SOAlive Service Catalog: A Simplified Approach to Describing, Discovering and Composing Situational Enterprise Services

Ignacio Silva-Lepe, Revathi Subramanian, Isabelle Rouvellou, Thomas Mikalsen, Judah Diament, and Arun Iyengar

IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne NY 10532, USA
{isilval,revathi,rouvellou,tommi,djudah,aruni}@us.ibm.com
http://www.research.ibm.com/

Abstract. SOAlive aims at providing a community-centric, hosted environment and, in particular, at simplifying the description and discovery of situational enterprise services via a service catalog. We argue that a service community has an impact not only on users and services, but also on the environment itself. Specifically, our position is that a service catalog adds value to users, and is itself enriched, by its incorporation into a community-centric service hosting environment. In addition, analyses of web services directories suggest that a catalog service for enterprise services can be better provided by using a simpler content model that better fits REST, taking advantage of collaborative practices to annotate catalog entries with informal semantic descriptions via tagging, providing a mechanism for embedding invocations of discovered services, and allowing syntactic descriptions to be refined via usage monitoring. The SOAlive service catalog defines a flexible content model, a discovery function that navigates the cloud of tag annotations associated with services in a Web 2.0 fashion, and a service description refinement function that allows the actual use of a service to refine the service description stored in the catalog.

Key words: Service catalog, situational enterprise service, software as a service, service engineering, service assembly, SOA runtime

1 Introduction

As the field of service-oriented computing evolves, we observe a number of trends. In no particular order, one trend is the exposition of Web Services via REST[1] APIs. Another trend is the development of web applications using dynamic programming languages and frameworks, e.g., JavaScript with AJAX, and PHP.

[1] REST (Representational State Transfer) builds on the HTTP protocol principles to define an architectural style where entities, containers and behaviors can be seen as resources that are accessible via a uniform interface consisting of a fixed set of verbs [5, 15].

These languages enable rapid development and testing, provide expressive and powerful frameworks, and lead to the use of abstractions that are closer to the problem domain [8]. A third trend is the use of Web 2.0-style social and collaborative filtering practices such as bookmarking and tagging, as found in Web 2.0 services such as `del.icio.us` or `flickr`, where tagging is used to annotate shared content.

Situational enterprise services, as instances of situational applications [2], also result from these trends. An enterprise service (sometimes referred to as a situational enterprise service) is a small, primarily browser-based, situational application, typically exposing a REST interface through which it is invoked, for instance, a travel authorization application. More specifically, an enterprise service does not use a rigorous language, such as WSDL, to define its interface. In addition, it seems natural to take advantage of the sort of informal semantics that are conveyed by tagging, as opposed to using rigorous ontologies, to provide semantic descriptions of enterprise services. It is important to make it easy and painless for developers of enterprise services to: (1) publish and advertise services, (2) discover services, and (3) invoke or compose discovered services.

SOAlive aims at providing a community-centric, hosted environment that supports the development, deployment and management of enterprise services, and that provisions the required deployment and execution middleware. In particular, SOAlive aims at simplifying the description, discovery and composition of situational enterprise services via a service catalog. In [14], service communities are introduced as the combination of social and business communities with the purpose of exchanging services. In particular, service communities establish a dynamic platform where services of interest are contributed, grouped, consumed, and managed [3].

We argue that a service community has an impact not only on users and services, but also on the platform itself. Specifically, our position is that a service catalog adds value to users, and is itself enriched, by its incorporation into a community-centric service hosting environment. That is, users benefit more from a service catalog that is part of a service community than from an isolated service directory or even a limited form of service marketplace. Likewise, a service catalog that is incorporated into a service community environment can take advantage of service deployment and usage to improve on the quantity and quality of the information it provides.

Furthermore, the SOAlive service catalog aims at benefiting from some of the lessons learned from web services directories. In [6], Legner presents an analysis of web services directories that are based on the UDDI standard. Some of her observations and conclusions are: (1) "Despite the fact that private UDDI registries allow for advanced categorization and identification schemes, the investigated Web services intermediaries use rather simplistic search and categorization mechanisms", (2) "Unlike in other electronic markets, we were not able to observe increasing personalization and customization of the Web services offerings", (3) "Using the StrikeIron Marketplace API, software vendors are able to embed service invocations in their applications", and (4) "More sophisticated classification schemes which reflect the vocabulary of the target consumers are, in combination with complete and reliable service descriptions, a prerequisite for the discovery of suitable Web services".

All of this suggests that a catalog service for enterprise services can be better provided by: (1) using a simpler content model that better fits REST, instead of imposing a more complex model, such as UDDI, which relies on more complete technical specifications of services, (2) taking advantage of collaborative practices to annotate catalog entries with informal semantic descriptions via tagging, (3) providing a mechanism for embedding invocations of discovered services, and (4) using tagging as above, and allowing minimal syntactic descriptions to be initially entered, which can then be refined via usage monitoring. In this paper, we introduce the SOAlive service catalog, which provides a set of functions that satisfy these requirements.

