# OmniVoke: A Framework for Automating the Invocation of Web APIs

Ning Li, Carlos Pedrinaci, Maria Maleshkova, Jacek Kopecky, John Domingue
Knowledge Media Institute (KMi), The Open University
Milton Keynes, United Kingdom
{n.li, c.pedrinaci, m.maleshkova, j.kopecky, j.domingue}@open.ac.uk

***Abstract*—**Web APIs, characterized by their relative simplicity and their natural suitability for the Web, have become increasingly dominant in the world of services on the Web. Despite their popularity, Web APIs are so heterogeneous in terms of the underlying principles adopted and the means used for publishing them that discovering, understanding and notably invoking Web APIs is nowadays more an art than a science. In this paper, we present our work towards supporting the automated invocation of Web APIs. In particular, we describe a framework that provides a unique entry point for the invocation of most Web APIs that can be found on the Web, by exploiting non-intrusive semantic annotations of HTML pages describing Web APIs in order to capture both their semantics as well as information necessary to carry out their invocation.

***Keywords-semantic Web services; Web APIs; service invocation; grounding;***

## I. INTRODUCTION

The world of services on the Web has recently been marked by the proliferation of Web APIs, which seem to be preferred over "classical" Web services based on WSDL and SOAP [1-3]. Web APIs, also called RESTful services when they conform to REST principles [4] are used by most major Web sites such as Facebook, Flickr, Salesforce or Amazon to provide access to their data and functionality. This trend is largely impelled by the simplicity of the technology stack, essentially URIs and HTTP, as well as by the rapidity with which third parties are combining diverse APIs into mashups that provide added-value solutions [3]. Yet, Web APIs are most often described solely through HTML Web pages that are thought for humans and pose outstanding difficulties for their automated identification and interpretation in order to support, for example, automated Web API discovery and invocation [1].

Despite their popularity, the use of Web APIs still requires therefore extensive manual effort, which involves developing custom tailored software that can hardly be reused. A number of researchers and developers are devising generic solutions for better supporting the description, discovery, and composition of Web APIs [3, 5, 6, 7]. These approaches build upon the wealth of research on "classical" Web services and adapt it to deal with Web APIs. Yet, a quick look at some of the existing Web APIs shows significant differences when compared to "classical" Web services. The most notable distinction lies indeed in the fact that there is no established interface definition language (IDL), despite some proposals like WADL [5], or even WSDL 2.0 [8]. In practice, the uptake of these languages has been very limited and most Web APIs solely provide human-oriented HTML documentation [1]. In the light of these very characteristics, researchers have also focused on providing support for enriching the HTML documentation of Web APIs with semantic annotations and exploiting these annotations for the identification and advanced discovery of Web APIs, see for instance [7, 9, 10, 11].

Although considerable advances have been obtained, supporting the automated invocation of Web APIs is yet to be addressed comprehensively. To our knowledge, the main results in this respect, see e.g. [12, 13], fail to provide a solution generic enough to support the automated invocation of the extremely heterogeneous APIs that can be found on the Web nowadays. The reason for this is twofold. On the one hand, automated service invocation has not been prominently addressed by semantic Web services researchers because the invocation of "classical" Web services is directly supported by WSDL (the main concern in this case is data transformation). On the other hand, supporting the invocation of Web APIs generically presents a number of outstanding and somewhat unexpected challenges given the previous experience with WSDL services. These challenges originate from the lack of an established IDL and the remarkable heterogeneity in Web APIs idiosyncrasy, technical characteristics, specification formats, and even under-specification in many cases [1].

In this paper, we present OmniVoke, a framework that aims to automating the invocation of generic Web APIs. The framework provides a unique entry point for the invocation of most Web APIs that can be found on the Web, The framework thus abstracts away the heterogeneities of different APIs and consequently eliminates the need for developing a custom tailored client per Web API. Our framework relies on non-intrusive semantic annotations of HTML pages describing Web APIs, in order to capture both their semantics as well as the information necessary to carry out their invocation. The framework developed is based on RESTful principles to simplify its use and to adequately exploit the Web infrastructure for scalability. The engine includes a RESTful interface enabling the invocations as well as the monitoring or post-mortem analysis of APIs execution by publishing associated artifacts generated or used during the interaction with remote APIs, such as messages exchanged, etc.

This paper is organized as follows: In Section II, we firstly provide background information covering notably the heterogeneity of the current Web APIs, and the derived complexity

This work was partly funded by the EU project SOA4All (FP7-215219).

towards their automated invocation. Section III presents our extensions to an existing semantic model for describing Web APIs including support for capturing all the information necessary for invoking the majority of the existing Web APIs. In Section IV, we describe the functionality of the proposed framework OmniVoke. Section V provides the architecture, components and interfaces of OmniVoke. In Section VI, we give implementation details of the proposed framework with illustrations. Section VII presents related work. In Section VIII, we conclude the paper and discuss future works.

