ServiceBase: A Web 2.0 Service-Oriented Backend Programming Platform

Moshe Chai Barukh[1] Boualem Benatallah[1]

[1] University of New South Wales, Australia
{mosheb,boualem}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201212
May 2012

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

With the growing success of modern web-technology and the service-oriented paradigm, the Internet continues to flourish with a large and interesting plethora of web-services. What started as the supply of simple reusable application functions has rapidly evolved to far more powerful service-enabled technologies, such as Software, Platform and Infrastructure –as-a-Service. Although, while these services may expose their functionality in the form of an API, often integrating them in everyday application-development still remains a manual, complex and repetitive task. Such as, the technical knowledge required in connecting to services, as well as the ability in interpreting and manipulating run-time data to drive the application-logic. More importantly, applications need to be constantly maintained to keep up with the ongoing evolutions in the service-providers' API. In this paper, we propose to address these challenges by abstracting and simplifying access to web-services, where a common programming interface can be used to search, explore and also interact with services. A framework is also proposed for decomposing and mapping raw service-messages into more common data-constructs, thus making interpreting, manipulating and chaining services further simplified despite their underlying heterogeneity. Furthermore, unlike existing systems, we implement a Web2.0-oriented ServiceBus platform that fosters a community between service-curators, service-consumers and end-users. In this manner, knowledge about services can be incrementally registered, shared and reused by distributed application developers.

1 Introduction

The inception of the Service Oriented Architecture (SOA) paradigm has significantly helped shape the framework for modern software engineering, and in a large way assisted in solving some of the major challenges faced by early application developers. The fundamental principles which were founded upon the notions of loosely coupled, location transparent and protocol independent provisioning of services, made it possible to re-use existing application-logic despite their underlying heterogeneity. More significantly the advent of Web-Services technology has even further empowered these capabilities, and when coupled with other advances on the Internet, there is certainly the potential to derive yet far more powerful systems. For example, Web2.0 and collaboration technologies provide a common interaction and sharing infrastructure often supporting concurrent and real-time interactions. As did crowd-sourcing platforms that provide the means for leveraging the wisdom of large groups and communities to work independently to solve problems in the same way open-source did for software development. Inevitably, the Internet has thus presented us with a large and interesting plethora of web-services, which essentially expose access to rich resources via a service-API. However, while the primary goal of such Internet computing platforms has been to support development and deployment of applications, it is evident from recent research studies [1,2,3,4] there lies significant challenges that limit the wide-scale adoption of rich web-services in everyday application development. Upon careful analysis of these studies, which collectively canvas over a wide variety of settings, we may decompose the challenges associated with using APIs into three main cross-cutting concerns: (i) *Where* can a suitable API be found? (ii) *What* does the API do? (iii) *How* can it be integrated into an application?

If we visit early SOAP-oriented web-services technology, standards such as WSDL were defined as the accepted description language, but soon proved to be too verbose to be effectively understood by humans. Interestingly, despite the voluminous and still growing family of WS-* standards defined in support of WSDL, it is evident that the more recent yet standard-less RESTful services has by far outweighed SOAP service offerings. For example, in the last three-years, the reported number of SOAP services have risen from around 250 to just over 500, while in contrast RESTful services have risen from around 1,200 to now over 2,800 services reported, [6].

The reasons for this are clear: the value of a more easier understood and human-friendly API is by far better favoured. However, since the lack of standards often mean no automated support, this has consequently meant that programming with APIs still remain quite a considerable manual and time-consuming task. For example, consider the scenario of a Java-function implementation to calculate the percentage of user-contributions of a Google Document. (In fact, we specifically choose this scenario, as later in the paper, with the results of our work we show how this reasonably complex program can be implemented in only 26 lines of code). However, when using traditional techniques, we may identify the following main complexities: (i) This would require writing code to directly access Google's RESTful API, where concrete knowledge of the endpoint addresses, operation-types, and parameter names must be understood. More so, since these calls require authentication, code would be required to handle tasks such as signing requests with an appropriate signature encryption; (ii) Data returned from service-calls are usually raw, in this case XML, and must be both parsed in order to find the required information, and transformed into numeric variables so that the appropriate calculations can be applied to them; (iii) In the cases above, while support may be provided by third-party libraries, this inevitably creates additional dependencies on the overall project; (iv) Finally, the fact that service APIs frequently change, both with respect to endpoints, parameter-names, or even the structure, format or semantics of the data-messages, this therefore require that applications are constantly updated in order to keep up with the constant evolutions.

In order address these challenges we therefore present *ServiceBase*, which provides a common programming interface for both the curation and access to web-services. A common-interface means that applications that integrate services can be heavily simplified by using more high-level operations that automate a more concrete and detailed set of methods behind the scenes. In order to make this effective, a common model needs to be defined for both: the design-time components (i.e. representations of various service-types); as well as the run-time components (i.e. the actual data obtained from services). In this manner, applications can be fully decoupled from the underlying services that they access. Then, as for the middleware that mediates between applications

and services, we propose that this can be architected in the form of a Web2.0-oriented service-bus platform. A key component of the bus will also be the service-repository that maintains the meta-information about available services. However since it is clear that static repositories are only of limited value, we therefore draw inspiration from the Web2.0 paradigm – where just as open-communities such as Freebase and Wikipedia dissipate encylopediatic information in the form of user-contributed content, similarly technical knowledge about services could be both populated and shared amongst other developers who may require almost precisely the same logic. More specifically, to support the above requirements, this work offers the following main contributions:

- ***A Unified Service Representation Model,* (Section 2)**. Such a model is needed in order to deal with the heterogeneity of service-types and their representations. For example, services differ in their protocol, security policy, access-method, and data-representation of their input and output messages. Our model aims to unify and abstract the low-level detail into a more user-oriented high-level representation where we have targeted both the design-time and run-time models. We therefore propose that users can recognise and subsequently invoke services using simple string names, instead of having to remember the exact low-level endpoint addresses, precise operation names and parameter values. Using such a design also helps solve the change-management concerns, since if concrete properties such as endpoints or operation-names changes, they will not affect the users' applications. This also means that searching for services would return more meaningful results to the user, and to further enhance this we also employ tags to help provide semantic-based user-categorisation of services. The notion of Service-templates is also proposed as a means of providing functionality-based categorisation of similar services. More importantly, service instances that inherit a service-template can adopt common operation names and message formats, and therefore offer a structurally uniform interface amongst services in the same category.
- ***Framework for Mapping between Raw-Message formats and Common Data-Structures,* (Section 3).** To augment a further level of simplicity and flexibility, we provide a framework for mapping between native service message formats and more common data-structures. The means that input and/or output messages of services can be decomposed and represented as various types of atomic (string, numeric, binary, etc.) or complex (list, tuple) fields, thus making service-messages easier to interpret and manipulate. The mapping framework we provide works both ways, whereby fields can be mapped to input-messages; and likewise output messages can be mapped back into appropriate output fields. For example, a particular benefit of this could be that services with similar operations can be modelled to have a uniform interface despite their underlying heterogeneity. Another example being the process of chaining several services could be simplified, as input and output messages could be modelled in a way that they are compatible with one another, or which may only require minimum re-transformation.
- ***The ServiceBase System, with a service-bus middleware to mediate between heterogeneous services and application-level users,* (Section 4).** Empowered by both the unified service model and the mapping framework, the service-bus is then able to fully decouple service from the applications that are using them. A common programming interface is therefore provided to perform functions such as searching/updating services, invoking services, subscribing to feeds, performing authentication, parsing messages, etc. The service-bus also takes the role to interpret high-level user-operations and transform them into a more concrete set of calls. Moreover, the service-bus creates a community hub between curators and consumers, such that knowledge about service-models and mappings could be incrementally added and re-used.

This paper is therefore organised as follows: In Section 2 we present our unified web-services representation model, that simplifies access to web-services by abstracting a high-level user-oriented model from the low-level details of heterogeneous service APIs. In Section 3 we present a framework for decomposing raw service-messages into various common data-structures. We then present the *ServiceBase* system-architecture and implementation in Section 4. In Section 5 we present examples of typical application-development use-case scenarios, where we highlight the main features and potentials available using *ServiceBase*. In Section 6 we evaluate our system and proposed approach for simplified web-service access using a user-case study. We then examine and discuss related work in Section 7, where we finally conclude with a summarisation and short discussion in Section 8.

2 Unified Web-Services Representation Model

At the most basic level, standardizations such as WSDL have provided an agreed means for describing low-level APIs, and much less formally, the same might be true for the documentation of RESTful APIs. However, it is clear most of these have been focused primarily for the purpose of service-description and discovery, where the emphasis on simplifying service-execution is almost neglected, [2,5,7].

In this section, rather than treating services as isolated resources, we describe a unified representation model, where web-services can be: (i) described and registered onto a repository; (ii) discovered and incrementally evolve as service descriptions or data-representations change; (iii) can be called upon for purpose of executing and interacting with these service, therefore simplifying the utilisation and maintenance of services in application development.

2.1 Design Overview

Figure 1 illustrates a summarised view our unified web-services representation model, described in UML. The model encapsulates all the necessary information about a particular web-service API that is required for both description-level and run-time (or execution) level processing. This includes, functional information such as its various operations, events, interacting-protocol, authentication information, security access-policies, etc., but also other non-functional information such as textual tags for purpose of categorisation, etc.

