Adding Semantics To RESTful Web Services

Submitted in partial fulfillment for the award of the Degree of Bachelor of Technology in Computer Science and Engineering

Submitted By:

ABIL N GEORGE (09400002)
AKHIL PM (09400005)
DEEPAK KRISHNAN N (09400011)
DHANANJAY BALAN (09400012)

Under the guidance of:

DR. ABDUL NIZAR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
THIRUVANANTHAPURAM 16

May, 2013

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
THIRUVANANTHAPURAM 16

CERTIFICATE

This is to certify that the project report entitled **"Adding Semantics to RESTful Web Services"** is a bonafide record of the project work done by *Dhananjay Balan,Abil N George, Akhil PM, Deepak Krishnan* under my supervision and guidance, in partial fulfillment for the award of the degree of **Bachelor of Technology in Computer Science and Engineering** of the **University of Kerala** during the year **2013**.

Guide

Dr. Abdul Nizar M,
Professor(Guide)
Department Of Computer Science and Engineering
College Of Engineering
Trivandrum-16

Head of the Department

Prof. Ajeena Beegom A.S
Head of Department
Department Of Computer Science and Engineering
College Of Engineering
Trivandrum-16

ACKNOWLEDGMENT

First and foremost, we wish to place on records our ardent and earnest gratitude to our project guide **Dr. Abdul Nizar, Professor, Department of Computer Science and Engineering**. His tutelage and guidance was the leading factor in translating our efforts to fruition. His prudent and perspective vision has shown light on our trail to triumph.

We are extremely happy to mention great word of gratitude to **Prof. Ajeena Beegom A.S, Head of the Department of Computer Science and Engineering** for providing us with all facilities of this work.

Finally yet importantly,we would like to express our gratitude to our project coordinator **Mrs. Deepa S.S**, for her valuable assistance provided during the course of the project.

We would also extend our gratefulness to all the staff members in the Department. We also thank all our friends and well-wishers who greatly helped us in our endeavour. .

Abil N George
Akhil P M
Deepak Krishnan N
Dhananjay Balan

Abstract

Development of a semantic web is gaining a lot of traction recently. At the same time, another change is also getting a lot popular on the web - a move from complex SOAP based web services to the simpler RESTful services that work over the existing HTTP infrastructure. RESTful services were born out of an attempt to unify and simplify the creation and consumption of web based services, and to increase its adoption. Adding semantics to web service descriptions is an important step towards a better web that is accessible to both humans and machines alike. With the web taking a definite turn towards RESTful web services from the traditional RPC-like SOAP based services, it is important to devise a semantic description method for such services. However, for these solutions to be really adopted by developers, they should adhere to the RESTful philosophy and provide a low entry barrier. The proposed solution aligns with the REST philosophy and architecture and reuses existing service documentations to double them as machine-readable descriptions. The Microformats-like syntax used by the solution is simple and easy to write and maintain from a developer perspective. Further, the language describes services as resources with attributes instead of using an RPC-like input-operation-output concept used by most of the current solutions. This markup syntax is then extended to add interlinking between RESTful services, enabling automatic discovery and composition of services. The solution should further reduce the entry barrier for developers and thus increase adoption, resulting in a widely accepted standard. The existence of such a standard technique can make the wealth of resources available on the Internet accessible to machines, enabling the creation of intelligent software agents that can get a lot of work done with no to minimal intervention from humans.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Semantic web is a collective movement towards adding semantics or meaning to the data available on the internet thus making them machine readable. Data with semantics will be key in the future web where human interaction can be reduced for exploring and using this information. While the vision of a Semantic Web has been around for more than fifteen years it still has a long way to go before mainstream adoption will be achieved. One of the reasons for that is, the fear of average Web developers to use Semantic Web technologies. The reasons for this are manifold and should be further researched. Some developers are overwhelmed by the (perceived) complexity or think they have to be AI experts to make use of the Semantic Web and simply shut down when they hear the word Ontology. Others are still waiting for a killer application making it a classical chicken- and-egg problem. A common perception is that the Semantic Web is a disruptive technology which makes it a show-stopper for enterprises needing to evolve their systems and build upon existing infrastructure investments. Obviously some developers are also just reluctant to use new technologies.

There have been two main initiatives towards adding semantics to the web: Linked Data and semantic annotations.

Linked Data is an effort to create a web of data, parallel to the web of documents - the web we know and use today[1]. Since HTML, the language of the web was deemed insufficient to accurately and expressively describe strongly typed relation-

ships between data, a new XML based language named RDF (Resource Description Framework) was suggested[2]. RDF documents define relationships in the form of subject-predicate-object triplets and can thus model a wide variety of links. RDF documents are supported by schemas written in either RDFS[3]or OWL[4][5].

While Linked Data is very expressive when it comes to describing relationships, it has a downside as well: these descriptions are separate from the current web that humans use and leaves developers with another artifact to develop and maintain. This has lead into development of techniques that try to integrate the human web and the machine-readable web into one single entity. Two important standards developed in the area of semantic annotations are Microformats[6],RDFa[7][8] and SEREDASj[14].

1.1 Semantic Web Services

The concept of semantic web has also been applied to web services, using semantic web techniques to

- Describe web services themselves and
- Add meaning to the results provided by web services to make them machine readable and to increase the utility of the information gathered.

Most of the research in semantic web services has been around SOAP based services. These are the ”conventional” web services that have been the prime solution until a few years back. While SOAP packs a lot of power into accurately describing various aspects of a service, such verbosity is often undesired and leaves developers with a high entry-barrier that most chose not to take.

The recent attention gained by RESTful services[9] an entirely different service architecture - is a natural response to the prohibitive complexity of developing and

describing SOAP based services. The REST philosophy prioritizes simplicity over verbosity and works over the existing HTTP infrastructure. Resources are represented as URLS and CRUD operations are defined by the POST, GET, PUT and DELTE HTTP methods respectively. Its resemblance to the way the web works has resulted in widespread adoption since the days of Web 2.0.

The major problem of RESTful services is that no agreed machine-readable description format exists to document them. All the required information of how to invoke them and how to interpret the various resource representations is communicated out-of-band by human-readable documentations. This usually does not cause any problems in the human Web since humans inherently understand the representations and are thus able to quickly adapt to new control flows (e.g: a change in the order sequence or a new login page to access the service). Machines on the other hand have huge problems to understand such representations; just as disabled users sometimes have. A blind user for instance cannot make any use of information contained in graphics. Machines suffer even more from such usability and accessibility problems.

Currently machine-to-machine communication is often based on static knowledge and tight coupling to resolve those issues. The challenge is thus to bring some of the human Webs adaptivity to the Web of machines to allow the building of loosely coupled, reliable, and scalable systems. After all, a Web service can be seen as a special Web page meant to be consumed by an autonomous program as opposed to a human being. The Web already supports machine-to-machine communication, whats not machine-processable about the current Web is not the protocol (HTTP), it is the content.

Some research has gone into utilizing semantic web techniques specifically for RESTful services. These solutions suffer from either of the two problems.

- They are generalizations of existing mechanisms to describe SOAP services. REST being an entirely different architecture, such an approach results in a lot of unwanted markup and code. This conversion to another architecture steals some of the implicit properties of RESTful architecture like simplicity, focus on resources rather than services etc.
- These solutions require an external description, which has to be created and maintained by the developer adding to the effort required.

Further, the solutions for describing RESTful services do not take into account the possibilities of automatic discovery and composition of services. Papers had been published on new proposals that work together with one of these markup languages to provide description. However, these solutions usually suffer from the same issues as the markup languages - they are mainly aimed at SOAP based services and do not fit well into the REST architecture.