## II. BACKGROUND

### A. Invocation of Web APIs

The invocation of Web APIs, requires composing an HTTP request, sending it and processing the response. In order to do this, one needs information about the address of the service, the HTTP method, the input/output data type(s), input/output data format(s), error types, etc. These aspects are indeed highly heterogeneous depending on the actual service at hand, and are by no means a specific problem related to Web APIs. However, they certainly have highlighted in the past the need for capturing formally the semantics of services in order to adequately support their management and invocation.

An analysis of existing APIs [1] shows that Web APIs exhibit an additional and more fundamental level of heterogeneity concerning their idiosyncrasy. In particular, we shall use herein the service classification given in [14], whereby services can be one out of the three following types: 1) RESTful resource-centered services which conform to the representational state transfer (REST) paradigm [4] and expose a URI for every piece of data the client might operate on using HTTP methods; 2) REST-RPC hybrid services, which, although mostly resource-centered, expose a URI for every operation the client might perform on a resource, e.g. one URI to fetch the data and another one to delete the data; and 3) RPC-style services that expose one URI for every process, be it resource-centered or action-centered, capable of handling RPC over various protocols.

In correspondence with this classification, we name a few heterogeneities of Web APIs that are crucial for their invocation.

- **URI format**. The form of the URI used is closely related to the implementation style of the service. For resource-oriented APIs, resources can be identified and accessed with directory-structure-like URIs, or via operation URIs. For example, the same sports news can be retrieved from a news website by using HTTP GET *http://url/.../news/sports* in a RESTful way or by using HTTP GET *http://url/.../getNews?type=sports* in a REST-RPC hybrid way. A survey on ProgrammableWeb[1], the largest reposity for Web APIs online, has found that almost two thirds of the Web APIs are either REST-RPC hybrid or RPC-style services and only one third are strict RESTful APIs as of Feb. 2010 [1].
- **Input data grounding**. Closely related to the variations in URI forms, when it comes to invoking a Web API given input data, it is necessary to make decisions upon where the input data is grounded in. In some cases, input data is to be used as part of the URI. In other cases, it is used in message header or in message body. This is very specific to Web APIs as opposed to WSDL-based services, which were just about constructing a SOAP message. Taking authentication credentials input as an example, our survey found that 70% of the Web APIs that require authentication send authentication information directly in the URI, while less than one third require that the HTTP header be constructed. Accordingly, these numbers are similar for invocation in general, where about one third of the APIs require the construction of the HTTP header or body, while the rest send input information through the URI [1].
- **Input/Output data format**. While XML has been a predominant format for data transfer over the Web, e.g. in WSDL/SOAP-based services, JSON has become increasing popular in the realm of Web APIs. Content negotiation may be required in order to produce output as well as consuming input in the right representation formats to avoid service failure. Though there exist other formats, such as HTML, RSS, Object etc., providing support for the use of XML and JSON addresses the vast majority of the APIs [1].

### B. Semantic Annotation and Description of Web APIs

The information of Web APIs usually needs to be captured in common documentation formats in order to facilitate the proper usage, including invocation. The effort towards automating the interaction with APIs needs to firstly address the issues of how to explicitly describe the semantics of such information. Since most Web APIs rely only on HTML documentation with no fixed structure or content, hRESTS [7], a microformat using *class* and *rel* XHTML attributes, was proposed to enable the creation of machine-processable descriptions on top of existing HTML descriptions. hRESTS introduces tags for marking the **service** description as a whole, the HTTP **method** being used, the **operation** with corresponding access **address**, **input** and **output**. MicroWSMO [7] builds on top of hRESTs, similar to SAWSDL [15] extending WSDL, to support the semantic annotation of the intended meaning of the tagged entities, which represent links to URIs of semantic concepts. In it, the **model** tag indicates that the URI is a link to an ontology entity, while **lifting** and **lowering** point to links for lifting and lowering transformations between the level of syntactic and semantic descriptions, which result in the creation of SAWSDL-like [15] annotations. Take Nestoria API[2], a property search service, as an example, an excerpt of the HTML description enhanced with hRESTS and MicroWSMO annotations is shown in Listing 1. For clarity, only the "search listings"[3] operation is described and information irrelevant to hRESTS and MicroWSMO annotations in the original HTML document has been removed. It shows that the service *NestoriaService* has a *SearchListingsOperation*. It has three input parameters *place_name*, *price_min* and *price_max*, each of which has model reference(s) to external ontology entities, i.e. to a real

[1] http://www.programmableweb.com

[2] http://www.nestoria.co.uk/help/api

[3] http://www.nestoria.co.uk/help/api-search-listings

estate ontology, and share one common lowering script. Thanks to the Semantic Web sErvices Editing Tool (SWEET)[4] [10], any Web APIs with descriptions in plain HTML can be annotated with hRESTS and MicroWSMO, which later can be transformed to an RDF representation described by a service ontology model (described next) through XSLT for example.