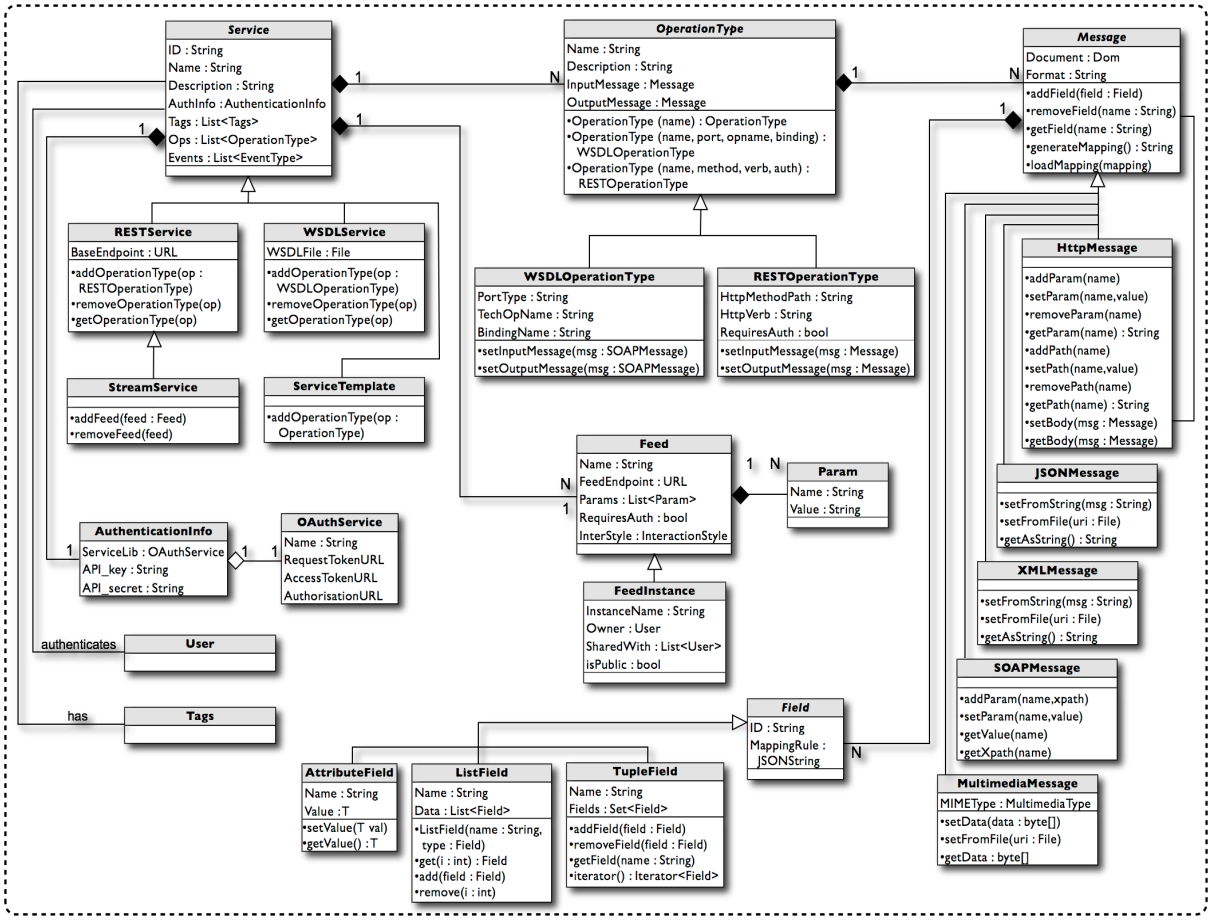

Figure 1. Summarised view of the Unified Service-Representation Model

2.2 Services

The class *Service* is an abstract superclass for all specialised Service types being implemented. In our current model, we support three main service types. This includes the two most widely accepted service representation-types, REST and WSDL, but also *StreamServices*, such as ATOM and RSS feeds. The model of an API for a given service can be expressed in three layers: (i) service-layer; (ii) operation-type layer; and (iii) message layer. These three entities represent the most minimalistic information needed for a user to interact with these services, thereby simplifying the method for service-access, in contrast to having to be aware of the complete technical details of a service.

A *RESTService* extends the class Service, and requires that a BaseEndpoint address be specified. This address is the common part of the URL used in making calls to RESTful methods. In the case of *WSDLServices*, the only meta-information required is the WSDL source file – the system then performs run-time parsing of the WSDL file in order to obtain necessary information about the service. *StreamServices* further extends *RESTServices*; we found this model appropriate since a very vast majority of feed-services also specifies a RESTful interface in conjunction. Typical examples of these being: Twitter, Facebook, Flickr, etc. A *StreamService* differs from standard RESTful services by allowing *Feeds* to be specified, which then allows feed-entries to be monitored.

An important challenge is dealing with access to secured service; the widely used protocol for this is OAuth [8]. Our model defines the class *AuthenticationInfo* in order to specify this necessary information, which is then linked to a particular Service. Services, which define operations that require authentication (for example getting comments of a Facebook wall), will need to have this information defined. End-users can then authorise the service to authenticate on their behalf, which only needs to be done once since the secure access-keys are retained for each user, and will expire only until the access-rights have specifically been revoked.

In addition to the above, we also introduce the notion of *ServiceTemplate*s which essentially provides a means to specify a common pre-defined structure for services of a particularly category. This may be particularly useful for services that offer similar functionality, whereby not only does this provide a way for grouping these similar services, but also service-instances that adopt a common template can be designed to inherit similar operation-names, message-structures and formats, etc. This therefore provides an addition level of uniformity from the perspective of the end-user despite the underlying technical heterogeneity amongst these different services. Likewise, from the perspective of users who add new services an added level of support is provided by the re-use of templates to assist in formulating service-descriptions.

2.3 OperationType

From a technical standpoint, the notion of a service-operation might take upon different meanings for different service types. For example, in the case of RESTful HTTP services, while there are generally four basic operation-types: GET, PUT, POST and DELETE; often what is more relevant to the end-user are the various methods that are available. For example, searching for photos on Flickr, requires making a `GET` call to the resource located at `http://api.flickr.com/services/rest/?method=flickr.photos.search`. In this manner, we could recognise `"Flickr"` as a service, while `"flickr.photos.search"`, as one possible user-operation.

WSDL services are slightly different in that operations are generally expressed as an RPC call. Although, each operation can only be recognised in relation to its associated port-type, and subsequently, each port-type is recognised in relation to its associated binding-name and endpoint.

However, we propose that in both cases service-access can be simplified by abstracting the low-level details from the end-user. For example, it would be much more convenient to express a call to Flickr as `"/Flickr/getPhotos"`. Likewise, if we for example had a WSDL service used to manage flight bookings, we could simply express a call to the service as `"/Flights/getBooking"`, rather than having to specify all of its associated information. Simplification is also further achieved since we can attach more meaningful names to service and operation names, rather than having to adhere to the more technical names. In our model, this is supported by the abstract superclass *OperationType*. Although since the low-level details of operations may differ between different service-types, we have specialised into various sub-classes, as shown in Figure 1.

2.4 Feeds

Our model also supports the detection of new feed-entries that are generated from feed-services, such as RSS or ATOM. As mentioned, we call such services *StreamServices*, and enable feed-sources to be added to the service definitions using the class *Feed*. The meta-data that is required to be specified here includes the endpoint of the feed-source, the interaction-style used to read feeds, as well as any instance parameters. There are three main interaction-styles supported: polling (i.e. periodic data pull at a predefined interval); streaming (i.e. an open call that allows data to be pushed to the caller); publish-subscribe (i.e. this involves registering to a hub that actively sends data only when new content is available).

Although, while the class *Feed* enable to define the basic meta-data required in order to interact with these feed-based services, its definition still remains abstract, in the sense that they may not point to any specific feed-source. For this reason it is necessary to define instantiated versions of abstract feed-sources, using the class *FeedInstance*. For purpose of example, consider that we would like to monitor when new posts are published to a discussion on Flickr. In this case we consider the abstract *Feed* defined as `"FlickrDicussion"`, which specifies the base-endpoint to this source, in this case: `http://api.flickr.com/services/feeds/groups_discuss.gne`. Although we also identify that this endpoint has a parameter, called `"groupID"`, that pertains to the group ID of the particular discussion feed. These parameters can then be registered in the abstract *Feed* object and subsequently instantiated in the *FeedInstance* object.

When an abstract *Feed* object has been registered, it can then be retrieved and reused by others at any time, in order to create several customised instances of this without having to redefine duplications of the common, low-level details. Examples of this could be discussion groups on: `LosAngelesDiscussion`; `OsXDiscussion`, etc., where in each case the instance-specific `"groupID"` parameter needs to be set. Moreover, as certain feed-sources may require authentication, the owner can appropriately restrict access to specific feed-instances by allowing to only share access with specific users. Alternatively, it could also just be defined as public-view.

2.5 Messages

Messages represent the various serialisation-types for data that is associated with services. This applies to both, incoming and outgoing messages. All message-types that we support are specialisations of the superclass *Message*, which internally stores all message data in the form of a DOM object.

The message objects provided here can be used in two main ways: Firstly at design-time, by service curators when defining a service API which can then be registered onto the service-bus; and secondly at execution-time, by application-developers when passing in messages in order to communicate with the services. In the first case, it is mainly the message 'type' which allows the service execution-engine to understand what type the input and output messages should be formatted in. Although in some cases as well, the content of the message itself that is specified by the curator could also be important, as it may enable, for example, certain parameters to be predefined, or act as a template from which more concrete messages can be built in order to specifically interact with the services. Thus in the second case, messages objects are also the key abstraction used for data interchange between services and the caller. In particular we support the following message-types:

An *HttpMessage* encapsulates the user-specified attributes of an HTTP call, as well as any payload that could be attached to the http-body. In the simplest case this involves the http-parameter name and/or value pairs. For example, `"?id=613&page_size=10"`. Although, we have also identified how dynamic endpoints can also be formulated using parameterised path values. For example, `"<base-endpoint>/questions/{id}/related"`. In this case there is a parametric path value, referenced here as `"{id}"`. The body of an http-message can itself accept a valid Message object, such as an XML, JSON or binary File payload, or simply left empty when none is required.

An *XMLMessage* is relatively straightforward, in that it is simply specified by the user either by a `STRING` input, or loaded directly from a `FILE URI`. Although, given that Messages are inherently stored as a `DOM` object, this means xml-messages can be manipulated using the same features and methods available as provided by the DOM API.

A *JSONMessage* is also relatively straightforward, in that it must also be specified either by a by a `STRING` input, or loaded directly from a `FILE URI`. Additionally, we provide inbuilt algorithms in order to serialise and convert between XML and JSON formats, when this is needed.

A *MultimediaMessage* represents any other type of Internet media file [9] that is not already directly supported by the message formats itemised above. For example, this could refer to image or audio files, or application files, such as PDF or DOC files, etc. These media-type files are loaded either directly from a `byte array`, or specified from a `FILE URI`. In which case, it is also required that an appropriate `MIMEType` is specified so that it can be known what type of media the file represents.

In this case of *SOAPMessages,* while it's body is inherently an XML document, we can still provide added support since we know the schema of the message-structure, provided as part of the WSDL file specified in the associated service-object. We define parameters of a *SOAPMessage* as the triple `<name, xpath, value?>`. Where: the *`name`* can represent a simple string name that can be used to reference the parameter; the *`xpath`* value is the path-query used in order to reach the respective data field; and the *`value`* field can assign the appropriate data to that field.