Chapter 2

SCOPE OF THE PROJECT

The main feature of this new approach is the automation of mashup creation. It enables automatic discovery and composition of services.The existence of such a standard technique can make the wealth of resources available on the Internet accessible to machines, enabling the creation of intelligent software agents that can get a lot of work done with no to minimal intervention from humans.

The process of creating mashups is a manual task which require a lot of human interaction with the existing standards. There exists two popular mashup creators ;Yahoo Pipes[21] and IFTTT[20].

Yahoo pipes is a free online service to aggregate,manipulate and mashup content from around the web. Modules can be dragged onto its interface and link between them by connecting output of one service as input to other services. It is developed to assist non-technical users to create mashups. 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. It requires no plugins or coding.

However Yahoo Pipes faces many drawbacks. It cannot create a mashup between any pair of services. It works well with a defined set of web services like RSS feeds etc. Pipes is being very buggy lately and has stopped saving new pipes these days. Also some times it is very slow in working. A typical operation with Yahoo Pipes takes atleast 200 ms. Fetching an RSS feed with a filter will normally take 400 ms in Yahoo Pipe. Also Yahoo Pipes lack processing power and cannot

works well with website from far east. It fails to generate results while processing complex regex from far multiple locations, with hundreds of posts every minutes.

IFTTT is a service that lets you create powerful connections with one simple statement: *if this then that* .The this part of a recipe is a trigger. The that part of a recipe is an action. The combination of a trigger and an action from a customer's active channels are called recipes. Pieces of data from trigger a are called Ingredients. The service offers triggers and actions for 61 channels such as Twitter,Foursquare,Flickr and Box etc.

There are two types of recipes, Personal recipes and shared recipes. Personal Recipes are a combination of a Trigger and an Action from your active Channels.Personal Recipes can be turned on and off. When turned back on, they pick up as if you had just created them. Personal recipes check for new trigger data every 15 minutes. Shared recipes are useful templates shared by the IFTTT community. By combining IFTTT with other services such as Yahoo Pipes one can build elaborate systems that enable easier consumption of content from a variety of sources.

The mashups in IFTTT are manually coded by contributers. IFTTT has lots of mashups publicly shared in its store. But the process of creating mashups is still not automated with IFTTT. All we can do with it is to reuse the shared mashups. Since IFTTT only scans Triggers every 15 minutes, so there is a delay between the triggers and actions. The main cited drawbacks of IFTTT are

- **Ifttt cant make compound decisions**. You can setup a a task to send you an email when tomorrows weather is below 40 degrees. You can also set up a task to email when the weather calls for rain. But if you want to get an email when BOTH of these happen, that cant be done at this writing.

- **Ifttt has no memory**. It cant save any information during a task. For example, if you'd like a task that keeps track of all of the people who followed you

on Twitter in the past week, and then includes them all in a thank you tweet at the end of the week. Right now, the best you can do is to instantly reply to any new follower.

- **Ifttt does not play nice with others**. The system works well when reacting to real human events and triggers, but not so well when interacting with other automated services.

Also there is another tool from google to create mashups called Google Mashup Editor(GME).The Google Mashup Editor is an incredibly powerful tool for rapid testing and deployment of mashup concepts, particularly those that utilize Google services or products. This opens the space to all those developers who don't have their own servers to play on and gives them a framework to kickstart development. Compared to Yahoo Pipes Google Mashup Editor wins in terms of power and flexibility. But GME has been discontinued it service(It is migrated to Google App Engine).

So by considering all the drawbacks of the existing approaches we can clearly specify the requirements of a new design. The new design automates mashup creation and user interaction is very less with this approach. The user interaction is needed only during automatic client generation to give inputs for invoking a particular service and to specify the output needed, which in any case cannot be avoided. Also the scalability and performance issues of Yahoo Pipes are addressed in the new design by using a Node.js server in the user system which also finds a solution to make cross domain calls. If a large amount of data is needed to be processed then, the Node.js server package can be moved to a high performance machine independent of the client side code.

Chapter 3

RELATED WORK

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.

3.1 Web Service Description Language (WSDL) 2.0

WSDL 2.0[10] 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.

3.2 Web Application Description Language (WADL)

Web Application Description Language (WADL)[11] 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.

3.3 hRESTS

hRESTS[12] 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.

3.4 SA-REST

Similar to hRESTS, SA-REST[13] 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.

3.5 SEREDASj

SEREDASj[14], 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.

Chapter 4

ILLUSTRATING RESTful WEB

Web APIs and RESTful Web services are hypermedia applications consisting of interlinked resources, oriented towards machine consumption. The orientation towards machine consumption manifests mainly in that the interactions between RESTful services and their clients are generally done with structured data (e.g. XML, JSON), as opposed to the standard Web document markup language, HTML, which is a human-oriented presentation language. In their structure and behavior, RESTful Web services are very much like common Web sites.

Figure: 4.1 illustrates an example RESTful hotel booking service, with its resources and the links among them. The service description is a resource with a stable address and information about the other resources that make up the service. It serves as the initial entry point for client interaction.

The service description resource contains a form for searching for available hotels, given the number of guests, the start and end dates and the location. The search form serves as a link to search results resources, one per every unique combination of the input data the form prescribes how to create a URI that contains the input data; the URI then identifies a resource with the search results. As there is a large number of possible search queries, there is also a large number of results resources, and the client does not need to know that all these resources are likely handled by a single program on the server.

The search results are modeled as separate resources (as opposed to, for in-

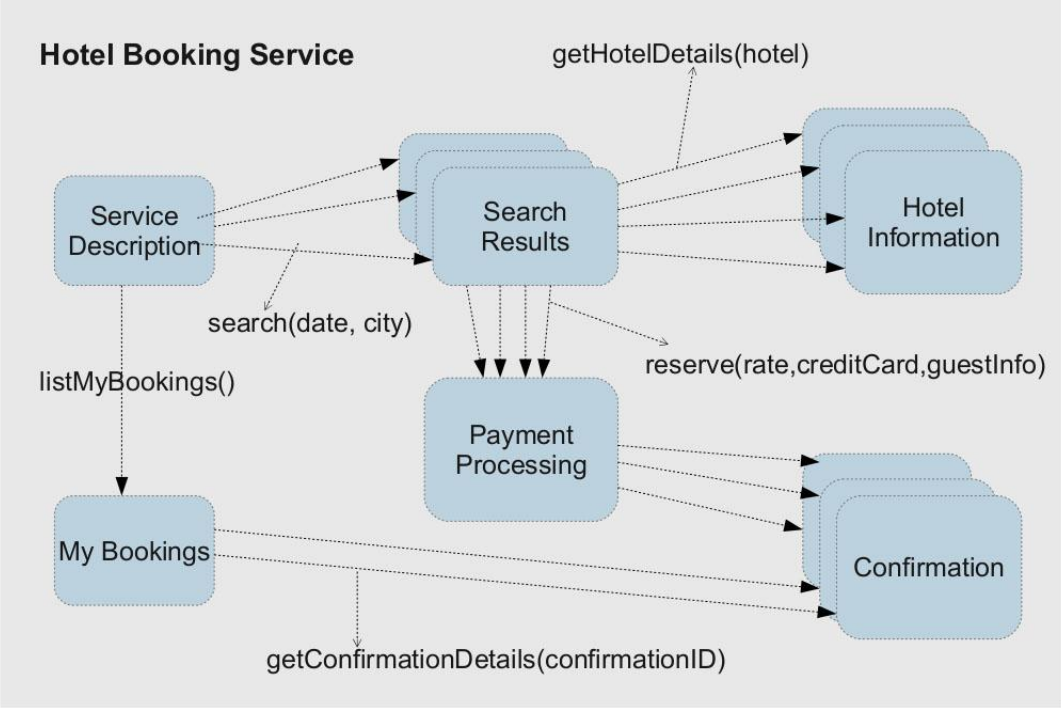

Figure 4.1: Structure of an Example Service

stance, a single data-handling resource that takes the inputs in an input message), because it simplifies the reuse of the hotel search functionality in other services or in mashups (lightweight compositions of Web applications), and it also enables caching of the results. With individual search results resources, creating the appropriate URI and retrieving the results (with HTTP GET) is easier in most programming frameworks than POSTing the input data in a structured data format to one Web resource, which would then reply with the search results.