LISTING 1. NESTORIA SERVICE HRESTS DESCRIPTION

```
1  <div class="service" id="s1"><h1> NestoriaService</h1>
2  <div class="operation" id="op1">
3  <h2><span class="label">SearchListingsOperation</span></h2>
4  <span class="address">http://api.nestoria.co.uk/api?action=search_listings
5  </span>
5  <span class="input">
6  <h3><a rel="lowering" href=
7  "http:// iserve-dev.kmi.open.ac.uk/lilo/SearchListingsLowering.xs">
8  lowering</a></h3>
9  <h3><a rel="model" href=
10 "http://iserve.kmi.open.ac.uk/ontology/location.rdf#hasName">
10 <a rel="model" href=
11 "http://ierve.kmi.open.ac.uk/ontology/location.rdf#hasPostcode">
12 place_name </a></a></h3>
13 <h3><a rel="model"  href=
14 "http://iserve.kmi.open.ac.uk/ontology/estate.rdf#hasMinPrice">
15 price_min</a></h3>
16 <h3><a rel="model" href=
17 "http://iserve.kmi.open.ac.uk/ontology/estate.rdf#hasMaxPrice">
18 price_max</a></h3>
19 </span></div></div>
```

The service ontology model, referred to as Minimal Service Model (MSM) [11], is a simple RDF(S) ontology that was designed to capture the core semantics of both "classical "Web services and Web APIs in a common model. From Web APIs perspective, it provides the RDF vocabulary to describe semantics associated with HTML service description annotated with hRESTS and MicroWSMO. It defines four main concepts, including **Service**, **Operation**, **MessageContent** and **MessagePart**. A **Service** has a number of **Operations**, which in turn have input and output **MessageContent**. Faults are also defined as input or output faults of operations in the form of **MessageContent**. The **MessageContent** itself may contain optional or mandatory **MessagePart**. The purpose of the message part mechanism is to enable the definition of individual parameter mappings, groundings as well as additional characteristics, which apply only to parts of the input as opposed to it as a whole massage. It uses the SAWSDL RDF vocabulary[5] to capture the three kinds of annotations, made through MicroWSMO, that hook service descriptions with semantic concepts or data transformation scripts. RDF vocabularies describing attributes particular to Web APIs are enclosed as hRESTS vocabulary, i.e. under hRESTS namespace. Originally, hRESTS included some basic grounding mechanisms, namely the concept URITemplate to describe the URI for invocation, but this had substantial limitations which we had to address while developing OmniVoke. For more details of the MSM, please refer to *http://purl.org/msm/1.0* and [11]. The RDF description for the Nestoria API transformed from the annotated HTML in Listing 1 is shown in Listing 2, which, besides those directly obtained from Listing 1, contains also descriptions that support invocation of the service. These additional descriptions are based on the MSM extensions developed in this paper which will be described in Section III. As such, the MSM provides a common vocabulary, able to describe services in a way that allows machines to directly interpret their semantics, for supporting service tasks like discovery. It initially provides supports for common publishing and search of services, yet still permitting extensions when such a need arises, such as invocation.

### C. *Invoking Web APIs through Semantic Description*

The information concerning the invocation of Web APIs, as described in Section II A, is fundamental for supporting the creation of client application. Similar to automating other service tasks, automating the invocation of Web APIs rely on semantic extensions to service properties. In Section III, we will describe how we have extended the MSM with description mechanisms to support service invocation. In Section IV, V and VI, we will present OmniVoke, a generic Web API invocation client framework able to provide a unique entry point to the invocation of Web APIs through their semantic descriptions. Therefore, once Web APIs are semantically described, thus obtained semantic service descriptions need to be made available for automated interpretation by OmniVoke prior to an invocation call to the actual APIs. The semantic Web service publishing platform iServe [11], developed in the context of the EU project SOA4All[6], can be used for hosting semantic service descriptions, although this is not a requirement of the OmniVoke framework.

## III. EXTENDED MINIMAL SERVICE MODEL AND HRESTS

The Minimal Service Model, in its original design, was not particularly targeted nor well suited for supporting the invocation of Web APIs. The model especially failed to define input data grounding, which needs to specify whether the input values are transmitted as part of the URI, HTTP headers or the HTTP request message body (applicable to HTTP POST and PUT requests). In this section, we describe how we have extended the MSM with data grounding description mechanisms to support automated invocation of Web APIs. The extended MSM is published at *http://purl.org/msm/1.1*. As it relies on the syntactical structuring of the HTML documentation in terms of identifying service properties given by hRESTS annotation, an extension to the MSM is embodied in the extension to the hRESTS vocabulary. In Sections IV, V and VI we describe and illustrate how the OmniVoke framework we have developed exploits these annotations to automate the invocation of heterogeneous Web APIs.

As originally described, hRESTS expected a single lowering transformation, as shown in Listing 1, that would apply to the whole input message. In our extension, we allow finer-grained (and thus more reusable) lowering transformations on individual message parts. Every message part corresponds either to a URI parameter (specified in the URI template of a service's operation), a particular HTTP header or the HTTP body. To capture this correspondence, we introduce a property

[4] http://sweet.kmi.open.ac.uk

[5] http://www.w3.org/TR/sawsdl/#rdfmapping

[6] http://www.soa4all.eu

**isGroundedIn** whose value is either an identifier from the HTTP vocabulary [16] or a literal string. The HTTP vocabulary allows the **isGroundedIn** property to identify that the result of the lowering transformation becomes the value of a particular HTTP header, or of the HTTP body. If the **isGroundedIn** property has a literal value, it names the URI parameter that will contain the value of the lowering transformation.