Finally, in addition to the above, Message *Fields* help to provide a further level of abstraction when representing services. By default, all services when modelled will need to specify an input and output message format, such as those which has been itemised above, each of which will directly reflect the native formats that are defined as part of the service. For example, a particular Twitter service operation that takes an Http input and returns JSON will need to define the appropriate messages to correspond to this. However, *Message-Fields* can be provided to further abstract this layer of heterogeneity amongst services, such that services can be invoked and consumed via a set of message-fields, (based upon a common set of field-types), rather than having to deal with the native message formats. In this manner, a further level of simplicity and flexibility can be made available when interpreting and manipulating message data.

3 Decomposing and Mapping Messages into Fields and Vice-versa

As mentioned previously, message-fields provide an additional level of abstraction for modelling messages, where the key motivation has been to further simplify working with service message data in the way that they are interpreted, manipulated and re-used. In this section, we therefore present the framework proposed for decomposing messages into various fields, as well as a the definition of a simple language used in order to specify the mapping between raw-message types and the more higher-level message field types. To further consolidate these concepts, we demonstrate this via a motivating example, which is defined latter in this section.

3.1 Message-Field Types

We propose three message field-types, where it appears these types can be employed either directly or in combination with one another, to appropriately abstract raw message data made available by services. We first support one atomic field-type, called *AttributeField*, which can be instantiated for common primitive data-types applicable for web-services, such as, *string*, *integer*, *date*, as well as *binary-array* to handle media files. Attribute fields are relatively simple and define a name and value pair. We then support two complex types, which may itself contain other atomic or complex fields nested within them. The complex type *List* represents an indefinite, ordered list of field-elements where each item must be of the same type. In order to define a List field, it is therefore required that the field-type be specified for which this list will be associated to. The second type of complex field supported is called *Tuple*, which represents a finite collection of fields of arbitrary field-type. The structure of the Tuple-field is quite similar to what the notion of a Struct or Class objects provides in an object-oriented programming environment. This means that the structure of the collection must be defined at 'design-time' and thus this will be known to contain a finite set of field-objects. This is in contrast to the *List* field-type, which is rather designed to handle an unknown collection of items often only obtained during the 'run-time'.

3.2 Mapping-Rules Structure

Once a set of fields have been defined and associated to a particular message, the next step involves defining the information needed in order to map between instances of the raw message-type and the corresponding field-object structure. We propose that this information can be specified in the form of mapping-rules that binds together the raw-message with the various fields. We define a mapping rule specification to be a block of JSON that allow us to enter formulas needed in order to document the mapping. We have specifically chosen to use JSON as the representation format particularly due to its characteristic nature being well suited to represent the organisation of name/value pairs and nested objects.

We define each Field-type to have a unique mapping-rule structure associated to it, which we have subsequently described in Listing 1 below. Moreover, in the case that we have complex field-type, such as for example a *Tuple<String,String,Integer>*, the corresponding mapping rule specification that would be generated could be represent a nested JSON object composed of several of the individual blocks.

Since atomic fields are defined effectively as a name-value pair, we define the mapping-rule block to be a JSON object that consists of one label named `value`. A label means that a formula will need to be associated to this attribute. For example, a formula might specify how a data-value of a field can be located from within the raw-message. In the case of List-field, the mapping-rule specification is defined to have two parts: the first which may be optional in some cases and specifies a `nodepath` expression; and the second which nests the mapping of the field-type for which this list has been defined. The concept of a node-path expression has been defined in the work presented at [10], and is only necessary when dealing with raw messages schema that are hierarchically organised XML/JSON files. Although since a very large number of services use XML/JSON as the message-exchange format in some way or another, we have found it quite necessary to provide explicit support for this as part of the mapping architecture. The value defines the path to the node (i.e. sub-tree) that are to be considered distinct elements of the list. Finally the mapping-rule specification for the Tuple-field is simply defined as an empty object, where it could potentially cascade or nest any number of sub-fields.

(A)	(B)	(C)

“attribute” : {
  “value” : “*formula*”
}
	“list” : {
	  “nodepath” : “*formula*”,
	  (nested field)
	}
		“tuple” : {
		  (nested field/s)
		}

Listing 1. Mapping-rule structures associated with message-field types, where: (A) Attribute-field; (B) List-field; and (C) Collection-field

3.3 Mapping Specification Language

We described above how the structure of mapping-rules might be defined for a given message-field, or a nested collection of message-fields. In this section we will show how the information for these rules can be specified:

The specification language can be broken up into two main categories of operators: (i) *Selectors,* which provide an explicit means for identifying a particular element from the raw message object; and (ii) *IOoperators*, which allows us to either read or write the data to or from the source and destination objects. More precisely, a mapping that is applied to an input message-field would be concerned about *writing* (or *appending*) data to the raw-message object from the data held in fields, while a mapping that is applied to an output message-field would be concerned about *reading* data from the raw-message object and populating to the fields. It seems to be unnecessary to support other functions such as manipulators, as this all can be adopted from within the selector languages, such as XPath or JSONPath.

```
Selectors   => xpath(expr)          IOoperators   => read(selector)
            => jsonpath(expr)                     => write(selector)
            => httpparam(expr)                    => writeappend(selector,string?)
            => httppath(expr)
            => httpbody()
            => httpurl()
```

3.4 Motivating Scenario

As a motivating example, we consider an application that provides visualisation of data, where data can be obtained from various database web-services. The application in this case utilises *Google Image Chart API* [11] in order to render visualisations of the data, which for example may be in the form of bar graphs, scatter-plots, pie charts, etc. Hosting data on web-service platforms has in fact gained considerable momentum in recent times, particularly due to its ability to deal with huge amounts of data in a relatively efficient manner, [12]. The requirements of this application is to therefore be able to read data from various services and in some cases provide visualisations that combine data from multiple sources, such as for the purpose of providing comparisons, etc. In this example, we consider two specific such data-services, namely *OrientDB* [13] and *Amazon's SimpleDB* [14].

The traditional development approach in implementing this application would typically result in having to connect separately to each different data-services; this is because even though the data ultimately obtained might be similar, each data-service provides a different interface for accessing their system. In contrast however we therefore show how message-fields can rather be employed to provide a common-level of abstraction amongst heterogeneous yet similar services, despite their different data-formats and/or message-structures. For instance, in this case, database services could be abstracted to a set of operations such as: *put*, *get*, *read*, *delete*, etc. If we choose the common "get" operation as an example, we can illustrate what the corresponding input and output message-fields would be as show in Listing 2, which could also be expressed as *get(sqlQuery) returns ResultSet*.

```
Attribute<String>      [SQLQuery]
```

(A)

```
List<•>                          [ResultSet]
  List<•>                        [Row]
    Tuple<•,•>                   [Cell]
      Attribute<String>          [CellName]
      Attribute<String>          [CellValue]
```

(B)

Listing 2. Common Message-Fields for DB-services "get" operation, where: (A) Input-field; and (B) Output-field

Moreover, we can now demonstrate the mapping logic for these data-services, as shown in Figures 2 and 3, for *SimpleDB* and *OrientDB* respectively. We can also describe the data-flow in the following stages: (i) Firstly, the input message-fields need to be mapped to an appropriate raw input-message; (ii) Once this is obtained the service can be invoked using the service-bus; (iii) Then in reverse the resulting raw output-message needs to be mapped back into the appropriate output message-fields.

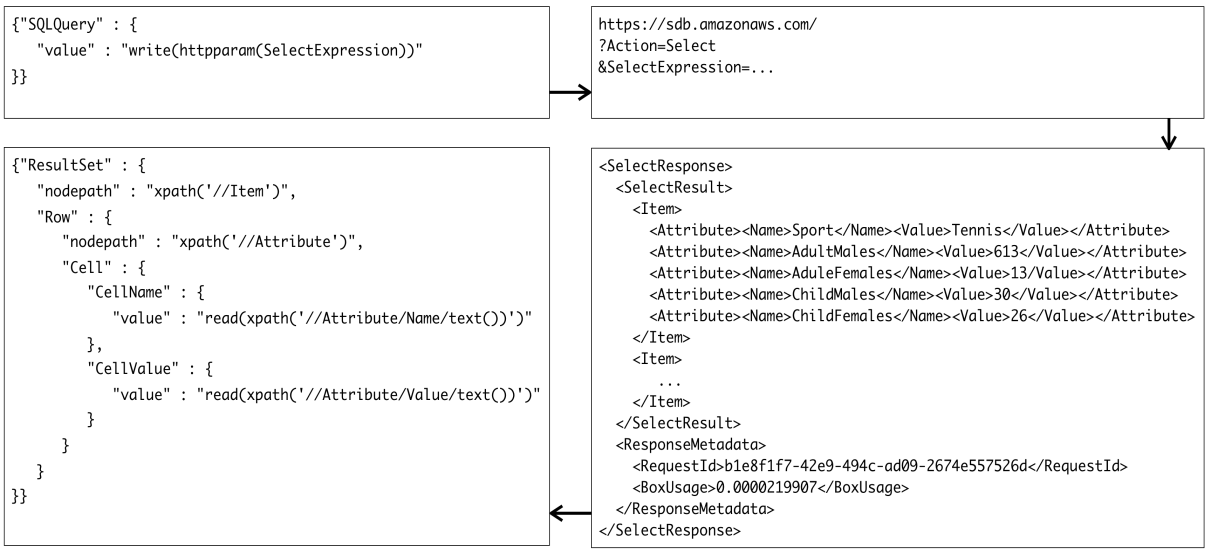

Figure 2. Mapping Input and Output Message-Fields for "get" operation of Amazon's SimpledDB

Figure 3. Mapping Input and Output Message-Fields for "get" operation of OrientDB

As another example, we may similarly demonstrate how the Google Image Chart API can also be abstracted, as shown in Figure 4 below. For instance, we may define an operation to create a pie charts, which could be expressed as *createPieChart(size, chartdata) returns Image*. In this case we have defined two input fields: The field *size* is defined as a simple String Attribute, while the field *chartdata* is defined as a List of a Tuple `segment`, which itself contains two string Attributes; one that refers to the `label` of the segment, while the other specifies the numeric `value` of the segment. The output field *Image* is defined as an Attribute of byte-array, in order store the binary data of the PNG image file returned. In fact, we may also note that the *chartdata* field is quite similar in structure to the nested-field called `"Row"` which we defined earlier for database services. Therefore, this further shows as an example the additional benefit provided by message-fields in its ability to assist in increased compatibility when chaining various services.

