IADIS International Journal on WWW/Internet
Vol. 12, No. 1, pp. 35-51
ISSN: 1645-7641

DEVELOPING AND INTERACTING WITH DIALOGUE-BASED WEB SERVICES

José Javier Durán. *CETINIA, University Rey Juan Carlos. c/ Tulipán s/n, Móstoles, Madrid, Spain.*

Alberto Fernández. *CETINIA, University Rey Juan Carlos. c/ Tulipán s/n, Móstoles, Madrid, Spain.*

ABSTRACT

In this article we tackle the problem of developing and accessing to dialogue-based applications that require using Web technologies, such as REST services. In particular, we focus on services that are not executed in one-shot (e.g. pipe process), but that require engaging in a dialogue where several messages are exchanged (with the number of messages not known a priori). We propose a protocol for such service interaction, and offer a framework that assists in their life-cycle: programming, deployment, search, invocation, and feedback management. A running example is used to illustrate our proposal and assess how it improves the experience of service developers.

KEYWORDS

Web services, Platform as a Service, Service directory, Middleware

1. INTRODUCTION

HTTP REST[1] services are getting fast adoption in current Web applications because of their simplicity both in programming and integration (Guinard et al. 2012). Despite the fact that developing such services is easy, as well as creating composed services, there is not a standard protocol for deploying dialogue-based services. These kinds of services require an interaction between the user and the server that cannot be expressed using Web service descriptions like WADL, or exchange languages like JSON.

In this paper we present a framework for deployment and use of dialogue-based applications that require using Web technologies, such as REST services. In a dialogue, a service may ask the user for additional information, and next steps depend on the nature of the information supplied. For example, a medical diagnosis service typically requires different

[1] Representational State Transfer

information (e.g. analytic measures) depending on the values of other parameters (symptoms) already analysed. We cover the main issue of such applications: services do not necessarily need to be used only in one-shot workflows. However, others can only be performed using a dialogue workflow. This is the case, for example, of a medical diagnosis service, where it is not necessary to send the whole patient health records but just the requested measure.

The proposed framework covers different needs of an ecosystem for Dialogue-Based Web Services (DBWS), such as service description, registration, invocation and reputation. The main contributions of this paper are an interaction protocol, a middleware for supporting Web services development and a Web interface for searching and invoking Web services.

The rest of the paper is organised as follows. Section 2 summarises other related works. In section 3, we introduce the running example that is used to illustrate the different aspects of our framework. The proposed architecture is presented in section 4. The next three sections are devoted to detail the main contribution of this article: the development support middleware (section 5), the service directory for registration, search and reputation (section 6) and a Web user interface for service invocation (section 7). We finish with conclusions and future works.

2. RELATED WORK

Description languages and transport protocols are important parts of Web services development. There are two main technologies: REST services with JSON payload (mainly described using WADL), and SOAP (as WSDL services). The former is lightweight, easier for developers to understand, and more adaptable. The latter is more widely adopted in industry due to existing standards (WS-*) and tools (Guinard et al., 2012; Pautasso et al., 2008). Deployment environment is another important aspect in the development of Web services. Nowadays industry is moving towards PaaS (Platform as a Service) environments (Lawton 2008) in which different applications are deployed together sharing resources and its highly useful when different applications share a common structure and/or they are used in the same way (e.g. Heroku platform is running more than 3 million applications[2]).

There are different solutions focused on the creation of dynamic interfaces for Web services. Usually, the user interface is created depending on the type of service to use, or the parameters required for its execution. Some of these solutions translate a WSDL description into a Web interface that represents the different kinds of restrictions and input types using HTML widgets (Kopel et al., 2013). Others are focused on testing services by creating requests based on service definitions, but offering an interface more appropriate to software developers (Bartolini et al., 2009). There are other options that integrate both a directory of services with a test user interface for such services, even including options for user feedback. In particular, there are several existing public service directories.

In Table 1 we enumerate the different characteristics that we think should be present in a Web Service directory, and how they are implemented in different solutions. The first characteristic is whether the directory provides *search* capabilities. *Registry* defines whether users can register their own services or the directory is closed. A useful information for selecting services is *reputation*. There are different mechanisms for reputation, such as: rating, users' feedback as comments, or wiki-like in which users can update the description of a

[2] https://blog.heroku.com/archives/2013/4/24/europe-region

service in order to correct any wrong information. By *execution* we mean if it is possible to invoke the service directly from the directory web interface, without needing to develop an ad-hoc application, or if there is specific documentation of that process (e.g. example script, or unitary tests of the service). Finally, *format* represents the kind of services that can be registered (SOAP/WSDL, REST, …).