The SOAlive service catalog defines a flexible content model that (1) requires little or no up-front user input, (2) can evolve over time, either automatically or from user input such as user-provided tags, and (3) facilitates the composition of services by generating snippets of invocation code that can be inserted into services under development that invoke discovered services. This content model is designed to contain both syntactic as well as informal semantic descriptions of enterprise services. The service catalog also provides a discovery function that navigates the cloud of tag annotations associated with services in a Web 2.0 fashion. Finally, the service catalog provides a service description refinement function that allows the actual use of a service to refine the service description stored in the catalog.

The remainder of this paper is organized as follows. Section 2 provides a short overview of the SOAlive hosted environment. Section 3 introduces the SOAlive service catalog and its content model. Section 4 provides more details on service discovery, and Section 5 focuses on the service description refinement function of the catalog. Section 6 describes one possible implementation of the catalog. Section 7 discusses related work. Finally, section 8 concludes the paper.

2 SOAlive Overview

SOAlive provides a hosted environment for developing, deploying and executing enterprise services. The diagram in Fig. 1 highlights these aspects with a focus on the interactions with the catalog.

At development time, a user can discover services, which can be composed into other services being developed via the use of code snippets generated by the catalog. Here, because the catalog is community-centric, users benefit from the usage and bookmarking of services by other users in the community to improve their discovery experience.

At deployment time, a user relies on the hosted environment, specifically the application manager, to publish his services to the catalog and to install them on the runtime that is provisioned by the hosted environment as well. Here, because the application manager and the catalog are integrated into the same

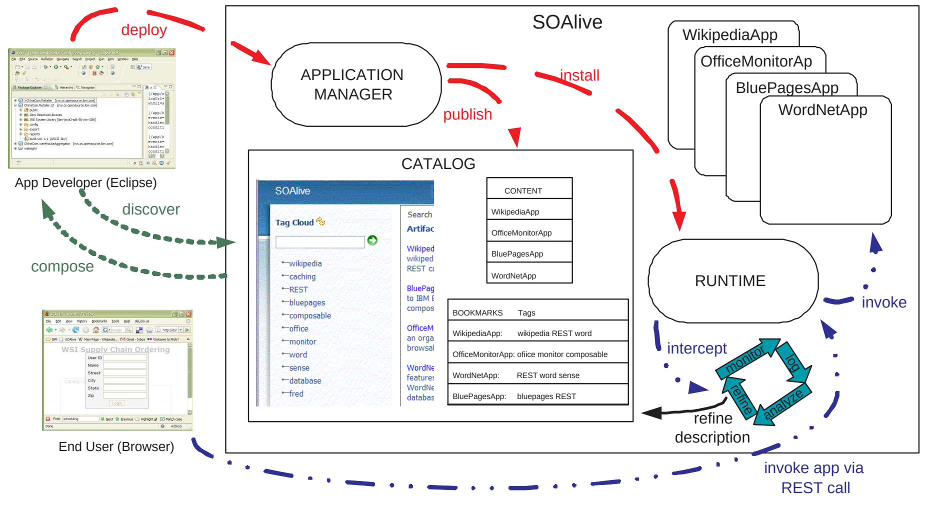


Fig. 1. SOAlive overview

hosted environment, deployed services are automatically published without any more intervention from the user.

At run time, service invocations, in addition to being routed to the target service, are also intercepted by the runtime. The runtime includes a monitor component that forwards the intercepted invocations to a logging component. Independently, a component that analyzes and processes the logged invocations interacts with the catalog to define and refine service descriptions. Here, because the runtime and the catalog are integrated into the same hosted environment, service invocations can be mined to extract valuable information that can be used in the definition and refinement of the corresponding service's description.

At any given time, users can also browse the catalog and, in a Web 2.0 fashion, share their discoveries of useful services by adding to the catalog bookmarks that are annotated with tags of their choice. This further improvement on the quantity and quality of the information provided by the catalog is a consequence of its community-centric nature and its integration into a service hosting environment.

3 SOAlive service catalog

The SOAlive service catalog is a core service in the SOAlive hosted environment. One main function of the catalog is to store descriptions of artifacts it knows about. As we shall see however, the catalog is not just a passive store of descriptive information. As the descriptions of the known artifacts are entered, the catalog can extract additional information from these descriptions in order to improve on them. Also, as the known artifacts are used, the catalog can process logged information about service invocations, to enhance the descriptions of those services.

3.1 Catalog artifacts

As a hosted environment, SOAlive supports not only the deployment, execution and management of services but also their development. As such, the SOAlive catalog is concerned with the description of not only services, but also of other artifacts that are used in the composition and development of services. An artifact that can be described in the SOAlive catalog is either a *service* that is deployed on SOAlive and is being managed by the SOAlive Application Manager, a *module* that is maintained in the SOAlive repository[2], or an external artifact. Deployed services can in turn be:

- **Callable:** These are services for which the catalog can include documentation which can be used to enable composition, e.g., a credit check service that can be composed into a larger mortgage approval service.
- **Browsable:** These are services that can be discovered in the catalog and invoked by just knowing the endpoint information, e.g., a retailer UI which can be acccessed by a simple HTTP GET with the artifact's URL.