In situations where input RDF data is grounded in HTTP body, the lowering schema mapping must lower the RDF data to a format supported by the actual Web API, for example JSON or XML, and OmniVoke must know what the format is, so it can include the appropriate Content-type HTTP header. Similarly, the response message body needs to be lifted to RDF using a lifting schema mapping transformation that supports its actual format; in other words, OmniVoke must select a lifting schema mapping transformation that supports the content type of the concrete response message. To do that, we add metadata on lowering and lifting schema mappings: for a lowering mapping, we describe the content type that the mapping produces (using the property **producesContentType**), and for a lifting mapping, we describe the content type that it can process (**acceptsContentType**). Both properties point to a literal string that names a MIME media type.

The three properties described above are RDF properties that extend the Minimal Service Model. In actual HTML service descriptions, they are included in the hRESTS microformat. The property **isGroundedIn** is represented with the new HTML class/link relation "grounding" (a class for literal values, a link relation for HTTP vocabulary identifiers) within an input description; and the properties **producesContentType** and **acceptsContentType** are captured with the new HTML class "content-type" within the lifting or lowering annotations. The HTML description with such annotation extensions will look similar to what has been shown in Listing 1.

## IV. OMNIVOKE FUNCTIONALITY

Appropriately supporting the use and management of heterogeneous services requires sharing the semantics of services through formal machine-processable descriptions as well as using a common syntax for representing these descriptions and the data exchanged or adequate transformation mechanisms. We previously introduced that we use RDF(S) for describing services semantically. We shall therefore also use RDF as lingua-franca for communicating with OmniVoke. Doing so carries out additional requirements like the need to provide mechanisms for transforming messages between RDF and the data format used by the Web APIs internally if necessary.

In more detail, we have identified the following requirements for OmniVoke:

- **Validate invocation request**. In general, an invocation request is issued to an operation of a Web API. Given semantic description of the operation to be invoked, OmniVoke should be able to carry out initial check whether the request is valid, e.g. whether the Web API indeed contains such an operation, by solely looking into its semantic description without communication to the actual Web API yet.
- **De-capsulate invocation request**, figure out what information should be handled in what way, i.e. what information is for OmniVoke's local use and what information should be passed onto the actual Web API for invocation use and how. For example, some information in service descriptions can be used to validate request by OmniVoke and the RDF input data, usually part of the body of the request message for its potentially large size, should be translated into the essential information for carrying out the actual invocation, such as a path parameter, a query string in address URI, a key-value pair in the HTTP header or an XML message in the HTTP body. This is facilitated by the "isGroundedIn", "producesContentType" service descriptions.
- **Map RDF input data to the expected data format used internally by the service implementation**, e.g. plain literal, XML, JSON etc. The decision may depend on where the input data is grounded in, e.g. the underspecified bits of address URI, message header, message body etc. The mapping process is referred to as "lowering" by the semantic web services community [15].
- **Compose the invocation request to the actual Web API** using the appropriate URI, method, message header, message body etc. Part of this information can be obtained directly from the service description, such as method, while other parts need to be constructed from the "lowered" input driven by its semantic service description. For example, the "isGroundedIn" property of the input determines where the "lowered" input is used, and the "producesContentType" property of the lowering schema mapping determines the format of the message body.
- **Invoke the actual Web API** when a valid request is composed.
- **De-capsulate the response message** obtained and figure out what information should be handled in what way. For example, non-RDF response message body needs to be "lifted" to RDF, as will be described next. Response status and error messages also need to be "lifted" if further usage of that information is anticipated. Standard HTTP error codes can be directly mapped to their counterparts in the standard HTTP vocabulary in RDF[7].
- **Map non-RDF data, e.g. XML, JSON in response message body to RDF**. This process is referred to as "lifting" in SAWSDL. If the API supports RDF output format, e.g., Sindice search API[8] or the Geonames search API[9], no lifting is needed. Lifting schema map-

[7] http://www.w3.org/TR/HTTP-in-RDF10/
[8] http://sindice.com/developers/searchapi
[9] http://www.geonames.org/export/ws-overview.html

ping will be required for non-standard HTTP error messages, as seen in Lastfm API[10].

## V. OMNIVOKE

### A. Architecture

With data sources on the Web undergoing a developing trend towards Linked Data [17], Web APIs, which provide on-the-fly computation of data resources through invocation, need to progress in order to continue playing their roles as Linked Data prosumers when invoked through semantic extensions. Therefore, OmniVoke takes RDF data as input and returns RDF data as response data, thus enabling a seamless integration of Web APIs, as semantic data prosumers [20], into the RDF linked data space. In order to carry out concrete invocations, the envisaged scenario is for applications to issue SPARQL queries to derive the data required for invoking a particular Web API [2]. Alternatively appropriate user-interaction interfaces can be provided to allow the user to provide his/her input, which, together with response data, may be collected into shared data space for further manipulations like inspection, reuse etc. In latter case, the user is typically presented with a set of input fields, which need to be completed in order to invoke the service. Semantic annotations or descriptions of the service can be attached here to aid the user. Additional information such as comments can be provided to support the user resolving any potential ambiguities. OmniVoke supports both means of deriving request data. Fig. 1 depicts the architecture of OmniVoke.

