The page is fetched by the RDF generator module. The page is parsed by the hRESTS parser module to

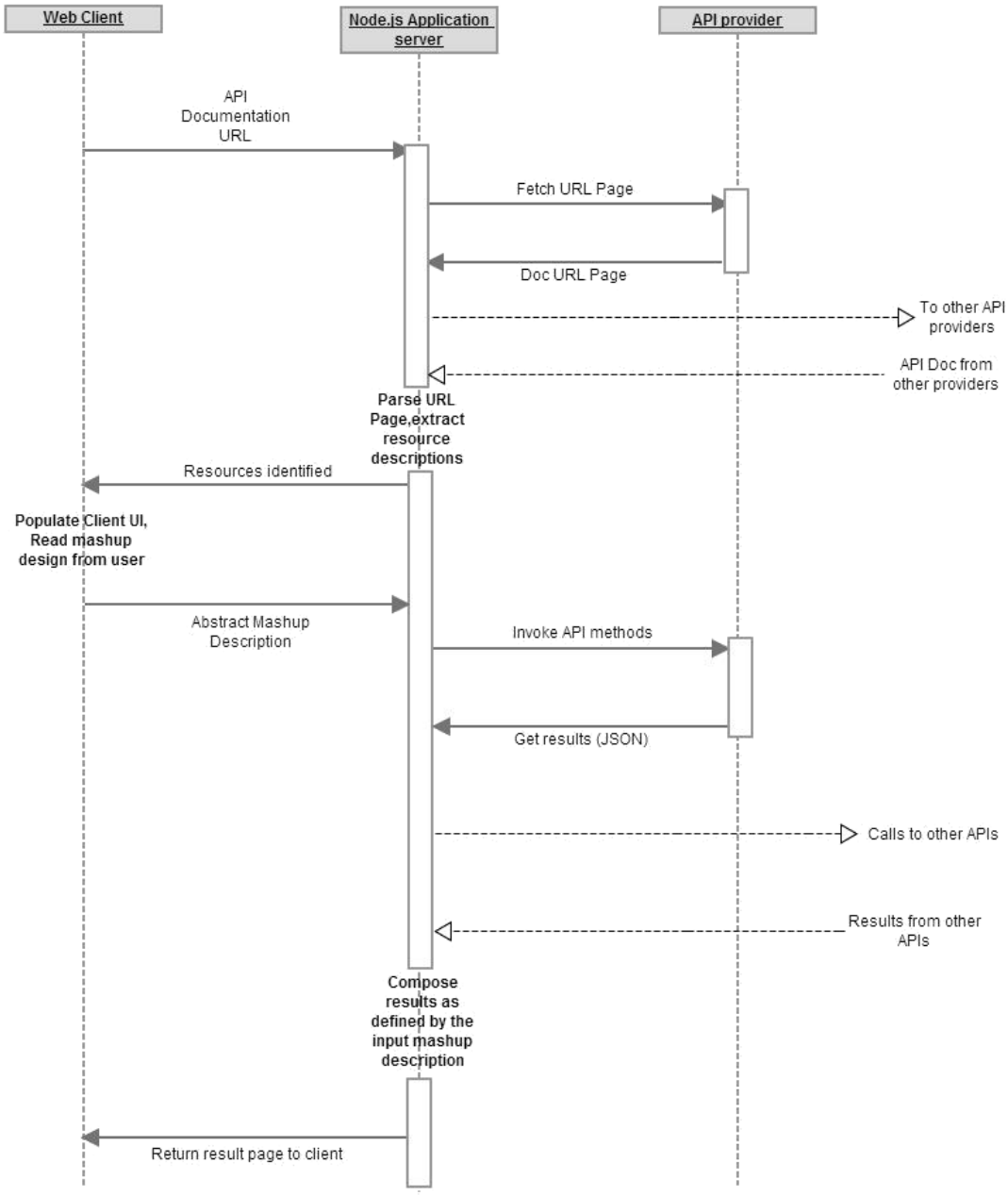

Fig. 3. Interaction flow sequence

produce the output JSON. The produced-by,consumed-by,is-a relations in the annotated page forms a graph of resources each of which is required for the generation of the RDF description. The RDF generator module recursively traverses the graph and fetches each of the required resources up to some arbitrary level of nesting.