Modules in the SOAlive repository can in turn be:

- **Deployable:** These are self-contained artifacts that can be deployed on their own, such as ProjectZero [11] applications or SCA [13] composites.
- **Composable:** These are artifacts that cannot be deployed in isolation, such as utility classes, ProjectZero application fragments, or SCA component implementations. Composable artifacts are typically composed to form deployable artifacts. As an example, a Bite extension activity [1] is a composable module that must be included as part of a larger flow in order to be deployed.

An external artifact is not hosted on SOAlive, e.g., an enterprise user directory service, but its description can also be in the catalog so that other SOAlive applications can discover and use it. This way, users can also benefit from bookmarking and tagging services that are not hosted by SOAlive. In many cases, SOAlive will host a proxy to an external service. This way, SOAlive can also monitor the use of external services in order to refine their description (see Section 5).

Catalog entries for artifacts are created by system entities such as the SOAlive Application Manager or the SOAlive Repository, or by an admin authority in the case of external artifacts, on behalf of a SOAlive user.

3.2 Content model

The SOAlive catalog maintains two kinds of artifact information: (1) the main description that is initialized at the time the artifact is published, often including information provided by the artifact's author, and (2) bookmarks (or references to artifacts) on behalf of users other than the artifact's author.

[2] The SOAlive Repository allows users to store and share application artifacts as modules. Modules are packaged into archive files files (e.g., zips and jars), and the SOAlive Repository provides an interface for uploading and downloading modules.

Main Description The main description of an artifact includes its name, owner identification, version number and visibility rules, all of which the catalog can provide default values for. The visibility rules are used for access control and can default to `public`, meaning any user can view and access the artifact. The description also includes a relative URI that is used as an endpoint reference, either as the URL of a deployed service, or as the URI of the artifact in the SOAlive repository. In addition, the main description of an artifact inculdes a syntactic as well as an informal semantic description.

Syntactic Description The syntactic description of a deployed or a deployable service is maintained as a stylized RESTful documentation of the interface exposed by the service. This interface enumerates the resources exposed by a service and, for each resource, information about its methods' data format, success and error codes, as well as typical examples of request and response. Representative examples of RESTful documentation that can be used in the syntactic description of a service in SOAlive are Project Zero's RESTdoc [11] and WADL [12]. The syntactic description of a composable artifact depends on the type of the artifact and so the catalog cannot assume any pre-determined schema or grammar. For instance, if the composable artifact represents a Bite extension activity [1], its syntactic descrption can be given by a piece of textual XML, or by a JSON serialized object containing the attributes that pertain to the extension activity.

Informal Semantic Description To a modern developer it would seem natural to be able to use collaborative filtering practices to publish and discover hosted enterprise services in a hosted environment such as SOAlive. To this end, the SOAlive catalog allows the inclusion of tags as an informal semantic description of a service, or an artifact in general. As in other Web 2.0 services, such as `del.icio.us` and `flickr`, tags used to annotate catalog artifacts induce a tag cloud that can then be used to navigate the space of artifacts.

Unlike other Web 2.0 services however, developers of enterprise services will be interested not only in the services that they have bookmarked but also in any service that is visible to them, regardless of who has published it or bookmarked it. For instance, if I am developing a mortgage application that requires the use of a credit check service, I would like to be able to discover, using a Web 2.0 mechanism such as a tag cloud, any service that may have been published to the hosted environment that may perform a credit check function. Therefore, the tag cloud available to a SOAlive developer should include any tag that annotates any service or artifact in the catalog. And as it is not uncommon for tag clouds in traditional collaborative filtering systems to become very large, it seems reasonable to expect a lower bound on the size of our tag cloud to be in the order of hundreds of tags.

To help in reducing the size of its tag cloud, the SOAlive catalog provides a tag consolidation facility. This facility relies on the ability to collect groups of tags given by some relationship amongst tags and then define the tag cloud as the collection of the representatives for each group of tags. In particular, tag grouping can be given by the hypernym relationship between the meanings or senses of any two tags. A tag group's representative in this case is given by the tag in the group for which there is no hypernym in the tag cloud. In turn, tag meaning or sense can be obtained from a public lexical database such as $WordNet$ [4].