Figure 4. Mapping Input and Output Message-Fields for "createPieChart" operation of Google Image Chart API

4 ServiceBase System Architecture & Implementation

Figure 5 illustrates the system design and interaction of the main components of the *ServiceBase* system. Our architecture uniquely demonstrates the Web2.0-inspired ecosystem creating a community between service-curators, service-consumers and end-users. In this manner, not only has the primary goal been to simplify access to web-service and encourage increased wide-scale usability, our goal has also been to provide an infrastructure where service-knowledge such as API-models and mappings can be incremental enriched, shared and re-used in this community ecosystem. In order to provide simplification in dealing with web-services, we have adopted the unified service-representation model that we presented above. We have then implemented this model exposing a set of APIs for programmatic access to the service-bus, available as both a Java-client library and RESTful service bundle.

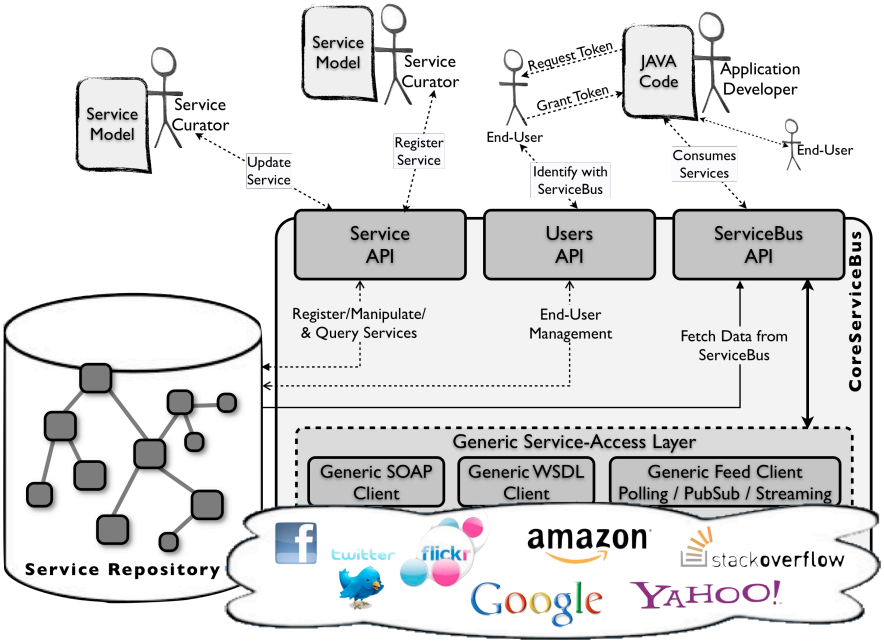

Figure 5. ServiceBase System Architecture

4.1 ServiceBase APIs

The programmatic interface to *ServiceBase* offers the following APIs:

This *ServiceAPI* would primarily be used by service-curators in order to *register* new services into the service-repository, but also *update* and *delete* service-definitions that have already been registered. The API also provides *read* access and *search* functionality of registered service, so that service-consumers can explore and retrieve service objects.

The *ServiceBusAPI* would primarily be used by service-consumers, i.e. application-developers, as the main gateway in order to interact with registered services. In particular, the API provides methods for simplified invocation of services, subscribing to services, querying (i.e. pull) of events, listening (i.e. asynchronous callback push) of events, authenticating services, etc.

The *UsersAPI* provide a means for end-users to identify themselves with the service-bus. When a user is identified and registered in the service-bus, they are then able to assign/revoke authentication privileges to various services that they choose. In this manner, application that are written on-top of services that require access to secure resources (for example, invoking an operation to get the specified user's collection of Google Docs), can then be further simplified, as the entire logic for handling the secure calls is managed by the service-

bus, rather than the application developer. At the application-level, secure calls can be processed on behalf of a specific user simply by requesting from the user, or having shared an *access-key*, (done via OAuth, [8]), which is then passed into the invocation method. The *access-key* shared by the user can be *restrictive* (i.e. only allows the specified application access to certain services), or it may be *non-restrictive* (in which case the user allows the specified application to access on their behalf all services that they have authenticated).

4.2 Service-Model Example

The service-curator's primary role is to register and maintain service entities on the ServiceBus, such that it can be explored and used by other service-consumers in the process of application development. As an example, we will show how the *Google Docs* Service API, [15] can be registered, where for purpose of simplicity we have chosen 3 operation-types and 1 feed-source as an example.

This process may be best demonstrated in a set of stages: The first step involves employing the *unified service-representation model*, in order to model the service and it's API, whereby the code as shown at Listing 3 would be produced. At this stage we omit to introduce any message-fields, and mapping information, in which case the code shown below illustrates the most basic model for representing and registering services. At the next step, we will then move to show how message-fields could also be defined in order to further abstract the technicalities of working with services.

```
1.  //Define OperationType1: SearchDocuments
2.  HttpMessage msg_in1 = new HttpMessage();
3.  msg_in1.addParam("q");

4.  XMLMessage msg_out1 = new XMLMessage();
5.  msg_out1.setDocumentFromFile("samples/documentlist.xml");

6.  OperationType otype1 = new OperationType("SearchDocuments");
7.  otype1.setHttpMethod("get");
8.  otype1.setHttpPath("/default/private/full");
9.  otype1.setInputMessage(http_msg1);
10. otype1.setOutputMessage(xml_message1);
11. otype1.setRequiresAuth(true);

12. //Define OperationType2: GetSharedUsers
13. HttpMessage msg_in2 = new HttpMessage();
14. msg_in2.addPath("resource_id");

15. XMLMessage msg_out2 = new XMLMessage();
16. msg_out2.setDocumentFromFile("samples/sharedusers.xml");

17. OperationType otype2 = new OperationType("GetSharedUsers");
18. otype2.setHttpMethod("get");
19. otype2.setHttpPath("/default/private/full/{resource_id}/acl");
20. otype2.setInputMessage(http_msg2);
21. otype2.setOutputMessage(xml_message2);
22. otype2.setRequiresAuth(true);

23. //Define OperationType3: GetDocumentRevisions
24. HttpMessage msg_in3 = new HttpMessage();
25. msg_in3.addPath("resource_id");

26. XMLMessage msg_out3 = new XMLMessage();
27. msg_out3.setDocumentFromFile("samples/docrevisions.xml");

28. OperationType otype3 = new OperationType("GetDocumentRevisions");
29. otype3.setHttpMethod("get");
30. otype3.setHttpPath("/default/private/full/{resource_id}/revisions");
31. otype3.setInputMessage(http_msg3);
32. otype3.setOutputMessage(xml_message3);
33. otype3.setRequiresAuth(true);

34. //Define AbstractFeed1: DetectChanges
35. Feed feed1 = new Feed("ActivityFeed");
36. feed1.setFeedSourceEndpoint("https://docs.google.com/feeds/
                                {user_id}/private/changes");
37. feed1.addPath("user_id");
38. feed1.setInteractionStyle(InteractionStyle.POLLING);
```

```
39. //Define: Authentication Information:
40. AuthenticationInformation authInfo = new AuthenticationInformation(
        OAuthServices.GOOGLEDOCS,     //ServiceLibrary
        “anonymous”,                  //API-key
        “anonymous”);                 //API-secret
```

Listing 3. Example Model of Google Docs Web-Service API

The next step involves defining message-fields that can be associated with the input and output messages of services, together with the appropriate mapping rules that define the information needed in order to transform between raw messages and message-fields. As we mentioned the benefit of this means any remaining technicalities of working with the native-message structure of services can be masked and replaced with more familiar message-fields. In reality this step could be considered optional, such that if omitted (as shown in the code above), the service-bus still enables interacting with services using native message types. Message-fields are thus provided as a further level of abstraction. In order to demonstrate this we show at Listing 4 below how one of the operation-types we defined earlier, namely `/GoogleDocs/SearchDocuments` could be decomposed into appropriate input and output message-fields. The code below would typically be inserted at Line 3 and 5 into the code shown earlier, where a similar definition could also be specified for any of the other operations. We also note in order to further simplify the process associated with writing mapping-rules, a template-file can be generated by calling `generateMapping()` on the appropriate message-objects (the output of which is shown in bold-text at Listing 4(b) and (c), whereby the rest could then be filled-in by the curator).

```
1.  AttribteField<String> search_keys = new AttributeField<String>(“Search_Keys”);
2.  msg_in1.addField(search_keys);
3.  msg_in1.loadMapping(“searchdocs_in_map.json”);
    ...
4.  TupleField document = new TupleField(“Document”);
5.  document.addField(new AttributeField<String>, “resource_id”);
6.  document.addField(new AttributeField<String>, “title”);
7.  document.addField(new AttributeField<String>, “author_email”);
8.  document.addField(new AttributeField<String>, “last_modified”);
9.  ListField document_list = new ListField(“DocumentList”, document);
10. msg_out1.addField(document_list);
11. msg_out1.loadMapping(“searchdocs_out_map.json”);
```

Listing 4(a). Defining Message-Fields for the operation /GoogleDocs/searchDocuments

```
1. {"Search_Keys" : {
2.    "value" : "write(httpparam(q))"
3. }}
```

Listing 4(b). Contents of “*searchdocs_in_map.json*”

```
1. {"DocumentList" : {
2.   "nodepath" : "xpath('/Document/entry/')",
3.   "Document" : {
4.      "resource_id" : {
5.         "value" : "read(xpath('/Document/entry/resourceId'))"
6.      },
7.      "title" : {
8.         "value" : "read(xpath('/Document/entry/title)')"
9.      },
10.     "author_email" : {
11.        "value" : "read(xpath('/Document/entry/author/email'))"
12.     },
13.     "last_modified" : {
14.        "value" : "read(xpath('/Document/entry/lastModifiedBy'))"
15.     }
16.  }
17.}}
```

Listing 4(c). Contents of “*searchdocs_out_map.json*”

Finally a service-object can be created and subsequently registered onto the service-bus, as shown at Listing 5 below. Putting all the code above formulates what we may refer to as the complete *service-model*, which when registered defines the appropriate meta-information used to both describe and execute services. Although it is important to note that a particular service-model may never be complete, in the sense that it can be retrieved and incrementally evolve by other service-curators at different times and often continuously.

```
1. //Define: GoogleDocs Service Object Definition
2. StreamService googleDocs = new StreamService("GoogleDocs",
                                                 "https://docs.google.com/feeds");
3. googleDocs.addOperationType(otype1);
4. googleDocs.addOperationType(otype2);
5. googleDocs.addOperationType(otype3);
6. googleDocs.addFeedSource(feed1);
7. googleDocs.setAuthInfo(authInfo);

8. //Finally, register the service on the bus:
9. String service_id = ServiceAPI.registerNewService(googleDocs);
```

Listing 5. Defining the GoogleDocs Service-object and Registering on the Service-Bus

4.3 Generic Service-Access Layer

While the *ServiceBus API* provides a high-level interface for users to interact with services, these high-level calls need to get translated into more concrete instructions in order to interact with the various heterogeneous services that they represent. Accordingly, depending on the type of service, an appropriate access-client needs to be selected. We implement three main types of generic service-access clients: a RESTful client, a WSDL client and a set of Feed-based clients, one for each different type of interaction-style, such as: polling, streaming or publish-subscribe.