The service description also contains a link to a page with the bookings of the current user (which requires authentication functionality). With such a resource available to them, client applications no longer need to store the information about performed bookings locally.

Search results are a list of concrete rates available at the hotels in the given location, for the given dates and the number of guests. Each item of the list contains a link to further information about the hotel (e.g. the precise location, star rating and other descriptions), and a form for booking the rate, which takes as input the payment details (e.g. credit card information) and an identification of the guest who is going to stay in the room. The booking data is submitted (POSTed) to a payment resource, which processes the booking and redirects the client to a confirmation resource. The content of the confirmation can serve as a receipt.

Finally, the my bookings resource links to the confirmations of the bookings done by the authenticated user. The confirmations may further provide a way of canceling the reservation (not shown in the picture).

Together, all these resources form the hotel booking service. However, the involved Web technologies actually work on the level of resources, so service is a virtual term here and the figure shows the service in a dashed box.

So far, the description of the example hotel reservation service has focused on the hypermedia aspect: we described the resources and how they link to each other.

Alternatively, and in fact more commonly, we can also view the service as a set of operations available to the clients as an API. The search form in the homepage represents a search operation, the hotel information pages linked from the search results can be viewed as an operation for retrieving hotel details, the reservation form for any particular available rate becomes a reservation operation, and so on.

While the resources of a service (the nouns) form the hypermedia graph (shown in Fig. 3.1), a programmer making a mashup or an automated client program rather thinks of the operations that can be invoked; therefore public RESTful Web services are generally called APIs and are described in terms of the operations. The following might be a typical operation description:

The operation `getHotelDetails()` is invoked using the method GET at `http://example.com/h/{id}`, with the ID of the particular hotel replacing the parameter id. It returns the hotel details in an ex:hotelInformation document.

Chapter 5

DESIGN

This new solution uses a combination of Microformats-style markup and RDFS to provide a comprehensive framework for describing, discovering and composing RESTful services by adding semantics. Microformats, being simple and reusing a lot of the properties of HTML, provides users with a low entry-barrier for developers, which can increase adoption rate. These annotations are not visible to user but hidden in the HTML source and hence do not come in the way of users browsing the site.

In order to enable strong interlinking between services, a more robust solution like RDF and a backing RDF Schema is needed. This is achieved by providing an adapter for automatic conversion from the Microformat to RDF and providing a ready-made RDF Schema for the purpose(Figure: 5.1). This way, the developers need not be concerned with the RDF descriptions that work in the background.

The solution currently addresses RESTful services that represent resources using JavaScript Object Notation (JSON). It is possible to extend the idea to XML based services as well by making some assumptions about the XML serialization.

5.1 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

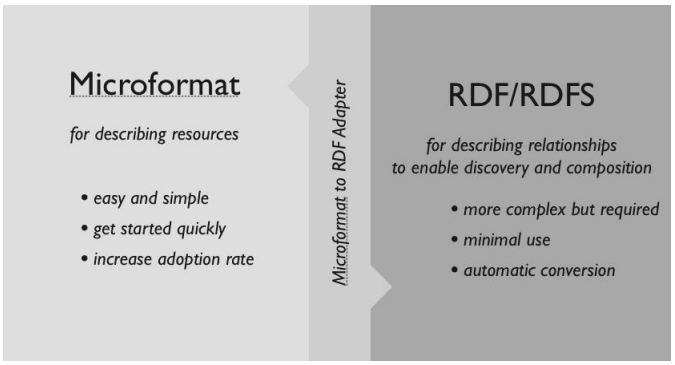

Figure 5.1: Overview of the new architecture

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.

An intelligent agent extracts information about the service by traversing the DOM tree of the HTML page. APIs for traversing trees are available in almost all programming languages including JavaScript which is used extensively on the web and also to make extensions for many popular browsers such Chrome and Safari. Browsers or browser extensions could then could parse this data and automate the creation of clients for these services.

5.2 Annotations

Services are described using special purpose annotations in the HTML code. These annotations are specified in the class attribute of the associated tags.

5.2.1 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.

5.2.2 Annotation of Methods

Annotates Permissiable methods (GET,POST,PUT,DELETE)

1. **method**: This is the root annotation that marks the permissiable method. It contains following sub-annotations

   - *type*: HTTP request type. GET,POST,PUT,DELETE
   - *input*: (optional) input attribute
   - *output*: (optional) output attribute
   - *header*: (optional) attributes that should be passed as header to HTTP request

5.2.3 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: 5.1

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

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 5.1: Data types

5.2.4 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>
```

The relationship between these sematnic annotataion is shown in figuere 5.2 (hresource-produced-by and hresource-consumed-by are described in section 5.3)

An intelligent agent extracts information about the service by traversing the DOM tree of the HTML page(See figure 5.3. APIs for traversing trees are available in almost all programming languages including JavaScript which is used extensively on the web and also to make extensions for many popular browsers such Chrome and Safari. Browsers or browser extensions could then could parse this data and automate the creation of clients for these services.

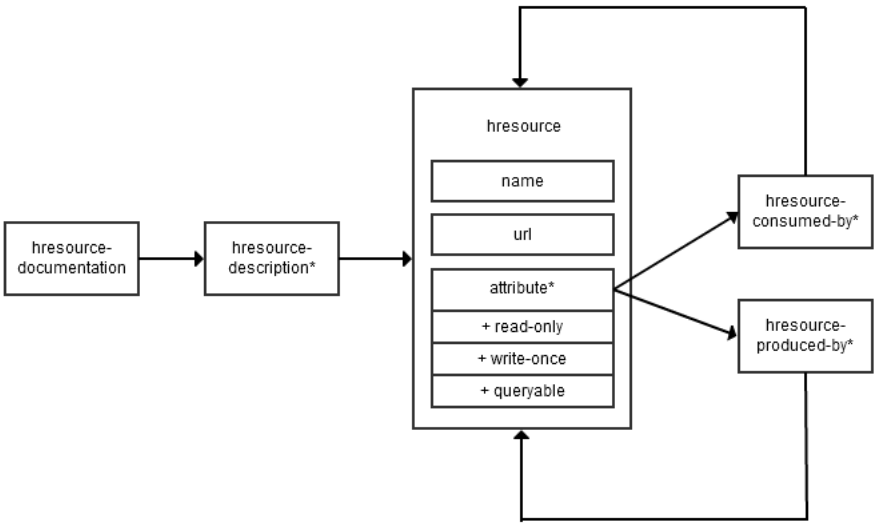

Figure 5.2: Relationship between the semantic annotations
('*' indicates a one-to- many relationship)

5.3 Service Discovery

The discovery mechanism works to enable discovery of new, similar and related services. Service discovery addresses two different aspects of the problem: discovery by users and discovery by services.

1. *Discover-as-you-browse*: This deals with enabling browsers (or browser extensions) to hint users about the presence of resources as they browse a website. This works similar to how users discover RSS feeds(Figure 5.5). link element provided by HTML is used to alert browsers about the presence of REST resources on the website. This can be supplemented with automatic client creation since the browser can now link to the resource descriptions and read the data from the page using DOM traversal. The syntax of the link tag to use is as follows:

```
<div class="hresource">
  <h1 class="name">User</h1>
  URL:
  <span class="url">
    http://example.com/api/user
  </span>
  <p class="comment">
    api desc(Human Readable) - 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">userid</li>
      </ul>
  </div>
  <ol>
    <li>
      <code class="attribute required write-once queryable">
        username
      </code> -<span class="comment">
      username -required, must be unique.</span>
    </li>
    <li>
```

Figure 5.3: Description for a User resource in an annotated web page -Part 1

```
      <code class="attribute required">
        password
      </code> -<span class="comment"> required.</span>
    </li>
    <li>
      <code class="attribute queryable">
      country</code>(<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">
       userid
      </code>-<span class="comment">userID returned as result</span>
    </li>
  </ol>
  <p>ERRORCODE</p>

  <ol>
    <li class="hresource-error">
      <code class="error-code">550</code> -<span class="comment">
      no such user</span>
  </ol>
</div>
```

Figure 5.4: Description for a User resource in an annotated web page. - Part 2

```
<link rel="hresource-documentation"
    href="http://example.com/api/" />
```

The tag goes into the header of a page and the value hresource-documentation in the rel attribute specifies that the linked item is a REST resource. From the documentation page, links to individual resources in the API are annotated with hresource-description as the value of the rel attribute:

```
<a rel="hresource-description"
        href="http://example.com/api/user/">User</a>
```

The annotation is added to each resource linked from the API documentation start page. The client thus traverses from the homepage to the documentation page and from there to the individual resource descriptions to discover the resources and present this to the user.

2. *Automated Discovery*: Automated discovery deals with the ability of a service to discover similar services in the same domain and to link to them. Service discovery allows clients to find services that could provide data required to access another service or could consume data received from another service. Existing service discovery mechanisms use a directory-oriented approach and are suited only for SOAP based services. The new description syntax provides a discovery mechanism for RESTful services that works peer-to-peer without any dependence on a central controller.

   The system works by identifying the different links as it comes across new resources and building up a graph connecting them. At a later stage, this graph could be traversed to discover new possibilities and to look for other sources of input.

Figure 5.5: The RSS discovery mechanism already present in modern browsers. The toolbar button shows a badge with the number of resources found when visiting a site with embedded resource descriptions.

The following annotations for inter-links between services are defined to enable discovery:

1. *Link to Superclass*: 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 superclass 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="hresource-is-a"
                href="http://dublincore.org/book/">
                Book
        </a>
```

   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://bookshop.com/book-order\#isbn
            </li>
            <li rel="hresource-consumed-by">
               http://library.com/rented-book\#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`.

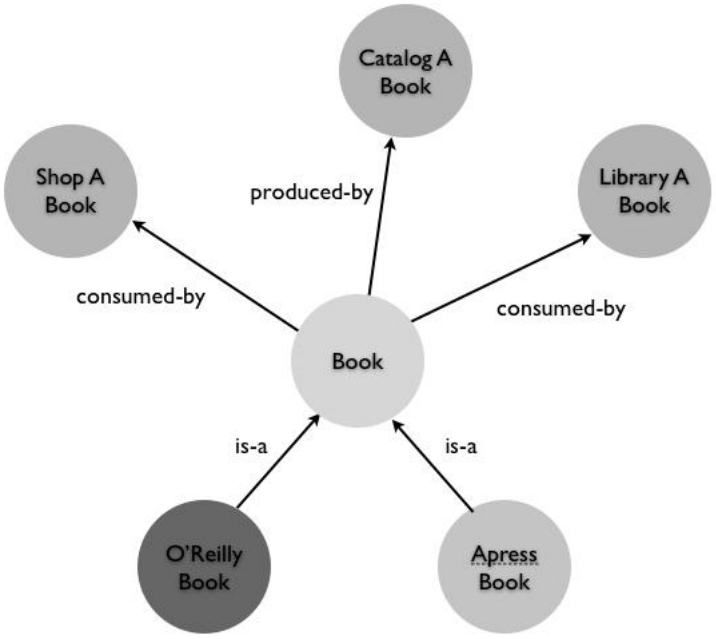

Figure 5.6: A graph constructed from the links between the resources.

5.4 Composition

Service composition is made possible by using the same annotations that were made for discovery. A graph is constructed starting from a resource and then traversing the parent, consumer and producer links recursively. At each page, the descriptions are extracted and converted to RDF to update the graph. This way, a software agent that does not have the identifier or a search parameter to access a specific resource could traverse the graph to figure out what other information could be used to lookup the identifier and present the choice to the user.

For e.g., the service provided by a bookshop might need the ISBN to order a book. However, the traversal of the graph could reveal a catalog service

that retrieves book resources using titles or author names. This allows the software agent to provide a choice to the user where he can enter either the ISBN or the title.

This works in the reverse direction also. Having received access to a resource, the software agent can suggest what all operations can accept the resource. So effectively a service that has a book resource can provide options for services from shops that let the user order the book or libraries that let the user lend the book .

Chapter 6

IMPLEMENTATION

The proposed system currently addresses the composition of RESTful web services that represent resources using the JavaScript Object Notation (JSON). The system expexts 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.

The system uses a web UI developed in HTML, JavaScript as the frontend 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 orocessing 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 illustratred in Figure: 6.1

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 users input of which APIs to use for composition, the server fetches each of the required annotated API documentation pages and parses them. Based on the information

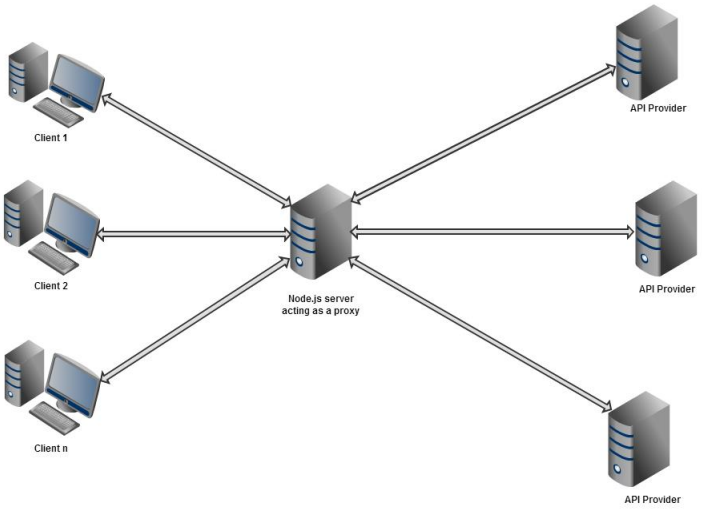

Figure 6.1: System architecture

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 descipbe the required composition of web services. Once the mashup is graphical design of the mashup 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 reqired API calls asynchronouly and composes them as required and produces an HTML formatted output which is the required mashup. The HTML output is then served back to the client system for the user. The interaction flow in the system is illustrated in Figure: 6.2