More specifically, when a developer publishes an artifact to the catalog and annotates it with tags, if any such tag t_1 is given a sense s[3], then the catalog looks in the tag cloud for a tag t_0 whose sense s_0 is the hypernym of s. If t_0 is found then t_1 is added to t_0's group or hierarchy. If t_0 is not found in the tag cloud but it exists in the lexical database, then both t_0 and t_1 are added to the tag cloud under the same group. We refer to t_0 as a derived tag, given that it was not explicitly entered by a user but rather derived by the tag consolidation facility. In addition, if t_0 did not exist in the tag cloud, and there is a tag t_2 in the tag cloud whose sense is a hyponym of s_0, then t_0 and t_1 are added under the same group as t_2. For instance, suppose that a tag *investment* exists in the catalog as an annotation to a service that was published by a user u_1. When user u_2 publishes a service and annotates it with tag *banking*, the catalog determines that $finance$ has a sense that is a hypernym of both *investment* and *banking*, adds $finance$, and groups both *investment* and *banking* under it. Later, when the tag cloud is displayed, it will show $finance$ as a top level tag in the tag cloud.

Tag grouping using hierarchies provides a simple way to consolidate the size of the tag cloud, and to generalize the search for an artifact. For instance, after *investment* is grouped under $finance$, a search for artifacts under $finance$ will yield all entries that are tagged with any hyponym of $finance$, including *investment* and *banking*, which were entered by different users. Notice that although $funding$ is also a hyponym of $finance$, if it does not actually annotate any service in the catalog, it won't be part of the tag cloud and thus it won't be considered during the search. Finally, the SOAlive catalog also defines a number of system-architected tags to annotate artifacts according to their type. These tags include *callable*, *browsable*, *deployable*, and *composable*. These tags also include the corresponding pseudo-derived tags *deployed* and *in-repo* (this last one to annotate any module in the repository).

Bookmarks Bookmarking is another kind of collaborative filtering practice that a modern developer expects to be able to use to collect references to services of interest and to annotate those references with meaningful tags. Bookmarks can enrich the description of a given enterprise service S by allowing one or more users to refer to S with their own tags. This way, a hosted, community-centric platform such as SOAlive promotes collective informal semantic descriptions of enterprise services[4]

[3] Notice that it is also possible not to associate a sense with a tag, in which case such a tag does not participate in any grouping.

[4] Notice that a single user need only maintain a single bookmark per distinct service S.

The SOAlive catalog allows users to define and maintain bookmarks to collect references to services of interest, and to annotate those references with meaningful tags. Users can also browse other users' bookmarks; this contributes to making the SOAlive catalog a community-centric environment (not unlike `del.icio.us` and `flickr`) where developers share not only service offerings but also knowledge about other users' services that can be used for a particular purpose. The use of bookmarks also provides increased personalization of the SOAlive catalog by giving a user a view of the catalog contents through his or her bookmarks.

3.3 Code Snippet Generation

One goal of including a syntactic description of an artifact in its catalog main description is to provide support in the composition of services that use the artifact. From the RESTful documentation of a service it is straightforward to generate a code snippet that performs a simple invocation of the service from Javascript, Java or Groovy. For example, the following code snippet template can be used to invoke a service from Javascript, provided the fields that have the `<_cg_ ... >` pattern are filled out. These fields can be supplied by a sufficiently complete syntactic description of the service.

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
   if (xhr.readyState == 4) {
      if (xhr.status == <_cg_successCode>)
        {
           var response =  xhr.responseText;
        } else {
           <_cg_comment:_cg_errorCodes>
           alert("Error getting data from the server");
        }
    }
}

xhr.open("POST", <_cg_url>, true);
xhr.setRequestHeader('Content-Type', <_cg_format>);
var post_body = null;
<_cg_populate_body_from_example>
xhr.send(post_body);
```

Notice that, in particular, this code snippet does not perform any processing of the response. In general, there can be enough variability in what such a code snippet can do to make it infeasible for the catalog to attempt at a generic code snippet generation feature. Instead, the approach taken by the SOAlive catalog is to provide a plugin mechanism that allows a catalog-based tool developer to inject any arbitrary code snippet generator that uses the contents of a service's syntactic description as input. As a baseline, the SOAlive catalog includes a default plugin that generates simple code snippets for Javascript, Java and Groovy.

3.4 Discussion

As we have seen, the SOAlive catalog uses a simple content model that is suitable for describing enterprise services that are defined in terms of REST resources.

This content model places minimal requirements on the author of a service, a relative URI is enough as an initial description. Given a sufficiently complete syntactic description, in the form of RESTful documentation, the SOAlive catalog can provide support for embedding invocations of selected services by means of code snippet generation. The SOAlive catalog also takes advantage of collaborative practices to annotate service descriptions with informal semantic descriptions via tagging. The collection of all tags that annotate any artifact known to the catalog, published by any user, becomes a catalog-wide tag cloud that can then be used, as we shall see in the following section, to discover any artifact in the catalog. Finally, minimal syntactic descriptions can be made more complete and reliable by the use of a description refinement facility that the catalog provides as a way to enhance the content model. Section 5 elaborates on this refinement facility.

4 Discovery

As illustrated earlier, suppose that a credit check service has been published to the SOAlive catalog by Bob, and it can be used in the composition of a mortgage enterprise service being developed by Fred. In order for the credit check service to be effectively and efficiently composed into the mortgage service, the SOAlive catalog needs to provide a service discovery function that considers any service or artifact in the catalog, and is intuitive to Web 2.0 kinds of users.

The SOAlive discovery function is designed to allow a user to discover a catalog entry by drilling down on the tag cloud. That is, a user selects a tag from the tag cloud, which brings up all the catalog entries that contain the selected tag. Notice that if the selected tag t has any hyponyms in the tag cloud, then all tags in the hierarchy rooted at t will be considered when selecting catalog entries. That is, the selected entries will be all those that contain any tag in the hierarchy rooted at t. At this point the discovery function also displays a drill cloud, which is a tag cloud containing only tags in the current entry selection. The user can then select another tag from the drill cloud to further narrow down the contents of the entry selection. This procedure can be repeated until there is only one entry left or there are no more tags in the drill cloud. This procedure is an adaptation of [9].

However, notice that in [9] there is a separate drill cloud for each of a fixed number of categories, e.g., a keyword cloud and an author cloud. In addition, drill clouds are populated from the current selection of search results, which is initially obtained by using a domain-specific search form. In our case, the starting point of a search is the main tag cloud. In addition, tags do not fall under any given set of fixed categories; rather, tags are grouped according to their lexical sense. In some sense, our tag cloud can be thought of consisting of a dynamically changing set of categories, one for each top level tag at a given point in time. Thus, instead of trying to define separate drill clouds, a single drill cloud is used that collects all tags associated with any catalog entry in the current entry selection.

More precisely, the drill down discovery procedure consists of the following steps:

```
1. $currentTag \leftarrow$ Select[5] $tag$ from main tag cloud
2. $currentTags \leftarrow \{currentTag\}$
3. $entrySelection \leftarrow \{entry|$ for some $t \in currentTags$ hierarchy, $t$ annotates $entry\}$
4. do
   (a) $\left(\begin{array}{r} drillCloud \leftarrow \{tag|tag \text{ annotates some } entry \in entrySelection\} \\ -currentTags \end{array}\right)$
   (b) $currentTag \leftarrow$ Select $tag$ from $drillCloud$
   (c) $currentTags \leftarrow currentTags + \{currentTag\}$
   (d) $\left(\begin{array}{ll} entrySelection \leftarrow & \{entry|entry \in entrySelection \text{ and} \\ & \text{for } t \in currentTags,\ t \text{ annotates } entry\} \end{array}\right)$
   until $entrySelection$ is singleton or user selects one entry explicitly
```

Given that the main tag cloud collects all tags that annotate any entry in the catalog, this procedure considers all such entries when starting a search. In our scenario, as Fred knows a credit check service has been published that he would like to invoke from his mortgage service, he looks in the tag cloud and clicks on *deployed*. This brings all services with tags *deployed*, *callable* and *browsable*, given that these last two tags have been entered for several other deployed services, including credit check, and given that these two tags are hyponyms of *deployed*. Fred then selects *callable*, as he knows he wants to incorporate the credit check service by adding a call to it to his code. At that point he notices a *credit* tag in the drill cloud and selects it, narrowing down the entry selection enough to make it easy to find the credit check service. At this point, Fred can also examine further details about the credit check service, including its RESTful documentation, after which he decides this is the service he wants to invoke. Finally, at this point Fred can also obtain the simple JavaScript code snippet that performs the invocation, includes it in his code and makes a few changes to handle the response.

5 Service Description Refinement

As Legner [6] suggests, complete and reliable service descriptions are a prerequisite for the discovery of suitable (Web) services. On the other hand, it is important to make it easy and painless for enterprise service developers to publish, advertise and discover services. In addition, since typically an enterprise service exposes a REST interface through which it is invoked, as opposed to using a rigorous language, such as WSDL, to define its interface, there seems to be less of a motivation for a developer to provide an interface, not to mention a complete and rigorous one. Also, although the RESTful documentation of a service may not be mandatory, the more complete it is, the better the code snippet

[5] This selection is performed by the user via some appropriate user interface.

that can be generated. This would suggest to a developer a requirement on the development host (and its catalog in particular) that the documentation itself be generated. Furthermore, given the makeup of RESTdoc in particular, it seems feasible to infer the various documentation items (e.g., format, parameters) by example from successful as well as unsuccessful invocations of a service. In other words, given a log of service invocations that include: the URL of the service, invocation method, format, parameter values, and success or error codes, it seems feasible to synthesize the RESTdoc documentation or interface of the service.

The SOAlive catalog includes a service description refinement function that allows the actual use of a service to refine the service description stored in the SOAlive catalog. This in turn allows the catalog to produce an increasingly accurate service description, without requiring the service provider to specify a highly detailed service description. The basic mechanism of service description refinement is a Monitor, Log, Analyze and Refine loop.

- **Monitor** This is an interceptor that is registered with the SOAlive infrastructure, and that forwards service requests and responses to the log.
- **Log** As requests and responses arrive, the log extracts and collects information such as: request and response timestamps, identity of client, request method, request and response formats, parameters the service was invoked with, return value, sucess or error codes, and parent request correlator (that can be used to trace a chain of requests).