Operation Invocation. In order to demonstrate the process involved in invoking a service operation, consider the example shown in Listing 6 below. In this code snippet, we demonstrate how a client might invoke a call to the `/GoogleDocs/SearchDocuments` operation.

```
1. Message input = new Message();
2. input.addField(new AttributeField<String>("Search_Keys","service+oriented+design"));

3. Message output = ServiceBus.invoke(access_key, "/GoogleDocs/SearchDocuments", msg);
```

Listing 6. Invoking the operation /GoogleDocs/SearchDocuments with a sample Message

Behind the scenes, the service-bus then translates this simple high-level call into a more concrete set of instructions, which is then routed to the generic service-access client in order to actually perform the call. For the example provided above, the workflow would typically be implemented as follows:

1. Firstly, information about the service would be retrieved. Such as the *type*, which would reveal it is a `REST` service, as well as the *BaseEndpoint b, (https://docs.google.com/feeds)*. Accordingly, the appropriate generic REST-client would be selected in order to service out the request.
2. Next, information about the operation-type would be retrieved. In this case, the *HttpPath p (/default/private/full)*, and *HttpMethod m (get)*.
3. Next, the input message would need to be parsed, where the message-fields are mapped into the native message types. For example, in this case the mapping as shown in Listing 4(b) specifies that the field-value provided in `Search_Keys`, should be written to the Http parameter named "q"; this then results in having one parameter-value, *(q="service+oriented+design")*.
4. Finally, in order to derive the concrete call, the base-endpoint *b*, is concatenated with the path *p*, and then the parameter(s) *q* are attached. Accordingly, the following call is generated:
   GET *https://docs.google.com/feeds/default/private/full?q=service_oriented_design*
5. On return the data can then be returned to the caller. In this case it also involves similarly transforming the raw-message into the appropriate message-fields, using again the mapping-rules to accomplish this.

Access to Secure Resources. In addition to the above, the service-bus is also required to handle calls to secure resources, i.e. these are requests to operation or feed-source endpoints that require authentication in order access their resource. Whether or not a particular call to a service requires authentication is something that is specified by the service-owner. For instances, using the above example, we note Google requires authentication for the `SearchDocuments` operation-type, that we defined earlier. Accordingly we notice that when the service-curator models and registers this service API, the `setRequiresAuth` flag is assigned to `true` for this operation. (As can be seen at Line 11, Listing 3 in the example model shown above).

Therefore in this example, after the concrete call has been formulated, the service-bus checks whether this call requires authentication by retrieving this information from the defined API metadata. If authentication is required, a valid *access-key* would be expected to be passed in. As mentioned earlier, the *access-key* is a token shared between the end-user and application, using the OAuth protocol, [8]. The underlying idea here is quite similar to the notion in traditional application-development, where for example if an application is built on top of Facebook, and the application is required to access a specific user's comments, it would require that a secret token be shared between the user and application. However, in such traditional application development approaches, the application needs to explicitly manage how secure calls are made, such as signing the requests with the appropriate signature encryption, (e.g. HMAC-SHA1), and more so, it needs to manage this for 'each' secured service that it used. In contrast however, application built on top of the service-bus can have these complexities pushed down to the service-bus, where no additional code is required other than just the need to pass in a single valid *access-key*, which further helps in simplifying the process of application development.

Feed and Subscription Management. At the low-level, the detection of feed-events (such as those sourced from RSS or ATOM feeds) within the service-bus is managed by the generic feed client module as shown in Figure 2. The main objective for the service-bus is to provide to application-developers a uniform and simplified interface for low-latency delivery when reading feeds. For purpose of example, consider we would want to monitor the activity feed on Google Docs, for the user "Moshe". In this case, an instance of the abstract `/GoogleDocs/ActivityFeed` feed-source can be created, which will then allow us to query (or listen) to events that are detected from this feed. The code snippet in Listing 7 demonstrates the registration of this feed instance, as well as the method used to create a subscription to it.

```
1. FeedInstance activity_feed =
       new FeedInstance(googleDocs.getFeedByName("ActivityFeed"));
2. activity_feed.setInstanceName("MoshesGDocActivity");
3. activity_feed.setPathParam("user_id", "moshe…@gmail.com");
4. activity_feed.setPublic(false);

5. //Register the feed-instance on the service-bus:
6. ServiceAPI.registerFeedInstance(access_key, activity_feed);

7. //Create a subscription to this event:
8. String subscription_id =
       ServiceBus.subscribe("/GoogleDocs/ActivityFeed/MoshesGDocActivity");
```

Listing 7. Example of registering and subscribing to a Feed

In this particular example, we may note from the code earlier (Listing 1, Line 38), the interaction-style set for detecting events from this feed-source is *"polling"*. This information allows the service-bus to select the correct generic-client in order to interact with the feed. Other types of generic clients, as mentioned earlier, could be *publish-subscribe*, and *streaming*. However, the main point to note is that again we see how more simpler and abstract calls can be made by the application-developer in order to read feeds. This is because much of the complexities are pushed-down to the ServiceBus to process, and thus developers can be offered with a uniform interface for interacting with feeds irrespective of the underlying technical heterogeneity. In Listing 8 below we demonstrate how a feed-event callback could be set-up in order to listen and process feeds, in an asynchronous manner.

(The syntax below utilises customized Java-annotations for marking callback methods).

```
1. public class EventHandlers {
2.   @EventCallback(tag="handler_id")
3.   public void MyHandler(List<FeedMessage> results, String subscription_id){
4.      /*
5.       * Process this event on callback...
6.       */
7. }
8. ServiceBus.addEventListner(access_key, subscription_id,
                                new EventHandlers(), “handler_id”);
```

Listing 8. Example of setting up an event callback for asynchronous processing of feeds

Working behind the scenes, in order to provide this uniform interface, the service-bus maintains a flexible-sized event-queue that caches all incoming event messages. Therefore, regardless of how feed-events are being read, i.e. polling, streaming or pub-sub, it is possible to create a trigger on the queue, in order to be able to “push” events to clients that are listening to it. The caching of events allows decoupling between the feed-provider and client, and thus decreases latency of event-delivery. A subscription to an event means that a pointer to the latest read feed is maintained for each subscription, and thus feeds can be read at the subscriber’ own pace.

5 Use-Cases in Application Development

In order demonstrate the capabilities of the various features offered by *ServiceBase*, we have developed two applications written using the bus-platform that would be typically representative of real-world application-development activities. The objective is to demonstrate how effective solutions can be achieved in a relatively simplified manner than would otherwise be available using traditional application-development approaches.

5.1 Calculator for User-Contributions in Google Documents

Google Documents is a useful online word-processor that allows users to create and format text documents, and collaborate with other people in real-time. Similar to other tools in this category, the benefit of providing such software applications over the Internet means that multiple and distributed users can all work together to simultaneously contribute towards the production of a document. However, in the case of Google Docs, we notice one particular shortcoming: namely, there is no way to measure the contributions of each user. To solve this problem, we show how a simple application can be built using *ServiceBase* in order to achieve this goal.

To implement this application we first assume a service “GoogleDocs” has been registered and made available by an appropriate service-curator. Then the following application can be provided, as has been shown in Listing 9, in order to provide a numeric measurement of user-contributions of a specified Google Document.

```
1. public class GDocUserScore {
2.   public static void main(String[] args) throws Exception {

3.     DecimalFormat df = new DecimalFormat("#.##");
4.     String access_key = args[0];
5.     String resource_id = args[1];

6.     Message msg_in = new Message();
7.     msg_in.addField(new AttributeField<String>("resource_id", resource_id));

8.     ListField users = (ListField) ServiceBus.invoke(access_key,
            "/GoogleDocs/GetSharedUsers", msg_in).getField(“Users”);

9.     ListField revisions = (ListField) ServiceBus.invoke(access_key,
            "/GoogleDocs/GetDocumentRevisions", msg_in).getField(“Revisions”);
```

```
10.     Map<String,Integer> user_score = new HashMap<String,Integer>();
11.     for(TupleField user : users){
12.        String username = ((AttributeField<String>)
                              user.getField("username")).getValue();
13.        user_score.put(username,0);
14.     }

15.     for(TupleField revision : revisions){
16.        String username = ((AttributeField<String>)
                              revision.getField("username")).getValue();
17.        Integer current_score = user_score.get(username);
18.        user_score.put(username,current_score+1);
19.     }

20.     System.out.println("GDoc Contribution Scores for Resource ID: " + resource_id);
21.     for(String username : user_score.keySet()){
22.        double score = (double) user_score.get(username) / revisions.size();
23.        System.out.println("[" + username + ",\t" + df.format(score*100) + "%]");
24.     }
25.   }
26.}
```

Listing 9. User-Contribution Calculator App for Google-Documents

For simplification, we demonstrate the above application as a Java-based command-line program; whereby the code-snippet in Listing 10 shows a sample run of this application.

```
1. %java GDocUserScore b128d7c4-7893-4b9d-613-770 1JumUL4_5OYJ3Gte-IQU8Oz1zAHLQQ

2. GDoc Contribution Scores for Resource ID: 1JumUL4_5OYJ3Gte-IQU8Oz1zAHLQQ
3. [rranjans,         3.84%]
4. [jeffreydongwei,   11.53%]
5. [moshechaibarukh,  19.23%]
6. [alagares,         50.00%]
7. [eilishorouke,     15.38%]
```

Listing 10. Sample run of the Google contributions calculator program

5.2 Using Live Social-Networks to Enrich Local User-base

It is quite common amongst many applications where it is necessary to maintain the set of users who are associated with the particular application. This set of users, which is more commonly referred to as the "user-base" plays an integral role as it discloses key information about the users that could in turn be used to drive the application. For example, distinguishing between users of a certain age group, location, or expertise; or even, networking between friends or a certain groups of friends, etc. In general we notice that common user-base meta-information might include information such as: a user-profile, (e.g. full name, location, image, age, gender, etc.), as well the relationships between one user and another (e.g. friendship, followers, etc.).