3) *Request handling and mash-up generation*: The user specifies the required composition of services graphically in the client workbench. An abstract description of the required mash-up is generated at the client and is passed to the server as a JSON object. The server makes the required API invocations to fetch each of the resources to be composed into the final mash-up. The description JSON is parsed and the result resources are composed as specified producing an HTML output which is then served to the client system.

B. Client Side Implementation

Client provides a simple drag and drop based user interface for mash-up design which can be implemented using jsPlumb

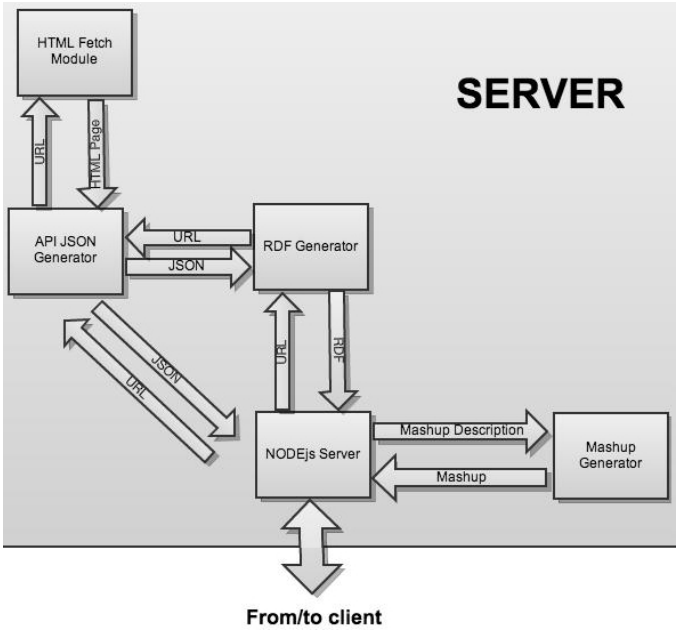

Fig. 4. Server Architecture

[5]. The elements in the UI are populated from the contents of the API documentations passed to the server. The different API calls can be simply dragged and dropped to the workbench and the inputs and outputs can be piped to one another by graphical connectors.

Client uses basic data structures which are simple JSON objects to keep track of attribute value or mapping .This forms a logical graph of the way in which the different service calls are composed. An abstract representation of the mash-up is created by traversing the graph. The traversal can be done by using a modified Depth First Search technique. The abstract representation is a JSON object which represents the initiation sequence for the API calls. This is then passed to the server.

VI. Conclusion and Future work

Web is increasingly getting complex these days. Thousands of services spawn millions of requests each day. This work envisioned a intelligent web framework in which a service can create interconnections between different services compatible. It opens up a multitude of possibilities, including a higher layer manipulation of information for what it represents than how it is represented.

Popularization of services like ifttt [4] have given us the proof that this service is inevitable in the future of web, and if machines were able to parse the information constituted by entire internet it can do wonders that no one envisioned, so it can only be compared with magic.

The prototype mash-up editor created can intelligently mash the services based on annotations But it is also limited in some aspects. But we hope this direction deserves more exploration since it is relatively easy for the developer and the machine to follow. The prototype was implemented in different paradigms to verify the computational comparability. The immediate future directions we would like to purse this works are

- Change to accommodate today's REST APIs. Most of the rest apis in use today does to conform to the RESTful paradigm. They use GET heavily to get most of the things done for performance purposes.
- A tool to convert existing REST APIs. But this comes with lot of challenge. The above one for a start, adoption is another problem.

References

[1] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, 26:19, 2007.

[2] Karthik Gomadam, Ajith Ranabahu, and Amit Sheth. Sa-rest: semantic annotation of web resources. *W3C Member Submission*, 5, 2010.

[3] Marc J Hadley. Web application description language (wadl). 2006.

[4] IFTTT. Ifttt service.

[5] jsPlumb. jsplumb documentation.

[6] Jacek Kopecky, Karthik Gomadam, and Tomas Vitvar. hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE, 2008.

[7] Markus Lanthaler and C Gutl. A semantic description language for restful data services to combat semaphobia. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, pages 47–53. IEEE, 2011.

[8] W3C. W3c's list of documents about rdf.

[9] Yahoo. Yahoo! pipes.