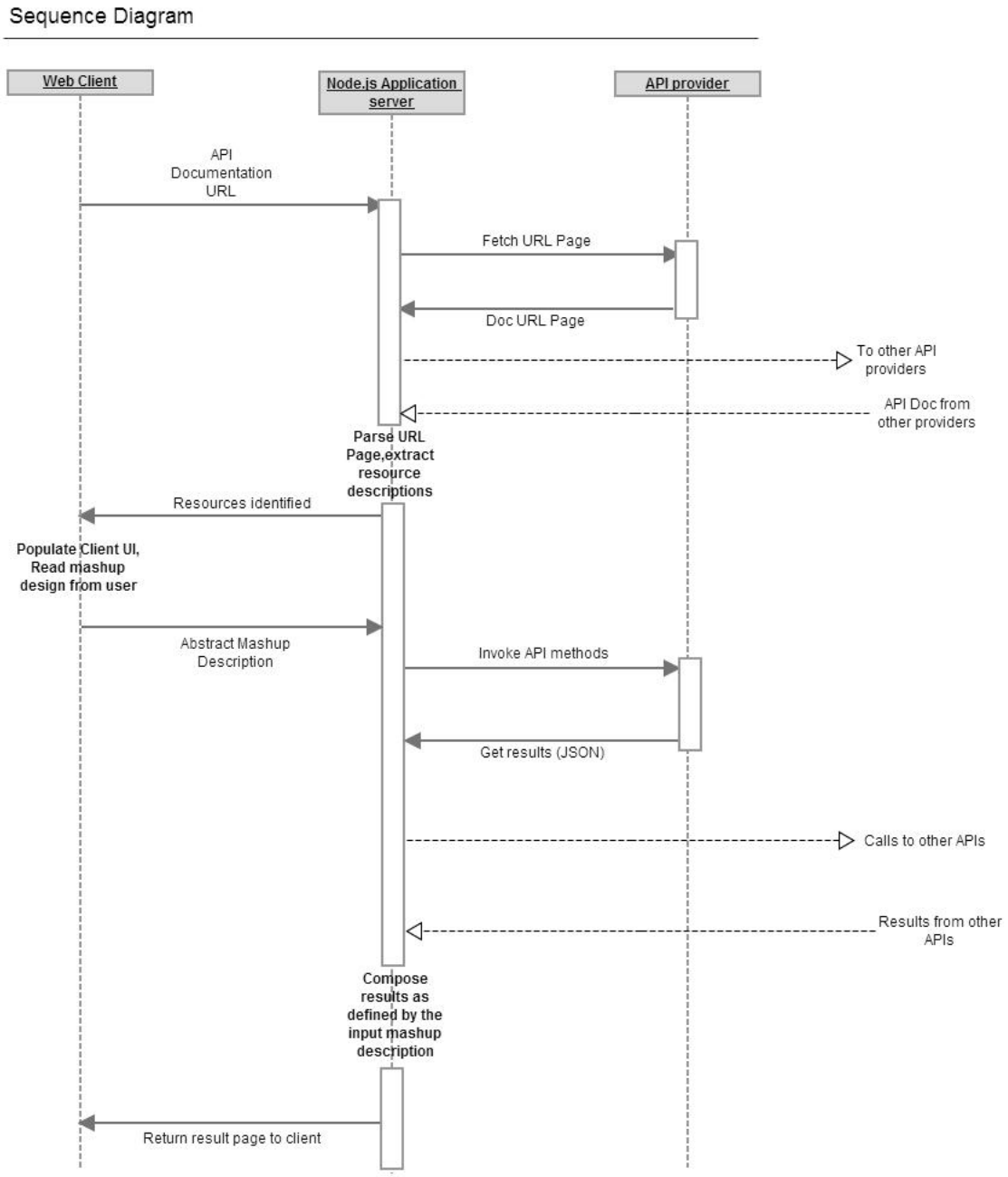

Figure 6.2: Interaction flow sequence

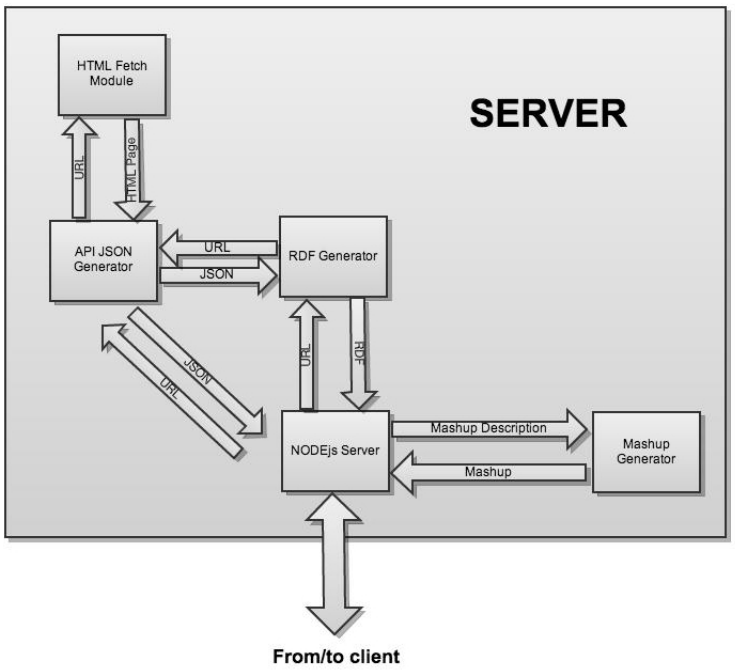

Figure 6.3: Server Architecture

6.1 Server side Implementation

The architecture of the server is as illustrated in Figure: 6.3.Server primarily deals with providing three services:

1. **Parsing API Documentation HTML**: Client side provides a URL to 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 produce the output JSON. The *produced-by - consumed-by* 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 producer-consumer graph and fetches each of the required resources upto some arbitrary level of nesting. Once the resource fetching is done RDF can be generated.

3. **Request handling and mashup generation**: The user specifeis the required composition of services graphically in the client workbench. An abstract description of the required mashup 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 mashup. 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.

6.2 Client side Implementation

Client provides a simple drag and drop based user interface for mashup design. The elements in the UI are populated from the contents of the API documantations 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 two basic data structures which are simple JSON objects.

- **docjsons**: An array of JSON description of all APIs loaded into the workbench
- **attributearray**: An associative array indexed using the ID of the API call used (one for each widget used in the workbench). It has the following components:

1. *jindex*: Index of document JSON in array of JSONs.
2. *apiindex*: Index of the particular API used in this JSON
3. *isStartNode*: True if there are no inputs piped into this node ,i.e, this node is a start point in the mashup design with inputs from the user and not from other API calls.
4. *method*: HTTP method selected by the user.
5. *inputs*: An associative array of inputs indexed by attribute name.
6. *outputs*: An associative array of outputs indexed by attribute name.
7. *headers*: An associative array containing the headers to be passed (if required) with an API call, indexed by attribute name.

Each element in the inputs array has two attributes :

- *value*: Value of the attribute or null.
- *link*: Null or a tuple (id of the producer API call,name of the attribute producing it)

Each element in the outputs array has a single attribute:

- *link*: Null or a tuple (id of the consumer API call,name of the attribute that consumes it)

The link attributes forms a logical graph of the way in which the different service calls are composed. An abstract representation of the mashup is created by traversing the graph. The traversal is done by using a simple modification to the 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.

6.3 Tools and Technologies

6.3.1 Resource Description Framework (RDF)

RDF is a W3C standard[18] for modeling and sharing distributed knowledge based on a decentralized open-world assumption. Any knowledge about anything can be decomposed into triples (3-tuples) consisting of subject, predicate, and object; essentially, RDF is the lowest common denominator for exchanging data between systems.RDF is designed to be read and understood by machines.It is normally written in XML.RDF is a part of W3C's semantic web activity.