Although, in the traditional methods that applications are developed today, there are clear challenges in order to achieve this, such as: (i) obtaining the information – often this needs to be entered by the user; (ii) maintaining the information – when users status change their profile needs to be updated; (iii) as the number of users increase so does the size of the user-base and thus this large graph of user information and relationships must be stored.

However, in this section we show how many of these issues can be solved in a relatively simple manner by employing the service-bus to mediate between data held on social-network services, and the applications that are built on top of them. We therefore propose that: instead of maintaining the user-base information manually, much of the information can be crawled from various social-network services. The benefit of this is several-fold: (i) the physical store of data only needs to be relatively small, and instead of maintaining a complete graph of information, much of the information nodes can be obtained virtually, and on-demand when queried; (ii) virtual-data also means that there is no need to worry about having data updated, as this is done by the 3rd-party, and thus every time we query for information, we can assume to get the most current data; (iii) we can get access to

far more accurate data than otherwise what an individual application could accomplish in its own merit. For example, services such as *StackExchange* [16] are specialised applications particularly designed to maintain what could arguably be, a reliable reputation score for its users, and therefore could be used as a reliable score to assess the technical reputation for user. While, the concept itself of building data from services is not new, often this involves applications having to connect separately to various different services, and thereby also having to interpret and aggregate raw-data to appropriately extract the required information. This consequently makes implementing such applications very time-consuming and heavy to maintain.

In contrast however, we demonstrate how the service-bus could be employed to provide seamless access to a range of social-network data, in this case, for the purpose of crawling and constructing a virtual user-base. The particular model we implement has been illustrated in Figure 6 below.

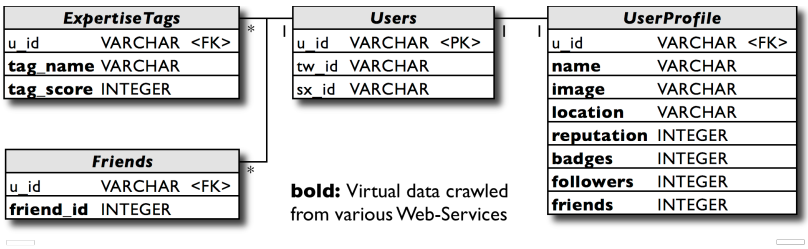

Figure 6. Entity-Relationship Model for Service-Enriched User-Base Store

In fact, the solution we provide here could be considered a unique extension to *ServiceBase* since we show how access to certain services could be embedded within a native database environment – thus exposing a familiar SQL-interface. Since the application has been implemented for PostgreSQL, we have used PL/Java [27] as the procedural-language module-extension used to call upon Java classes and return virtual tables. Assuming that the required services *StackExchange* and *Twitter* has been registered and made available in the service-bus (for example, similar to the service-models show in Appendices A and B respectively), we can therefore show at Listing 11 below the code that we would be produced in order to implement one of the virtual-views, for example the *UserProfile* view, corresponding to the data-model above. We also note how again message-fields simplify the process of interpreting raw-data to more easily be represented in an appropriate virtual-table format.

```
1. public class UserProfile implements ResultSetProvider {

2.   public final Map<String,String> results = new HashMap<String,String)();
3.   public UserProfile(Map<String,String results){
4.     this.results = results;
5.   }

6.  public boolean assignRowValues(ResultSet receiver, int currentRow)
    throws SQLException {
7.     for(String att : results.keySet()){
9.        receiver.updateString(att, results.get(att));
9.     }
10. }

11. public static ResultSetProvider getProfile(String u_id, String tw_id, String sx_id)
    throws SQLException {
12.    Message msg_in1 = new Message();
13.    msg_in1.addField(new AttributeField<String>("user_id", tw_id);
14.    Message twitter_profile = ServiceBus.invoke(null,
              "/Twitter/GetUserProfile", msg_in1);
15.    TupleField tw_profile = (TupleField) twitter_profile.getField("Profile");

16.    Message msg_in2 = new Message();
17.    msg_in2.addField(new AttributeField<String>("user_id", sx_id);
18.    Message stackX_profile = ServiceBus.invoke(null,
              "/StackExchange/GetUserProfile", msg_in2);
19.    TupleField sx_profile = (TupleField) stackX_profile.getField("Profile");
```

```
20.     HashMap<String,String> results = new HashMap<String,String>();
21.     results.put("u_id",       u_id);
22.     results.put("name",       tw_profile.getField("full_name").getValue());
23.     results.put("image",      tw_profile.getField("image_url").getValue());
24.     results.put("location",   tw_profile.getField("location").getValue());
25.     results.put("reputation", sx_profile.getField("reputation").getValue());
26.     results.put("badges",     sx_profile.getField("badge_count").getValue());
27.     results.put("friends",    tw_profile.getField("friends").getValue());
28.     results.put("followers",  tw_profile.getField("followers").getValue());

29.     return new UserProfile(results);
30. }
31.}
```

Listing 11(a). PL/Java Function that uses the ServiceBus to call services, interpret and aggregate data and return as a virtual-table

```
1.  CREATE FUNCTION getUserProfile(u_id, tw_id, sx_id)
2.  RETURNS TABLE (u_id, name, image, location, reputation, badges, followers, friends)
3.  AS
4.    'my.package.UserProfile.getProfile'
5.  IMMUTABLE LANGUAGE JAVA;

6.  CREATE VIEW UserProfile AS
7.   SELECT (prof).u_id, (prof).name, (prof).image, (prof).location,
8.          (prof).reputation, (prof).badges, (prof).followers, (prof).friends
9.   FROM(
10.     SELECT
11.       getUserProfile(u_id, tw_id, sx_id) as prof
12.       FROM Users
13.  ) temp
```

Listing 11(b). Corresponding SQL Implementation to expose the getUserProfile function and view

We may now see, by using the above, an application can query the user-base in just the same way as it would if this was an ordinary local data-store. For example, finding all users from 'Sydney' would render a familiar query such as: `SELECT u_id from UserProfile WHERE location LIKE '%Sydney%'`. This therefore not only simplifies the interface for service-access since it completely hides the low-level complexities behind virtual tables; it also enables the developer to utilise all other standard database features, such as: Joins, Sorting, Filtering, or even other built-in functions, etc. More so, this also provides the opportunity to combine both material and virtual data.

6 Evaluation

As the primary goal of *ServiceBase* has been to simplify access and integration of web-services in application development, we have accordingly conducted a user-study in order evaluate the overall effectiveness of our proposed approach. In order to conduct the study, we have chosen two development scenarios, both of which have been adopted from the use-cases presented earlier: (i) The first involved writing a Java-application to calculate the percentage user-contribution of all users towards a specified Google Document; (ii) While, the second involved completing a PL/Java function in order to implement the *UserProfile* virtual function. We specifically chose the first scenario to be relatively easier to the second, and this helped to get a balanced evaluation. In the case of the second, although there was some dependence on using PL/Java, this posed no prerequisite knowledge requirements on the participants, as appropriate code-stubs were provided whereby the exercise only involved the implementation of the inherent Java-function.

Ultimately, the key factors used to measure effectiveness, were: (i) The total number of lines-of-code excluding white-space and comments; (ii) Number of extra dependencies needed; and (iii) Time taken to complete task. In addition, (iv) maintainability of the code was also tested, which was done by extending each of the scenarios with additional requirements, thus creating a second phase of development. We extended the first scenario by asking participants to also locate related documents and to perform a contribution-calculation for each such related document. Similarly, for the second scenario, we had asked participants to implement the *ExpertiseTags* and *Friends* virtual function. To ensure effectiveness, prior to the second phase, code was re-distributed amongst the participants, so that each would be asked to extend code that was not their own.

The study was conducted on a total of five participants, all of which possessed an average to moderately-high level of software development expertise. In order to further balance the evaluation, three participants were asked to attempt the implementation using traditional techniques first, and then secondly using *ServiceBase*; whereas the other two participants were asked to do this in reverse. Subsequently, during the second development phase, the order of implementation was again reversed. The results of our study are shown in graphs at Figure 7 below.