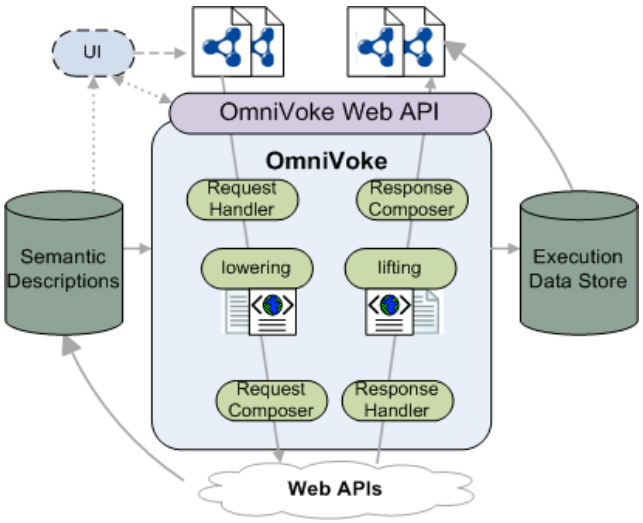


Figure 1. OmniVoke Architecture

### B. Components

In correspondence to its functionalities described in Section IV, OmniVoke consists of the following components, as drawn in Fig. 1.

- Request Handler

  The Request Handler is triggered when an invocation request is received. It carries out the tasks of validating and de-capsulating the invocation request. It acts in correspondence to the set of HTTP methods or verbs (e.g., GET, PUT, POST, or DELETE). If an action for a given verb is not defined, a request using such verb will be answered with HTTP code 405 (Method not allowed). There is no limit on the number of concurrent activation of such handler.

- Lowering

  The Lowering component undertakes tasks of "lowering" RDF input data to the format supported by the actual API. It works by executing the "lowering" scripts designed for each input that requires lowering and attached to that input in the service description, a mechanism proposed in SAWSDL. However, SASWDL imposes neither restrictions nor prescriptions on the choice of the script language. Programmatically, XSLT[11] together with SPARQL[12] has been used widely within the community. Lately, XSPARQL [18] which combines XQuery[13] and SPARQL have been recognized as a more effective language due to its advantages of avoiding the unnecessary detour of SPARQL query results.

- Request Composer

  The task of Request Composer is to construct a valid request for invoking the actual Web API using information given in service description and the "lowered" input, which now is in the form supported by the actual Web API.

- Response Handler

  Once the Web API is invoked and the response is returned, the Response Handler is triggered to de-capsulate the response, i.e. extract output information mainly status code, response data, out of response header, body etc., and decide whether lifting is required for each output, with the help of the service description.

- Lifting

  The Lifting component carries out the execution of "lifting" scripts attached to the output that requires lifting, as annotated in service description. Similar to "lowering", lifting scripts can be written in XSPARQL.

- Response Composer

  Once respective outputs are lifted to RDF, a new response, comprising only RDF data, is constructed by Response Composer and represented as the final response to the initial invocation request issued to OmniVoke.

[10] http://www.last.fm/api/show?service=270

[11] http://www.w3.org/TR/xslt
[12] http://www.w3.org/TR/rdf-sparql-query/
[13] http://www.w3.org/TR/xquery/

### C. Interfaces

The OmniVoke framework, when viewed as a semantic-processing layer wrapping around existing Web APIs for automating their invocation, should be exposed in a way that is amenable for access and manipulation by Linked Data as well as semantic applications. There are two aspects to the publication of the OmniVoke framework on the Web in correspondence to its function and resource aspects respectively, as explained in Section IV and illustrated in Fig. 1. The RDF resources, resulting from invocations, can be published in the resource-centric RESTful style. But how the functionality of the OmniVoke framework is exposed will have to take into account the characteristics of the actual Web APIs that are invoked through it.

Though every invocation request to a Web API is raised to the OmniVoke framework through a homogeneous interface, a URI that contains the identity of the API semantic description, seemingly a manipulation of resource and being RESTful. But when the request is parsed, after retrieving the semantic description and parsing RDF input data, to form the invocation request to the actual Web API, it no longer remains always as RESTful. As said, Web APIs include not just resource-oriented RESTful and REST-RPC hybrid services, but also the action-oriented RPC-style services. To support the invocation of all those service types and genuinely reflect, without modifying, adding or truncating, the information belonging to the actual Web APIs, OmniVoke should work out from their semantic descriptions the different styles, more specifically, the different interaction interfaces of the actual APIs being invoked and retain all information indispensable to the actual Web APIs.