What makes RDF suited for distributed knowledge is that RDF applications can put together RDF files posted by different people around the Internet and easily learn from them new things that no single document asserted. It does this in two ways, first by linking documents together by the common vocabularies they use, and second by allowing any document to use any vocabulary. This flexibility is fairly unique to RDF.

The use cases of RDF, as described by Richard Cyganiak on the W3C's Semantic Web activiy is as follows:

- to integrate data from different sources without custom programming.
- to offer your data for re-use by other parties.
- to decentralize data in a way that no single party *owns* all the data.

RDF describes resources as tripples with subject as the identifier of the resource,predicate as the property and object as the value of the property.The linking structure of RDF forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes.This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand

visual explanations.

6.3.2 Bootstrap Framework

Bootstrap is a front-end toolkit developed by twitter for rapidly developing web applications[19].It contains HTML and CSS-based design templates for typography, forms, buttons, charts, navigation and other interface components, as well as optional JavaScript extensions. At its core, Bootstrap is just CSS, but it's built with Less, a flexible pre-processor that offers much more power and flexibility than regular CSS. With Less, it gains a range of features like nested declarations, variables, mixins, operations, and color functions. Additionally, since Bootstrap is purely CSS when compiled via Less, two important benefits are observed:

First, Bootstrap remains very easy to implement; just drop it in your code and go. Compiling Less can be accomplished via Javascript, an unofficial Mac application, or via Node.js.

Second, once complied, Bootstrap contains nothing but CSS, meaning there are no superfluous images, Flash, or Javascript.All that remains is simple and powerful CSS for your web development needs.

We used bootstrap for cross browser compatibility.The front end of the client side is built on top of bootstrap framework.

6.3.3 JSPlumb

jsPlumb provides a means for a developer to visually connect elements on their web pages.It uses Bezier curve functions for plotting connections in the user interface.jsPlumb can be used with jQuery, MooTools or YUI3 (or another library of your choice).It provides an API with many options like bind(Bind to an event on jsPlumb),connect(establish a connection between two elements),select(select a set

of connections using filter option) etc.To connect between any two elements in the UI,the id of the source and target must be specified.

We used jsPlumb to visualize the creation of the mashups.When resource description embedded in a web page is parsed,the available resources are presented to the user.The user can then make connections between the resources to form mashups.Whenever a connection is specified by the user the input,output parameters to create mashups are collected from the user,and a client is generated automatically.

6.3.4 Node.js

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. It is a server-side system for writing scalable internet applications ,notably web servers.

Node.js contains a built-in HTTP server library[22],making it possible to run a web server without the use of external software such as Apache or Lighttpd, and allowing more control of how the web server works. Node.js enables web developers to create an entire web application in JavaScript,both server-side and client-side. Node.js is very fast(event-loop non-blocking) and also has very speedy native bindings. Performance is the main advantage of Node.js; it allocates a small heap per each connection, while other server side solutions create a (2MB) thread for each incoming connection, and of course creating a thread is much slower than allocating heap memory. Also being written in JavaScript lowers the barrier to entry for most front-end developers who are already used to working with the language.

Node.js is used in this design to solve the problem of making cross domain

calls. Apart from that, it provides good performance with large no of services used to create mashups. We stick to Node.js with the thought that an event driven architecture can lead to more scalable applications. Also it is lightweight and perfect for data-intensive real-time applications that run across distributed devices.

Chapter 7

OUTPUT

The screenshots of the prototype program developed. The entire source code is available in GitHub - *https://github.com/abilng/sMash.it*