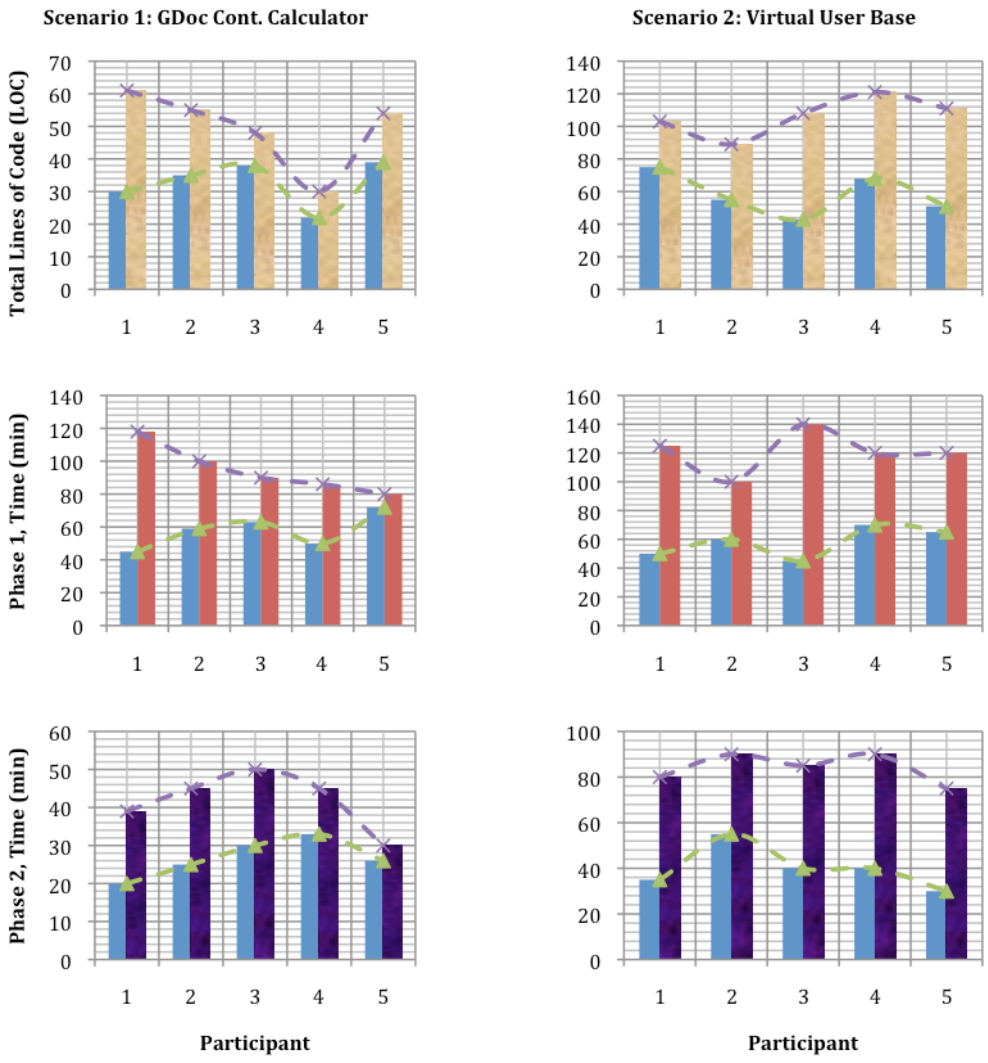

Figure 7. User-Study Results (Bar1: using ServiceBase; Bar2: traditional methods)

As most participants did not have much prior knowledge about the *ServiceBase* API, there was notably extra time needed for the participants to understand the API before they were comfortable to begin developing. However in most case, this was generally offset by there otherwise being no great need to be knowledgeable in any other library API. In contrast however to the traditional development approaches, while frameworks such as

the Jersey API [28] was generally familiar to all participants, notably all struggled with the less-common cases as was apparent when interpreting calls made from StackExchange. In this particular case, data-content returned was compressed into GZIP format and therefore required the developer to implement the appropriate decompression techniques in order to read and manipulate the data. Similarly, we also found that most participants spent a reasonable amount of time to find appropriate libraries for XML and JSON parsing. While, one particular participant chose a more manual approach this inevitably resulted in an increased number of source-code lines thus posing a greater risk of error-prone code.

As an overall analysis, it is clear that across all participants and for both development scenarios, the number of lines of code and time taken to complete the task is visibly reduced when using *ServiceBase* than in comparison to the traditional development approaches. In general as well, while the implementations using *ServiceBase* did not require any additional libraries, for these application development scenarios the traditional approaches required on average at least two (to three) additional libraries. In light of these results, this evaluation study successfully demonstrates the anticipated benefit of our proposed approach.

7 Related Work

Web-Service Types, Modelling Techniques & Concerns. There are two clear widely accepted representation approaches for services, namely SOAP and REST [2,5,7]. Although recently the notion of Feed-based services (as a form of REST service), has also been receiving considerable attention, [17]. Nonetheless, while both strive to achieve the same underlying goal of allowing descriptions of low-level APIs, there has in fact been much debate about whether "REST has replaced SOAP services!" [6], or questions posed relating to "which one is better?" [18]. While the conclusions of these debates are largely beyond the scope of our exploration, it is clear based on statistics that RESTful service has by far outweighed SOAP service offerings. In fact, at the time of writing, there has been a reported 500 SOAP services in contrast to over 2,800 RESTful services. The clear and widely understood reasons for this is due to the fact that RESTful services are by far more easier to understand and provides better support for modern web-technology. For example, whereas SOAP enforces XML, RESTful services have no problem in supporting JSON, which not only provisions a more human-friendly representation, but also enables increased support for more modern web-technology, such as embedding in JavaScript and Ruby. SOAP services on the other hand have mainly sprouted with the enterprise in mind resulting in a more verbose architecture, particularly WSDL for describing services guided by the increasingly family of WS-* standards. However, it is precisely the lack of standards surrounding REST that has been the key attributing factor that have polarized the community for or against REST being the next generation of web-services technology, [19]. In an attempt to fill this void not long ago JBoss had formally announced a new REST-* initiative [20] which aims to introduce a level of formalism whilst not overpowering the fundamental REST principles of being a platform aimed at the end-user. Although these formalisms still remain vague to be widely accepted.

In another direction of work, Semantic Web Services (SWS) for REST has been proposed and mainly focuses on providing a semantic description of a REST service. SA-REST [22] and hREST/MicroWSMO [23] provide a list of input and output parameters, methods, and URIs exposed by a REST service by means of property value pairs or RDFa [24] annotations. The description itself can be transformed to RDF using a GRDDL-based [25] strategy for generating a domain ontology in RDF, but no information about the REST resources themselves are retrieved. Nonetheless, while the benefit of representing service-data as RDF data-stores mean standard languages such as SPARQL can be used to manipulate this information, often they require manual and costly construction of a complex ontology before exploitation.

Web-Services Repositories, Access Techniques & Concerns. Ultimately, the value of service-models is primarily assessed by it's usefulness, primarily with respect to: whether services can be stored in repositories and explored; and whether the model enables us to leverage a degree of automated support when we want to utilize service in the course of application development. In the SOAP community, while standards such as UDDI were proposed some time back with the intention of acting as a global-repository, it seemed the ideas soon failed where the emphasis has shifted to simply relying on web-based engines in order to locate services, in just the

same way it is done for RESTful services. For example, repositories such as *ProgrammableWeb* list thousands of APIs, however clearly not much of the meta-information available would be useful to support or simplify service-execution or integration of the service in developments. Of course, while a WSDL could be used to bind to services, such as with the assistance of generated code-stubs, this often creates distributed applications that are too tightly coupled, lengthy and error-prone code.

Towards an abstract architecture for uniform presentation of resources. To address these challenges, we have thus been motivated us to propose a unified service representation model, which is an essential component in order to provide a common interface for interacting with services. This means the heterogeneity of services can be masked by more high-level operations that automate the concrete set of instructions behind the scenes. From an architectural perspective, there are in fact several works that share the same motivation, although for other more specific domains.

For example, in the case of data-storage services, BStore [26] is a framework that allows developers to separate their web application code from the underlying user data-storages. The architecture consists of three components: file-systems, which could be considered as data-storage APIs or services in our model; the file-system manager acting as the middleware or service-bus in our model; and applications that require access to the underlying user-data stored in various storage services. A common-interface is then proposed for both loading storage-services onto the file-system, as well as for applications accessing this data via the middleware.

Another example is the system SOS [12], which also defines a common-interface in order to interact with non-relational (also called NoSQL) databases. The motivation here is the same; while these database provide superiority in certain ways in comparison with traditional relational databases, the lack of standards make it hard to deal with the heterogeneity of the language and interface. Similar to the concept of the unified-model we propose, they provide a meta-model approach to map specific interfaces of various systems to a common one. However, since the work mainly deals with data-storage services, the common set of operations is relatively simple, namely: get, put, and delete. Also relevant though, is that the work deals with providing a common model for run-time data obtained from various data-services. In this manner further similarity can be drawn to the message-fields and mapping component of our system, where interestingly they too identify three main constructs for modelling heterogeneous data, which they refer to as String, Collection and Object.

In the case of Feed-based services, the work at [17] presents an architecture for consumers of feeds to organize the services that they are using, share them, or a sub-set of them, or use it to build tools which would implement a decentralized system for publishing, consuming and managing feed-subscription. We may identify the middleware in this framework to be the feed-subscription manager (FSM), which decouples consumers from the underlying feed-services. In this case, the common representation model for feeds are expressed itself via an Atom feed, holding relatively simple meta-data about feed-sources. Common operations to services from users or applications are then expressed via AtomPub in order to interact with the various underlying feed-sources.

However, in all cases mentioned above, while they share similar concepts of architecture, their applicability is still only limited to a particular domain.

8 Conclusions

Although the Internet continues to flourish with a growing number of APIs available, there still lie significant challenges in integrating services in everyday application development. Motivated by this need, we proposed in this paper a platform for simplified access to web-services. In order to achieve this, we first addressed the heterogeneity of various service representation types by proposing a unified service model – while in order to address the execution-time heterogeneity of service-message data we proposed a mapping framework. Empowered by both these works we then presented *ServiceBase*, a web2.0-oriented service-bus middleware offering a common programming interface for access to service, where high-level operations are transformed by the service-bus into more concrete calls. Furthermore, inspired from the Web2.0 paradigm, *ServiceBase* creates a community hub amongst service curators and consumers, such that service-knowledge can be incrementally enriched for the benefit of being shared and later reused by other distributed application developers.

References

[1] Yu. J., Benatallah, B., Casati, F., Daniel F., Understanding Mashup Development, IEEE Internet Computing, Volume 12, Issue 5, 2008, pp. 44-52, IEEE Computer Society.

[2] Pautasso C., et al., Restful Web Services vs. "Big" Web Services: Making the Right Architectural Decision, Proceedings of the 17 Intl. Conference on World Wide Web (WWW), 2008, pp. 805-814, ACM.

[3] Voida A., et al., Homebrew Databases: Complexities of Everyday Information Management in non-profit Organizations. Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI 2011). Vancouver, BC, May 7-12 2011, ACM Press.

[4] Chen, K., Hellerstein, J., Parikh, T.: Data in the First Mile, Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR), January 9-12, 2011, California, USA, Electronic Proceeding.

[5] Alonso, G., Casati, F., et al., Web services: Concepts, Architectures, and Application. 354 Pages. ISBN: 978-3-540-44008-6, 2004, Springer Verlag.

[6] How REST replaced SOAP on the Web: What it means to you. http://www.infoq.com/articles/rest-soap

[7] Geambasu, R., et al., Organizing and sharing distributed personal web-service data, Proceedings of the 17th International Conference on World Wide Web (WWW), 2008, pp. 755-764, ACM Press.