- **Analyze** On a thread separate to that of monitoring and logging, each log record is analyzed to determine what information in the log, if any, can contribute to the refinement of the service description. This analysis boils down to determining whether or not a log item has been accounted for in the service description. Items such as request format or error code may be as simple as determining whether they are included in a list. Other items, such as parameter values for requests that result in an error may depend on how the refine step accounted for them.
- **Refine** Service refinement targets both the syntactic and the semantic descriptions of a service. Syntactic description items include data format (such as JSON or XML), success and error codes, example request parameters, and example response values. Data format, and even sucess and error codes, are typically given by a relatively small (certainly finite) set of values. So in this case it makes sense to simply accumulate logged values into the description. Successful request parameters and response values, on the other hand, would not make sense to simply accumulate. Here, it makes more sense to learn an abstract description of the values seen so far. In the case of XML-formatted requests, an XML schema seems adequate. For JSON-formatted requests, although it is not a type-checked language, a similar descriptive schema could be abstracted from incoming request examples. Parameter values in requests that result in an error are more challenging. In addition to a schema that describes possible error values, it would also be useful for a user to know what actual values resulted in error. So the refinement must strike a balance between collecting too much raw data or abstracting it too much.

Service refinement can also be thought of as either intra-service or inter-service. Intra-service refinement targets the syntactic description of a single service. With inter-service refinement, the logged data pertain to more than one service. For instance, a parent request correlator can be used to refine the semantic description a service. Specifically, if the service that made the invocation is known, then its tags can be used as input to augment the tags of the invoked service. Here, a similarity metric (such as semantic distance) could be used to determine which tags from the invoking service to keep and which to discard.

6 Implementation

An implementation of the SOAlive hosted environment is available that supports the main aspects of developing, deploying and executing enterprise services.

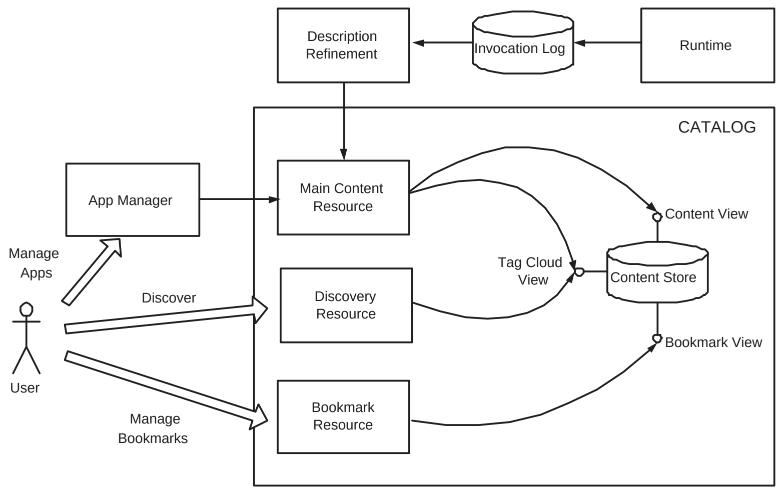
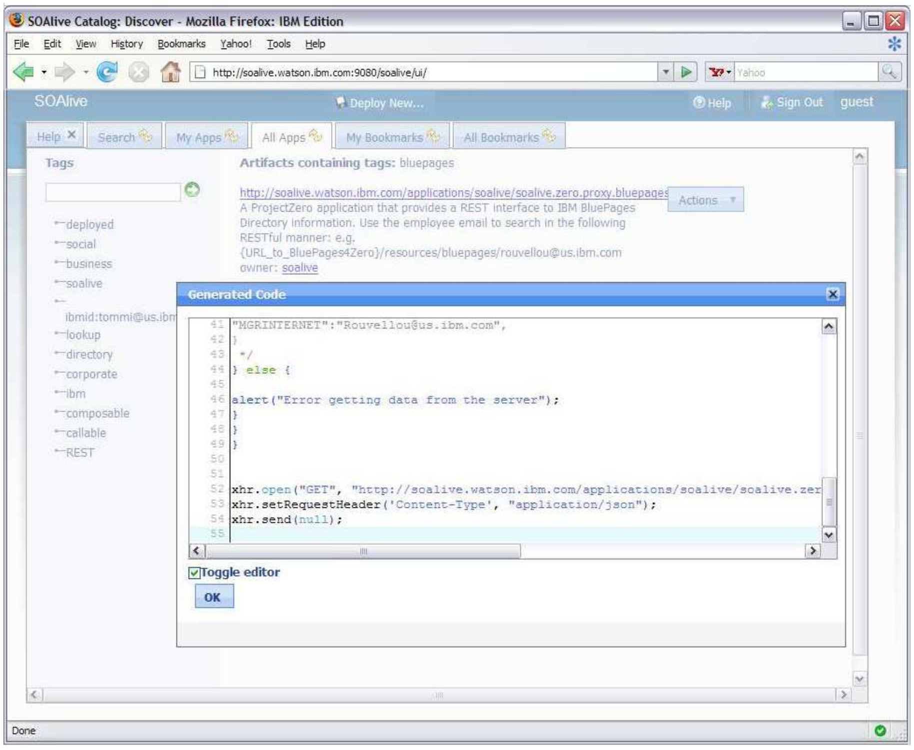


Fig. 2. SOAlive Catalog Architecture

The diagram in Fig. 2 illustrates the SOAlive catalog architecture. The service catalog is implemented as a number of REST resources that are also accessed locally by the application manager. In addition, the runtime and the service refinement function communicate indirectly via the log. The exposed REST APIs include create, read, update and delete operations for main content and bookmarks. There is also a REST API for discovery that retrieves the tag cloud and that returns a bookmark selection and corresponding drill cloud given a number of selected tags. In the current implementation, the actual drill down procedure is performed on the client via an encapsulated piece of JavaScript. A web-based graphical user interface (GUI) to the SOAlive hosted environment incorporates access to the catalog via its REST APIs. This GUI is illustrated in Fig. 3. Using a general search tab or a application manager or bookmark-specific search tab, a user can start a tag cloud-based drill down discovery of a desired service. Current selections are shown on the right side of the pane, where actions to perform on each selections include showing the service interface as RESTdoc, from which a code snippet can be generated for any operation of any of the service's resources, as shown in Fig. 3.



Artifacts containing tags: bluepages

http://soalive.watson.ibm.com/applications/soalive/soalive.zero.proxy.bluepages
A ProjectZero application that provides a REST interface to IBM BluePages Directory information. Use the employee email to search in the following RESTful manner: e.g. {URL_to_BluePages4Zero}/resources/bluepages/rouvellou@us.ibm.com
owner: soalive