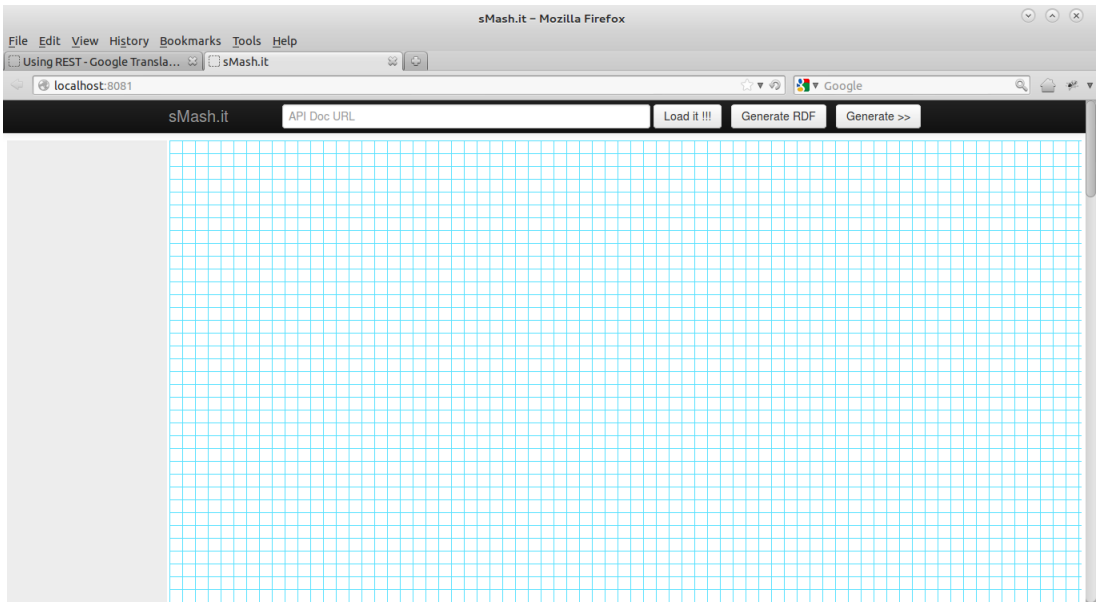

Figure 7.1: Client Homescreen and workbench

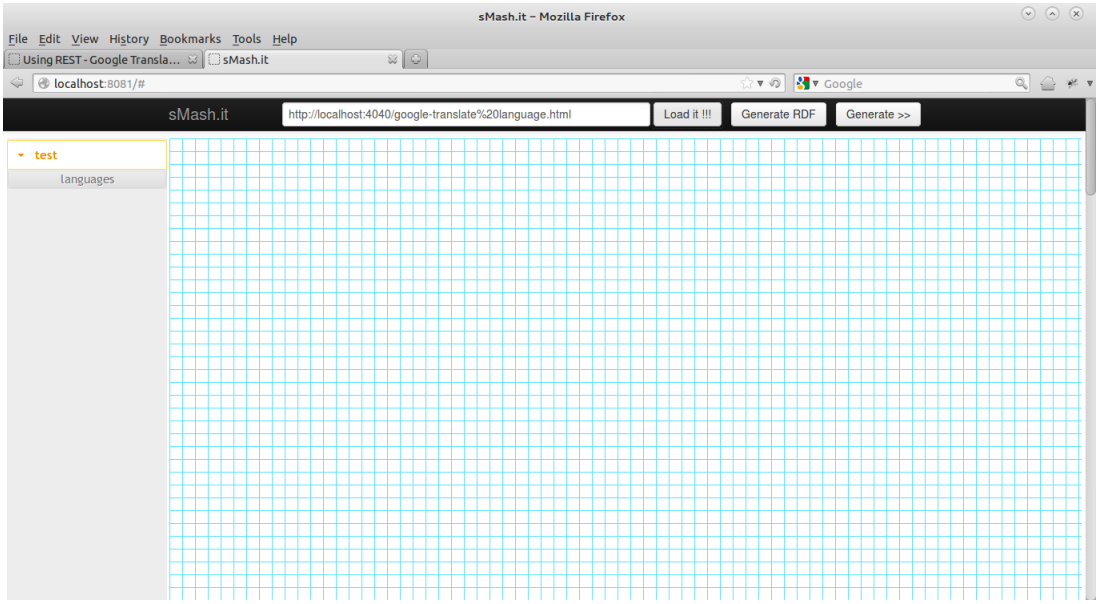

Figure 7.2: Loading an API

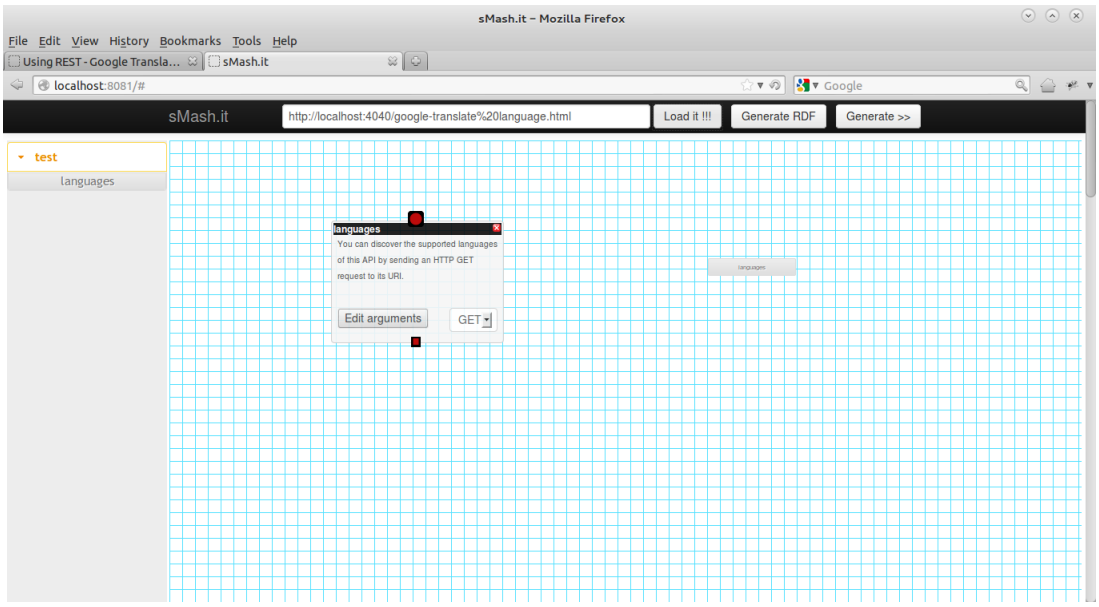

Figure 7.3: Mashup design using drag n drop - step 1

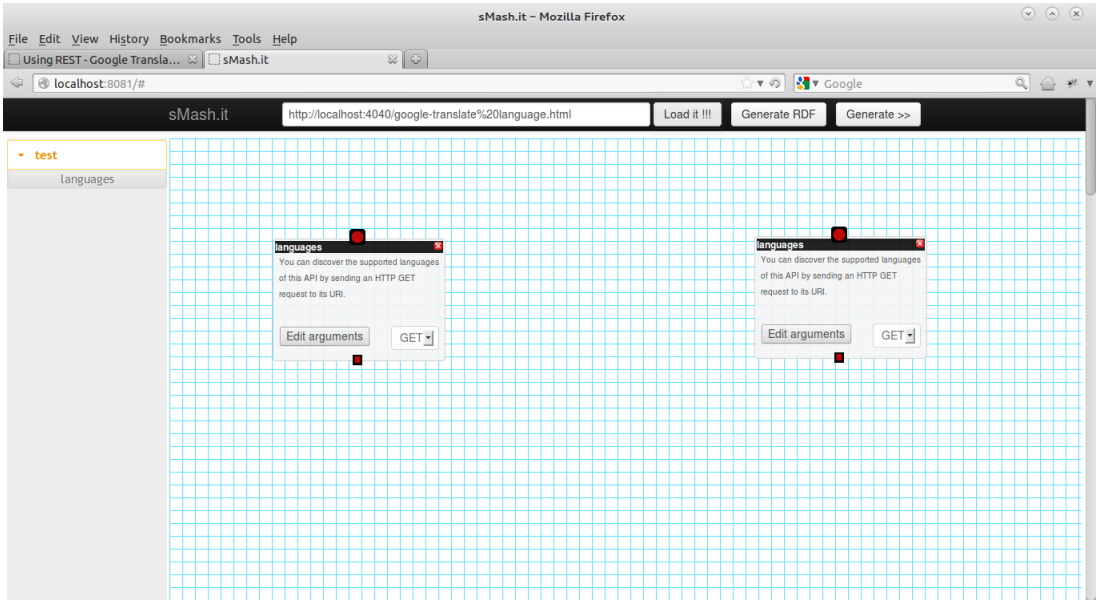

Figure 7.4: Mashup design using drag n drop -step 2

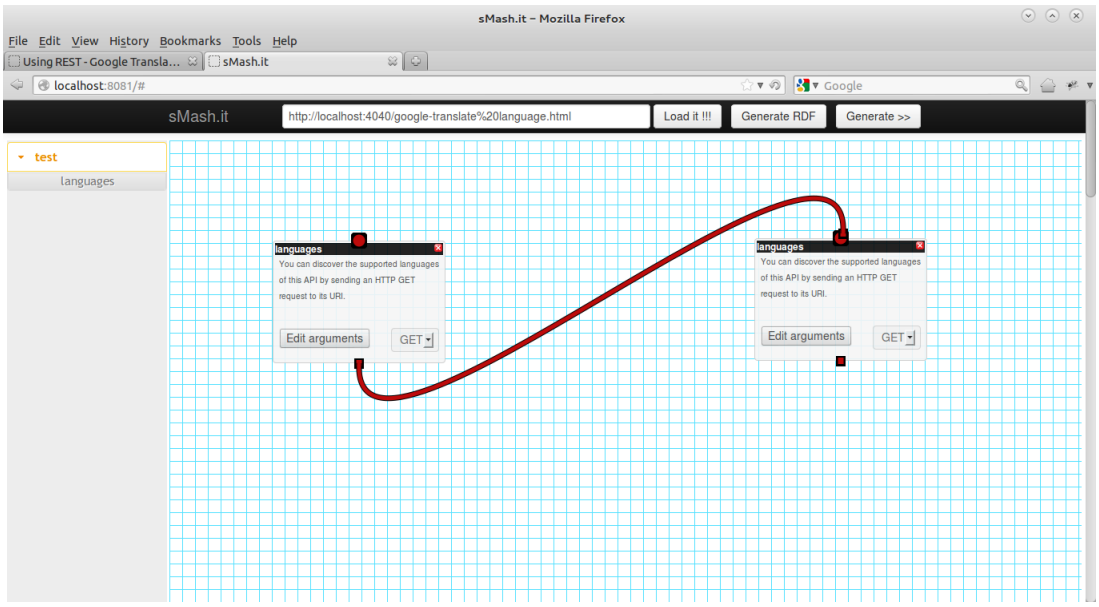

Figure 7.5: Mashup design using drag n drop -step 3

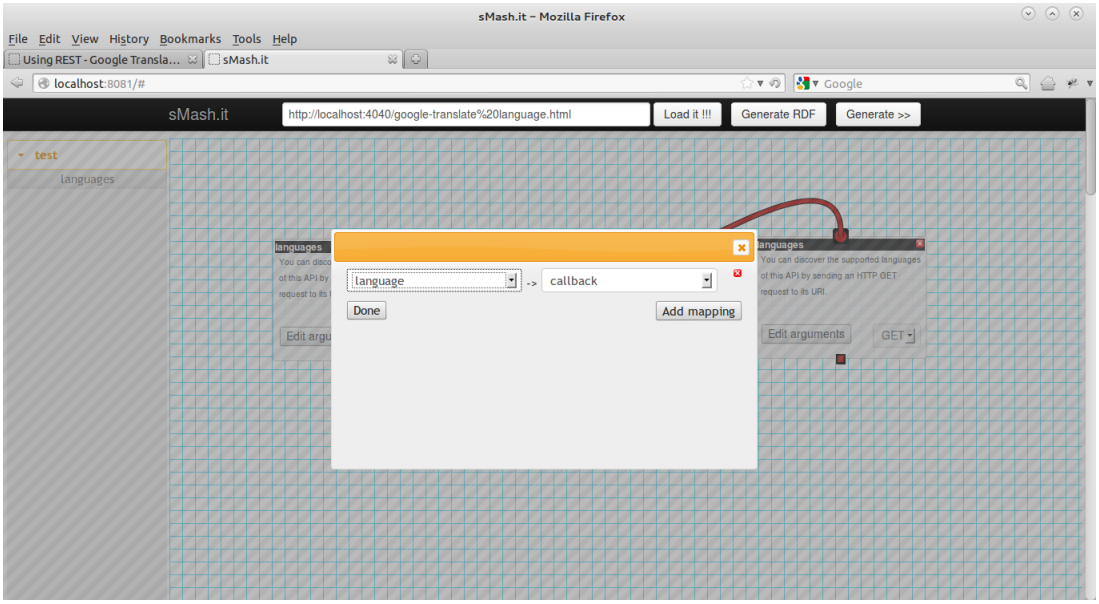

Figure 7.6: Mashup creation-step 4, Parameter mapping.

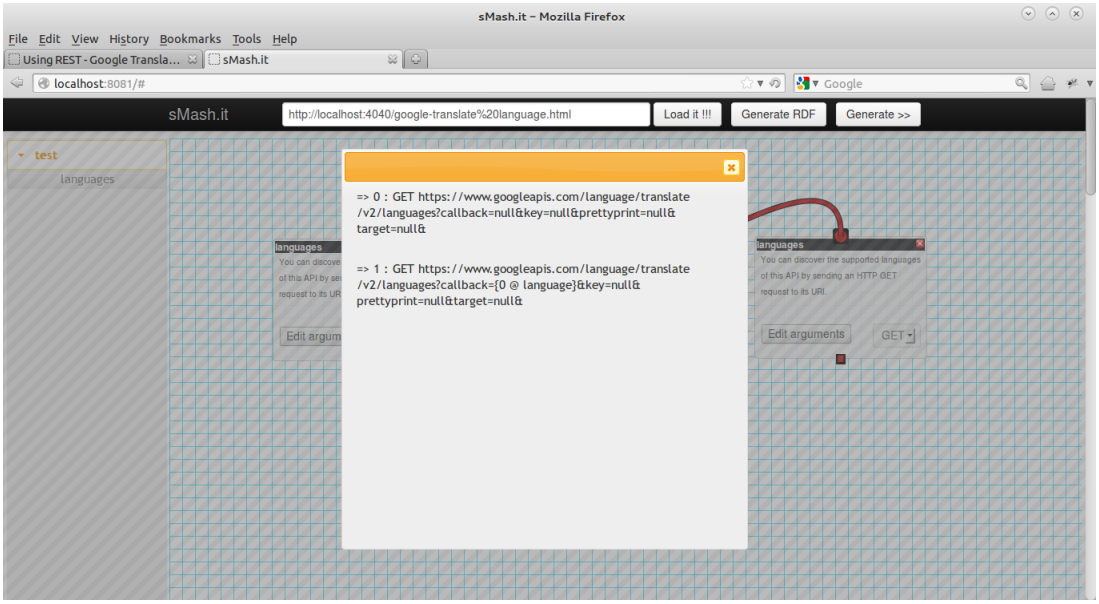

Figure 7.7: Invocation sequence generated.

Chapter 8

Conclusion

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

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

The prototype mashup editor created can intelligintly mash the services based on annotoations But it is also limited in some aspects. But we hope this direction deserves more exploration since it is relativly easy for the developer and the machine to follow. The prototype was implimented in different paradigms to verify the computatuiional completetability.

The imeediate future directions we would like to purse this works are

1. Change to accomodate todays REST APIs. Most of the rest apis in use today does to conform to the RESTful paradigm. They uses GET heavily to get most of the things done for perfomance purposes. This have to taken care of.
2. A tool to convert exishting REST APIs. But this comes with lot of challenge. The above one for a start, adoption is another problem.

Bibliography

[1] 1. Christian Bizer, Tom Heath, Tim Berners-Lee, *1. Linked Data - The Story So Far*. in International Journal on Semantic Web and Information Systems IJSWIS, 2009..

[2] Frank Manola, Eric Miller, Brian McBride, *RDF Primer*, W3C Recommendation, 10 February 2004.

[3] Dborah L. McGuinness, Frank van Harmelen, *OWL Web Ontology Language: Overview*, W3C Recommendation, 10 February 2004.

[4] W3C OWL Working Group, *OWL 2 Web Ontology Language: Document Overview*, W3C Recommendation, 27 October 2009.