[8] OAuth. http://oauth.net/

[9] Wikipedia entry on 'Internet Media Type'. http://en.wikipedia.org/wiki/Internet_media_type

[10] Kwok, W.: Bidirectional transformation between relational data and XML document with semantic preservation and incremental maintenance, PhD Thesis, University of Hong Kong

[11] Google Image Chart API. https://developers.google.com/chart/image/

[12] Atzeni, P., Bugiotti, F., Rossi, L.: SOS (Save Our Systems): A uniform programming interface for non-relational systems. In EDBT 2012. Proceedings of the 15th International Conference on Electronic Conference. Berlin, Germnay. (2012)

[13] OrientDB Open Source Graph-Document NoSQL dbms. http://www.orientdb.org/index.htm

[14] Amazon Simple DB. http://aws.amazon.com/simpledb/

[15] Google Documents List API, V3.0. https://developers.google.com/google-apps/documents-list/

[16] StackExchange API v2.0. https://api.stackexchange.com/docs

[17] Wilde, E., Liu, Y.: Feed Subscription Management. University of California, Berkley School of Information Report 2011-042. (2011).

[18] REST and SOAP: When Should I Use Each/Both? http://www.infoq.com/articles/rest-soap-when-to-use

[19] Duggan, D., Service Oriented Architecture: Entities, Services, and Resources. NJ.: Wiley-IEEE Computer Society; 2012

[20] REST-star. http://www.jboss.org/reststar

[21] M. Hadley. Web Application Description Language (WADL), August 2009.

[22] Lathem, J., Gomadam, K., Sheth, A.: SA-REST and (S)mashups: Adding Semantics to RESTful Services. In First IEEE International Conference on Semantic Computing (ICSC 2007), pages 469–476, Irvine, California, September 2007.

[23] Kopecky. J., Gomadam, K., Vitvar, T.: hRESTS: An HTML Microformat for Describing RESTful Web Services. In 2008 IEEE/WIC/ACM International Conference on Web Intelligence, pages 619–625, Sydney, Australia, December 2008

[24] Adida, B., Birbeck, M., McCarron, S., Pemberton, S.: RDFa in XHTML: Syntax and Processing — A Collection of Attributes and Processing Rules for Extending XHTML to Support RDF. World Wide Web Consortium, Recommendation REC-rdfa-syntax-20081014, October 2008.

[25] Connolly. D., Gleaning Resource Descriptions from Dialects of Languages (GRDDL). World Wide Web Consortium, Recommendation REC-grddl-20070911, September 2007

[26] Chandra, R., Gupta, P., Zeldovich, N.: Separating Web Applications from User Data Storage with BStore. In WebApps, 2010.

[27] PL/Java Wiki. http://wiki.tada.se/index.php?title=Main_Page

[28] Jersey Framework. http://jersey.java.net/

Appendix A. Snippet of code showing registration of *StackExchange* Web-Service

```
1. public class RegisterStackexchange {

2.   public static void main(String[] args) throws Exception {

3.     //Define OperationType1: GetUserProfile
4.     HttpMessage http_msg1 = new HttpMessage();
5.     http_msg1.addPath("user_id");

6.     http_msg1.addField(new AttributeField<String>(“user_id”));
7.     http_msg1.loadMapping(“sx_getuserprofile_in_map.json”);

8.     JSONMessage json_message1 = new JSONMessage();
9.     TupleField profile = new TupleField(“Profile”);
10.    profile.addField(new AttributeField<String>(“display_name”));
11.    profile.addField(new AttributeField<String>(“location”));
12.    profile.addField(new AttributeField<String>(“reputation”));
13.    profile.addField(new AttributeField<String>(“badge_count”));
14.    json_message1.addField(profile);
15.    json_message1.loadMapping(“sx_getuserprofile_out_map.json”);

16.    OperationType otype1 = new OperationType("GetUserProfile");
17.    otype1.setHttpMethod("get");
18.    otype1.setHttpPath("users/{user_id}/");
19.    otype1.setInputMessage(http_msg1);
20.    otype1.setOutputMessage(json_message1);

21.    //Define OperationType2: GetTopAnswerTags
22.    HttpMessage http_msg2 = new HttpMessage();
23.    http_msg2.addPath("user_id");
24.    http_msg2.addParam("page_size");

25.    http_msg1.addField(new AttributeField<String>(“user_id”));
26.    http_msg1.addField(new AttributeField<String>(“page_size”));
27.    http_msg1.loadMapping(“sx_topanswers_in_map.json”);

28.    JSONMessage json_message2 = new JSONMessage();
29.    TupleField topanswers = new TupleField(“TopAnswers”);
30.    topanswers.addField(new AttributeField<String>(“tag_name”));
31.    topanswers.addField(new AttributeField<String>(“answer_count”));
32.    topanswers.addField(new AttributeField<String>(“answer_score”));
33.    json_message2.addField(profile);
34.    json_message2.loadMapping(“sx_topanswers_out_map.json”);

35.    OperationType otype2 = new OperationType("GetTopAnswerTags");
36.    otype2.setHttpMethod("get");
37.    otype2.setHttpPath("users/{user_id}/top-answer-tags");
38.    otype2.setInputMessage(http_msg2);
39.    otype2.setOutputMessage(json_message2);

40.    //Define: StackExchange Service Object Definition
41.    RESTService stackexchange =
               new RESTService("StackExchange", "http://api.stackoverflow.com/2.0/");
42.    stackexchange.addOperationType(otype1);
43.    stackexchange.addOperationType(otype2);

44.    //Register Service on ServiceBus:
45.    String service_id = ServiceAPI.registerNewService(stackexchange);

46.  }
47.}
```

Listing A1. RegisterStackExchange.java

```
1. {"user_id" : {
2.     "value" : "write(httpparam(user_id))"
3. }}
```

Listing A2. sx_getuserprofile_in_map.json

```
1.   "Profile" : {
2.      "display_name" : {
3.         "value" : "read(xpath('/items/display_name'))"
4.      },
5.      "location" : {
6.         "value" : "read(xpath('/items/location)')"
7.      },
8.      "reputation" : {
9.         "value" : "read(xpath('/items/reputation)')"
10.     },
11.     "badge_count" : {
12.        "value" : "read(xpath('/items/badge_count)')"
13.     }
14.  }
```

Listing A3. sx_getuserprofile_out_map.json

```
1. {"user_id" : {
2.      "value" : "write(httpparam(user_id))"
3.  },
4.  "page_size" : {
5.      "value" : "write(httppath(page_size))"
6. }}
```

Listing A4. sx_topanswers_in_map.json

```
1.   "Profile" : {
2.      "tag_name" : {
3.         "value" : "read(xpath('/items/tag_name'))"
4.      },
5.      "answer_count" : {
6.         "value" : "read(xpath('/items/answer_count'))"
7.      },
8.      "answer_score" : {
9.         "value" : "read(xpath('/items/answer_score'))"
10.     },
11.  }
```

Listing A5. sx_topanswers_out_map.json

Appendix B. Snippet of code showing registration of *Twitter* Web-Service

```
1. public class RegisterTwitter {

2.   public static void main(String[] args) throws Exception {

3.     //Define OperationType1: GetUserProfile
4.     HttpMessage http_msg1 = new HttpMessage();
5.     http_msg1.addParam("user_id");

6.     http_msg1.addField(new AttributeField<String>(“user_id”));
7.     http_msg1.loadMapping(“tw_getuserprofile_in_map.json”);

8.     XMLMessage xml_message1 = new XMLMessage();
9.     TupleField profile = new TupleField(“Profile”);
10.    profile.addField(new AttributeField<String>(“full_name”));
11.    profile.addField(new AttributeField<String>(“image_url”));
12.    profile.addField(new AttributeField<String>(“followers”));
13.    profile.addField(new AttributeField<String>(“friends”));
14.    xml_message1.addField(profile);
15.    xml_message1.loadMapping(“tw_getuserprofile_out_map.json”);

16     OperationType otype1 = new OperationType("GetUserProfile");
17.    otype1.setHttpMethod("get");
18.    otype1.setHttpPath("users/lookup.xml");
19.    otype1.setInputMessage(http_msg1);
20.    otype1.setOutputMessage(xml_message1);

21.    //Define OperationType2: GetFriends
22.    HttpMessage http_msg2 = new HttpMessage();
23.    http_msg2.addParam("user_id");

24.    http_msg1.addField(new AttributeField<String>(“user_id”));
25.    http_msg1.loadMapping(“tw_getfriends_in_map.json”);

26.    XMLMessage xml_message2 = new XMLMessage();
27.    AttributeField<String> friend_id = new AttributeField<String>(“friend_id”);
28.    ListField friendslist = new ListField(“FriendsList”, friend_id);
29.    xml_message2.addField(friendslist);
30.    xml_message2.loadMapping(“tw_getfriends_out_map.json”);

31.    OperationType otype2 = new OperationType("GetFriends");
32.    otype2.setHttpMethod("get");
33.    otype2.setHttpPath("friends/ids.xml?cursor=-1");
34.    otype2.setInputMessage(http_msg2);
35.    otype2.setOutputMessage(xml_message2);

36.    //Define: Twitter Service Object Definition
37.    RESTService twitter = new RESTService("Twitter", "https://api.twitter.com/1/");
38.    twitter.addOperationType(otype1);
39.    twitter.addOperationType(otype2);

40.    //Register Service on ServiceBus:
41.    String service_id = ServiceAPI.registerNewService(twitter);
```

Listing B1. RegisterTwitter.java

```
1. {"user_id" : {
2.     "value" : "write(httpparam(user_id))"
3. }}
```

Listing B2. tw_getuserprofile_in_map.json

```
1.   "Profile" : {
2.      "full_name" : {
3.         "value" : "read(xpath('/users/user/name'))"
4.      },
5.      "image_url" : {
6.         "value" : "read(xpath('/users/user/image_url'))"
7.      },
```

```
8.      "friends" : {
9.         "value" : "read(xpath('/items/friends_count'))"
10.     },
11.     "followers" : {
12.        "value" : "read(xpath('/items/followers_count'))"
13.     }
14.  }
```

Listing B3. tw_getuserprofile_out_map.json

```
1. {"user_id" : {
2.      "value" : "write(httpparam(user_id))"
3. }}
```

Listing B4. tw_getfriends_in_map.json

```
1. {"FriendsList" : {
2.    "nodepath" : "xpath('/id_list/ids/id')",
2.    "friend_id" : {
3.         "value" : "read(xpath('/id_list/ids/id/text()'))"
4.    },
5. }
```

Listing B5. tw_getfriends_out_map.json