## VI. OMNIVOKE IMPLEMENTATION

The OmniVoke framework exposes its functionality through a Web API and is published at *http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/*. Any Web API meant for autonomous invocation via the OmniVoke framework needs to have its interface semantically described using the extended MSM model. The semantic descriptions are then published on the semantic Web service publishing platform iServe in our implementation though this is not the only way to make service descriptions available and accessable. In iServe, a Unique Identity (UID) is allocated to every successfully published Web API description as semantic identity of the Web API, which will then be used to uniquely identify the URI for a request issued to OmniVoke for the ultimate invocation of the Web API. Given that a Web API usually contains not just one operation, an invocation request URI should also indicate which operation is to invoke. Therefore, an invocation request URI is available in the form *http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/service/{ServiceUID}/operation/{OperationName}/invoke*. The request is presented to OmniVoke via POST method because it triggers the creation of resources on the server and thus changes the state of the server. In particular, upon invocation, OmniVoke stores the RDF request data and the RDF response data for clients' later retrieval or inspection. The underspecified path parameters *ServiceUID* and *OperationName* can be obtained from iServe via the service discovery modules[14], or through the iServe brower[15] supported searching facility.

We still use the Netoria API to illustrate how OmniVoke is implemented and how it works with other components in the world of semantic web service to automate service invocation. First of all, Nestoria API semantic description is created based on its HTML documentation and then is published on iServe that allocates to it a UID, i.e. *c4e16ab6-3bad-47bb-b613-90ef78232e31*. The service description, in RDF, is retrievable through iServe's RESTful interface via a URI[16] featured by the UID for detailed inspection. A snippet of the RDF description is given in Listing 2.

LISTING 2. NESTORIA API SEMANTIC SERVICE DESCRIPTION.

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/services/c4e16ab6-3bad-47bb-b613-90ef78232e31#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix sawsdl: < http://www.w3.org/ns/sawsdl#>.
@prefix msm: <http://cms-wg.sti2.org/ns/minimal-service-model#>.
@prefix hrests: <http://purl.org/hrests/1.1#>.
@prefix estate: <http://iserve.kmi.open.ac.uk/ontology/estate.rdf#>.
@prefix location: <http://iserve.kmi.open.ac.uk/ontology/location.rdf#>.
1  :NestoriaService a msm:Service;
2      msm:hasOperation :SearchListingsOperation;
3      hrests:hasAddress "http://api.nestoria.co.uk/api?"^^hrests:URITemplate.
4  :SearchListingsOperation a msm:Operation;
5       msm:hasInput  :SearchListingsInput;
6       msm:hasOutput :SearchListingsOutput;
7       hrests:hasMethod "GET";
8       hrests:hasAddress
9  "action=search_listings&place_name={p1}&price_min={p2}
10 &price_max={p3}"^^hrests:URITemplate.
11 :SearchListingsInput a msm:MessageContent;
12       msm:hasPart :place_name, :price_max, :price_min.
13 :place_name a msm:MessagePart;
14       sawsdl:loweringSchemaMapping
15          "http://iserve-dev.kmi.open.ac.uk/lilo/PlaceNameLowering.xs";
16       sawsdl:modelReference location:hasName, location:hasPostcode;
17       hrests:isGroundedIn "p1"^^rdf:PlainLiteral.
18 :price_min a msm:MessagePart;
19        sawsdl:loweringSchemaMapping
20          "http://iserve-dev.kmi.open.ac.uk/lilo/PriceMinLowering.xs";
21        sawsdl:modelReference estate:hasMinPrice;
22        hrests:isGroundedIn "p2"^^rdf:PlainLiteral.
23 :price_max a msm:MessagePart;
24       sawsdl:loweringSchemaMapping
25          "http://iserve-dev.kmi.open.ac.uk/lilo/PriceMaxLowering.xs";
26       sawsdl:modelReference estate:hasMaxPrice;
27       hrests:isGroundedIn "p3"^^rdf:PlainLiteral.
28 :SearchListingsOutput a msm:MessageContent;
29       sawsdl:liftingSchemaMapping
30         "http://iserve-dev.kmi.open.ac.uk/lilo/SearchListingsLifting.txt";
```

It shows that the *NestoriaService* contains a *SearchListingsOperation*, its input message content SearchListingsInput that contains message parts *place_name*, *price_min* and *price_max*, and more importantly their links to external ontology entities,

[14] http://iserve.kmi.open.ac.uk/wiki/index.php/IServe_Higher_Level_Discovery_API

[15] http://iserve.kmi.open.ac.uk/browser.html

[16] http://iserve-dev.kmi.open.ac.uk/iserve/data/services/c4e16ab6-3bad-47bb-b613-90ef78232e31

i.e. to a real estate ontology, annotated with SAWSDL *modelReferece* as well as the links to *loweringSchemaMapping* and *liftingSchemaMapping* scripts, which are in the form of XSPARQL queries. The interested readers can obtain the concrete description from the given URI. Given the description, an invocation request can be issued to OmniVoke through the URI *http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/service/c4e16ab6-3bad-47bb-b613-90ef78232e31 /operation/SearchListingsOperation/invoke*.

In this example, the input to *SearchListingsOperation*, *place_name*, can be a village/town/place name, or a postcode. Its *loweringSchemaMapping* script, as indicated in Listing 2 line 14-15, is given in Listing 3, in which the *<file:StaticInputFile>* is a placeholder for input RDF data, dynamically obtained either from querying the shared RDF data space or from explicit user input through interaction interfaces. The service description indicates that *place_name* input is grounded in address URI as the value of the query string *p1* (Listing 2 line 17). That means *p1* will be substituted with the value of *place_name* once lowered from its RDF form to a plain literal by running the *loweringSchemaMapping* script. Similar pattern can be observed for inputs *price_min* and *price_max*, which are grounded also in address URI as values to the query string *p2* (line 22) and *p3* (line 27) respectively. With the actual invocation address URI jointly worked out from service address annotation and operation address annotation as *http://api.nestoria.co.uk/api?action=search_listings&place_name={p1}&price_min={p2}&price_max={p3}*, and a RDF input data as shown is Listing 4, the actual address URI will be derived as *http://api.nestoria.co.uk/api?action=search_listings&place_name=MiltonKeynes&price_min=100000&price_max=2000000*.

LISTING 3. LOWERINGSCHEMAMAPPING FOR PLACE_NAME INPUT