[5] Dan Brickley, R.V. Guha, Brian McBride, *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C Recommendation, 10 February 2004.

[6] Microformats - *http://www.microformats.org/.*

[7] Ben Adida, Mark Birbeck, Shane McCarron, Steven Pemberton, *RDFa in XHTML: Syntax and Processing: A collection of attributes and processing rules for extending XHTML to support RDF*, W3C Recommendation, 14 October 2008.

[8] Ben Adida, Mark Birbeck, *RDFa Primer - Bridging the Human and Data Webs*, W3C Recommendation, 14 October 2008.

[9] R.T. Fielding, *Architectural styles and the design of network-based software architectures*, PhD dissertation, Department of Information and Computer Science, University of California, Irvine, 2000.

[10] Roberto Chinnici,Jean-Jacques Moreau, Arthur Ryman, Sanjiva Weerawarana, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Recommendation, 26 June 2007.

[11] Marc Hadley, *Web Application Description Language*, W3C Member Submission, 31 August 2009.

[12] Jacek Kopecky, Karthik Gomadam, Tomas Vitvar, *hRESTS: an HTML Microformat for Describing RESTful Web Services,* IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 2008.

[13] Karthik Gomadam, Ajith Ranabahu, Amit Sheth, *SA-REST: Semantic Annotation of Web Resources*, W3C Member Submission, 05 April 2010.

[14] Markus Lanthaler, Christian Gtl, *A Semantic Description Language for RESTful Data Services to Combat Semaphobia*, 5th IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2011), 31 May -3 June 2011, Daejeon, Korea.

[15] Douglas Crockford, *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)*, Network Working Group, July 2006.

[16] jsPlumb documentation - *http://jsplumbtoolkit.com/doc/home.html*

[17] API Documentation - *http://jsplumbtoolkit.com/apidocs/files/jsPlumb-1.4.1-apidoc.html*

[18] W3C's List of Documents about RDF *http://www.w3.org/standards/techs/rdf* http://www.rdfabout.com/

[19] Bootstrap - *http://twitter.github.io/bootstrap/index.html*

[20] IFTTT Service *ifttt.com*

[21] Yahoo! Pipes *http://pipes.yahoo.com*

[22] Node.js API Documentation - *http://nodejs.org/api/*