```
"MGRINTERNET":"Rouvellou@us.ibm.com",
}
 */
} else {

alert("Error getting data from the server");
}
}
}


xhr.open("GET", "http://soalive.watson.ibm.com/applications/soalive/soalive.zer
xhr.setRequestHeader('Content-Type', "application/json");
xhr.send(null);

```

Fig. 3. SOAlive Graphical User Interface

The service refinement function is currently under development. The monitor and log portions of this function have already been incorporated into the SOAlive hosted environment, whereas a design of the analyze and refine portions is being completed, an outline of which is presented in section 5.

To illustrate the functions of the SOAlive catalog end to end, we now elaborate on the credit check service example that we first introduced in Section 3. Bob publishes a credit check service to SOAlive with the tag *credit*. This service gets published to the catalog. Users can now discover this service in many ways. A search on the term *credit* or $finance$ (a tag derived from *credit* by the tag consolidation facility, see Section 3) or *composable* (a system-architected tag, see also Section 3) can help users find this service. Jim is composing a mortgage approval service. He finds the credit check service and asks for a code snippet to include in his application. At this point, the credit check service's description is very minimal, and the catalog is able to provide very minimal code. Jim manages to fill in the gaps. He publishes his mortgage approval service to the catalog. The mortgage approval service is a very popular service and so the usage of the credit check service increases. Jane is composing a car loan approval service and is also in need of a credit check service. When Jane finds the same credit check service in the catalog and asks for help with code, she gets invocation code that is complete with error codes, success codes, examples, etc.

7 Related Work

Enterprise service discovery can be thought of as a recommendation system, where the discovery function recommends an initial selection of catalog entries based on a choice of tags. In [16], Zhao et al present a recommendation system based on collaborative tagging behaviors. There, two users are considered similar not only if they rate (or tag) items similarly (i.e., syntactically), but also if they have similar conginitions over these items. For instance if two users tag an item with the tags *photo* and *picture*, respectively, they could be considered similar even if their tags do not match exactly. This idea can also be applied to discovering items (entreprise services in particular) where even if an exact match query on a set of tags yields no results, a similarity-based match may yield a possible result. In this case, the set of tags are selected from the tag cloud. Catalog entries are then retrieved that match not only selected tags but also tags that are similar (via their hypernym/hyponym relationship in particular).

Web 2.0 techniques, such as wiki-based maintenance, are also related to service description. In [10], Paoli et al present an approach to describing (web) services that uses a UDDI registry complemented by a wiki-based semantic annotation subsystem. A developer publishes a service to the UDDI registry and "is encouraged to augment it by intuitive keywords found in the ontology". This in turn results in the generation of a wiki page containing the developer's keywords as well semantic links obtained by automatic reasoning from the ontology. This wiki page can then be used by other developer or business analysts for discovery and understanding. We notice however that this work depends "on an already existing and widely used taxonomy developed for the environmental information system of Baden-Wuerttemberg". We believe that more lightweight Web 2.0 techniques such as social and collaborative filtering are better suited to the cataloguing and discovery of situational enterprise services, given their more dynamic and community-based nature.

Semantic personalization has previously been used in service discovery. Lord et al [7] propose a solution to the task of discovering semantic web services in a Bioinformatics Grid domain. This solution consists of a UDDI registry, augmented by a personalised view service and a semantic find service. The personalised view service provides a way to add user-specific metadata and thus filter the results returned by a query. The semantic find service relies on domain ontologies and a description logic reasoner. The personalised view service can be used in isolation or in combination with the semantic find service. This work also depends on a rigorous, UDDI-based, syntactic description of services. In addition, while an ontology-based semantic description is suitable to this work, given its specific Bioinformatics Grid domain, it is less suitable to the more generic and Web 2.0-based domain of situational enterprise services. We should point out that by allowing the bookmarking and tagging of catalog entries, given its Web 2.0 motivation, the SOAlive service catalog is in effect providing a personalization approach to describing enterprise services.