```
declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare namespace loc="http://iserve.kmi.open.ac.uk/ontology/location.rdf#";
declare namespace estate="http://iserve.kmi.open.ac.uk/ontology/estate.rdf#";

{ for $myLocation $myEstate $name $postcode from <file:StaticInputFile>
    where {  $myLocation a loc:Location.
               optional { $mylocation loc:hasName $name.}
               optional { $mylocation loc:hasPostcode $postcode.}
             $myEstate a estate:Property.
               optional { $myEstate estate:hasLocation $myLocation .}
          }
let $place_name :=if($name!="") then $name else $postcode

 return
     {$place_name}
}
```

LISINTG 4. RDF INPUT DATA FOR SEARCHLISTINGSOPERATION

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:loc="http://iserve.kmi.open.ac.uk/ontology/location.rdf#"
     xmlns:estate="http://iserve.kmi.open.ac.uk/ontology/estate.rdf#">
     <estate:Property rdf:about="#estate0">
            <estate:hasLocation rdf:resource="#loc0"/>
            <estate:hasMinPrice>100000</estate:hasMinPrice>
            <estate:hasMaxPrice>2000000</estate:hasMaxPrice>
     </estate:Property>
     <loc:Location rdf:about="#loc0">
            <loc:hasName>Milton Keynes</loc:hasName>
            <loc:hasPostcode>MK7 6AA</loc:hasPostcode>
     </loc:Location>
</rdf:RDF>
```

As can we seen, we let place name supersede postcode if both are provided. Any other information required to form a valid request for invoking the actual API can be extracted directly from service description or worked out in a similar manner. Once the Web API is invoked, the invocation response, in forms implemented by the Web API, will be lifted back to RDF format, if required, by running respective *liftingSchemaMapping*[17] scripts. In the given example, when the invocation succeeds and a 200 OK status is returned, a mapping to its RDF counterpart *http://www.w3.org/2006/http#200* will be returned as the final response status. A complete cycle of invocation is then finished. Invocation artifacts incurred from this service invocation, including execution status, inputs/outputs, are available for access through a RESTful API at *http://iserve-dev.kmi.open.ac.uk:8080/RestInvoke/service/c4e16ab6-3bad-47bb-b613-90ef78232e31/operation/SearchListingsOperation/data*

## VII. RELATED WORK

The work presented in this paper is in line with the efforts from the SWS community that target to raise the level of automation for service tasks through semantic descriptions. We are not the first to address the need for automating the task of service invocation. Previous work includes a Web-based tool, abbreviated as SPICES [12], which automates the process of consuming a Web service by making use of service semantic descriptions. Unlike our work that is especially focusing on Web APIs, SPICES supports both traditional WSDL services and RESTful ones and offers end-users the possibility of interacting with them. Reliant upon similar service description models for describing RESTful services, i.e. WSMO-Lite, hRESTS and MicroWSMO, the robustness and scalability of the tool is yet to be addressed due to the insufficient description capacities of the description models at that time. Besides, working only for APIs that follow strictly RESTful principles could not address the wide range of heterogeneities observed in the variety of Web APIs. With similar interests, the work in [13], aiming for automated invocation of RESTful and RPC-style services, presented an approach that draws the service's interface into a HTTP ontology, and use backward-chaining rules to translate between semantic service invocation instances and the HTTP messages passed to and from the service. This approach discards the widely recognized lowering and lifting mechanisms and works solely at ontology level. It is a plausible approach in the context of Semantic Web, but the tasks of writing translation rules as well as auxiliary ontologies can be difficult given that the information of Web APIs are usually described in HTML documents. Out of the realm of SWS, some of the issues related to the invocation of Web APIs have been studied in the context of service composition [3] and dynamic invocation [19]. This research is based on the introduction of the new HTTP binding in WSDL 2.0 as a promising approach by wrapping a RESTful service and then describing its interface using the WSDL language [8]. Semantic descriptions of services used in this work could apply to the then WSDL-described services. However, from a practical point of view, a wide adoption of

[17] Full text is omitted here. Please follow the link as given in service description for details.

such approach is yet to be seen [3]. In addition, WSDL is not directly capable of handling JSON format.