8 Conclusions

The SOAlive service catalog provides a simplified approach to describing and discovering situational enterprise services. It incorporates support for light-weight description and discovery of enterprise services into the SOAlive community-centric service hosting environment. As we have seen, this incorporation not only improves the functionality of the SOAlive hosted environment but it also enriches the service catalog itself. Specifically, discovery is enhanced by its association with the environment's service deployment function and by the feedback from the service community supported by the hosted environment. Service refinement is enhanced by its integration with the environment's runtime that intercepts service invocations.

The service catalog, by its integration into the SOAlive hosted environment as a number of REST resources, becomes an enterprise service itself, one that is available to the other services hosted by the environment. This has the unintended consequence that services hosted by the SOAlive hosted environment can mash up the function of the catalog and extend its functionality. For instance, a simple enterprise service can provide user information for the developer of a hosted service by invoking the catalog's REST API, looking up the owner of the hosted service, and looking up detailed information for the owner in an enterprise user directory that is registered into the catalog as an external service.

The SOAlive service catalog's content model and code snippet generation function are designed with simplicity and extensibitlity in mind. This should prove useful as we look towards federation with heterogeneous catalogs, as well as towards integrating the catalog with other environments, which may have specific code snippet generation requirements.

References

1. F. Curbera, M. Duftler, R. Khalaf, and D. Lovell. Bite: Workflow Composition for the Web. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC 2007)*, Vienna, Austria, September 2007.

2. Wikipedia definition. Situational application. http://en.wikipedia.org/wiki/Situational_application.
3. N. Desai, P. Mazzoleni, and S. Tai. Service Communities: A Structuring Mechanism for Service-Oriented Business Ecosystems. In *DEST '07: Digital EcoSystems and Technologies Conference*, 2007.
4. Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
5. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
6. Christine Legner. Is there a Market for Web Services? - An Analysis of Web Services Directories. In *Proceedings of Mashups 2007, 1st International Highlightr Workshop on Web APIs and Services Mashups*, Vienna, Austria, September 2007.
7. P. Lord, C. Wroe, R. Stevens, C. Goble, S. Miles, L. Moreau, K. Decker, T. Payne, and J. Papay. Semantic and personalised service discovery. In *Proc UK e-Science All Hands Meeting 2003*, pages 787–794. EPSRC, 2003. ISBN 1-904425-11-9.
8. E. Michael Maximilien, Hernán Wilkinson, Nirmit Desai, and Stefan Tai. A domain-specific language for web apis and services mashups. In *ICSOC*, pages 13–26, 2007.
9. Glen Newton. Drill Clouds for Search Refinement. http://zzzoot.blogspot.com/2007/10/drill-clouds-for-search-refinement-id.html, October 2007. Blog by Glen Newton.
10. Heiko Paoli, Andreas Schmidt, and Peter C. Lockemann. User-driven semantic wiki-based business service description. In *3rd International Conference on Semantic Technologies (I-Semantics 07), Graz*, 2007.
11. ProjectZero. http://www.projectzero.org/. RESTful Documentation: ~/wiki/bin/view/Documentation/CoreDevelopersGuideRESTdoc.
12. WADL Specification. https://wadl.dev.java.net/#spec.
13. SCA Specification. http://www.oasis-opencsa.org/sca/.
14. S. Tai, N. Desai, and P. Mazzoleni. Service communities: Applications and middleware. In *SEM '06: Proceedings of the 6th International Workshop on Software Engineering and Middleware*, pages 17–22, New York, NY, USA, 2006. ACM.
15. Multiple wiki authors. REST for the Rest of Us. http://wiki.opengarden.org/REST/REST_for_the_Rest_of_Us.
16. S. Zhao, N. Du, A. Nauerz, X. Zhang, Q. Yuan, and R. Fu. Improved Recommendation based on Collaborative Tagging Behaviors. In *Proceedings of the International ACM Conference on Intelligent User Interfaces (IUI2008)*, Canary Islands, Spain, 2008.