## VIII. CONCLUSIONS AND FUTURE WORK

This paper, targeting automated invocation of Web APIs, presents a framework that contains the extensions to the existing service description model in order to describe the heterogeneous particulars to enable their automated invocation and the OmniVoke framework, which works on semantic descriptions of Web APIs using the extended service model to automating the invocation of actual Web APIs. The OmniVoke framework, including API invocations and a view over their execution states as well as associated artifacts, is published through Web API interfaces, the form amenable for access and manipulation by Web data/applications.

Future work will be devoted to two main aspects following the vision previously presented in [20]. On the one hand we shall integrate the OmniVoke framework developed with a process engine in order to support the orchestration of semantically annotated Web APIs. Given the native support for RDF, the resulting orchestration engine shall thus also form the basis for developing Linked Data applications and mashups easily. On the other hand, we shall exploit the OmniVoke framework to provide generic means for exposing data behind legacy Web APIs as Linked Data by developing an additional layer able to transform directly HTTP requests for Linked Data resources into Web APIs invocations.

## IX. REFERENCES


[1] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating web APIs on the World Wide Web," In Proc. of the 8th European Conf. on Web Services (ECOWS), 2010.

[2] A. Duke, S. Stincic, J. Davies, F. Lecue, N. Mehandjiev, C. Pedrinaci, M. Maleshkova, J. Domingue, D. Liu, and G. Alvaro, "Telecommunication mashups using RESTful services," ServiceWave, 2010

[3] C. Pautasso, "RESTful web service composition with BPEL for REST," Data & Knowledge Engineering, vol. 68, issue 9, pp. 851-866, 2009.

[4] R. Fielding, "Architectural styles and the design of network-based software architectures," PhD thesis, University of California, Irvine, 2000.

[5] M. J. Hadley, "Web Application Description Language (WADL)," http://wadl.dev.java.net/, 2006.

[6] K. Gomadam, A. Ranabahu, M. Nagarajan, A. Sheth, and K. Verma, "A faceted classification based approach to search and rank web APIs," In Proceedings of the International Conference on Web Services, 2008.

[7] J. Kopecky, K. Gomadam, and T.Vitvar, "hRESTS: an HTML microformat for describing RESTful Web services," In Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence, 2008.

[8] E. Chinthaka, "REST and web services in WSDL 2.0," http://www.ibm.com/developerworks/webservices/library/ws-rest1/, 2007.

[9] A.P. Sheth, K. Gomadam, and J. Lathem, "SA-REST: semantically interoperable and easier-to-use services and mashups," IEEE Internet Computing 11(6) pp. 91–94, 2007.

[10] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Semantic annotation of Web APIs with SWEET," 6th Workshop on Scripting and Development for the Semantic Web at Extended Semantic Web Conference, 2010.

[11] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue, "iServe: a linked services publishing platform," Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference, 2010.

[12] G. Álvaro, I. Martínez, J. Gómez, F. Lecue, C. Pedrinaci, M. Villa, and G. diMatteo, "Using SPICES for a better service consumption," Poster at the 6th Extended Semantic Web Conference, 2010.

[13] D. Lambert, and J. Domingue, "Photorealistic semantic web service groundings: unifying RESTful and XML-RPC groundings using rules, with an application to Flickr," In the 4th International Web Rule Symposium (RULEML), 2010.

[14] L. Richardson, and S. Ruby, "RESTful web services," O'Reilly, May 2007.

[15] J. Farrell, and H. Lausen, "Semantic annotations for WSDL and XML schema," http://www.w3.org/TR/sawsdl/ (January 2007), W3C Candidate Recommendation, 26 January 2007.

[16] World-Wide Web Consortium, "HTTP Vocabulary in RDF 1.0," Working Draft May 2011, available at http://www.w3.org/TR/HTTP-in-RDF10/M.

[17] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," Int. Journal on Semantic Web and Information Systems (IJSWIS), 2009.

[18] W. Akhtar, J. Kopecky´, T. Krennwallner, A. Polleres, "XSPARQL: traveling between the XML and RDF worlds and avoiding the XSLT pilgrimage," In Proceedings of the 5th European Semantic Web Conference (ESWC2008). pp. 432– 447. Springer-Verlag, 2008.

[19] Y. Chen, J. Li, Y. Lv, H. Qin, L. Zhang, "DRWSC-to simplify dynamic invocation for RESTful web services," ICSES, IEEE Press, 2010.

[20] C. Pedrinaci, and J. Domingue, "Toward the next wave of services: linked services for the web of data," Journal of Universal Computer Science, 2010.