Table 1. Comparison of different web service directories

	Search	Registry	Reputation	Execution	Format
Membrane SOA registry	No (list)	Yes	Rating	Yes Low-level	SOAP
WS-index.org	Text	No	Rating	No	*Unknown*
API-Hub	Text + Filters	Yes	No	No	Any
Programmable web	Text + Filters	Yes	Rating	No	Any
X Methods	No (list)	Yes	No	No	SOAP
BioCatalogue	Text + Filters + In/Out	Yes	No (wiki)	Examples	SOAP, REST
Embrace	Text	No	Comments	Unitary tests	SOAP, REST, DAS, BioMOBY

Membrane SOA Registry[3] includes a five-star rating system and a (low-level) SOAP invocation user interface, but lacks of a search capability. *WS-index.org* is a directory of web-pages related to web services, but a standard format is not applied to the entries, and most of the entries are out-dated. *API-Hub*[4] and *Programmable Web*[5] focus on API documentation and both offer text and filter-based search. *X Methods*[6] offers a WSDL-only directory, but it lacks of search capabilities and reputation mechanisms. *BioCatalogue*[7] offers a complex search mechanism able to filter by text, tags, and kind of input and/or output, but instead of offering an execution mechanism, it serves as a repository of execution examples. *Embrace*[8] is a specialized directory for medical services (support for domain description formats like DAS and BioMOBY), which offers access to unitary tests that are run in background in order to measure the reliability of the services.

Despite the existence of all those tools, there is a lack of a solution that integrates all the important Web service mediation characteristics together. *Programmable Web* is the most complete regarding those characteristics, but it does not allow execution, which is only supported by *Membrane*. Moreover, they do not provide support for service development.

[3] http://www.service-repository.com/
[4] http://www.apihub.com
[5] http://www.programmableweb.com/
[6] http://www.xmethods.com/
[7] https://www.biocatalogue.org/
[8] http://www.embraceregistry.net/

3. EXAMPLE

In this section we introduce an example to illustrate the process of adapting a specific dialogue-based application to our architecture. We chose a simple application that assists users in deciding what cocktail to make, by asking the user questions about desired ingredients or restrictions (e.g. % alcohol). This application relies on a database of cocktails based on different ingredients that are required in order to mix a new drink. This example has been chosen for its simplicity and the continuous use of a question-answer mechanism for finding the solution. Also, this kind of problem cannot be solved using a questionnaire, as the number of items to check can be very extensive.

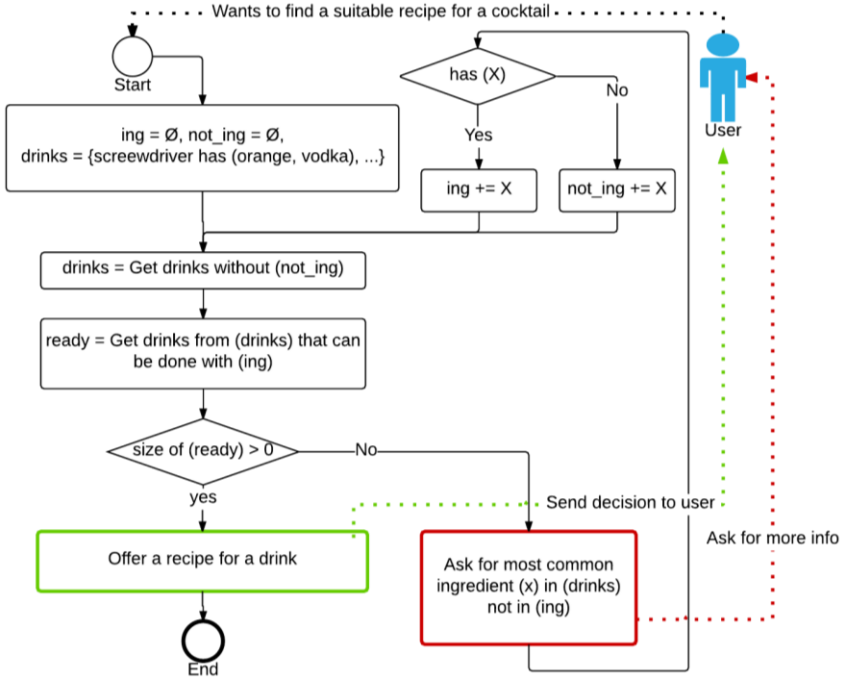

Figure 1. Cocktail advice algorithm

This algorithm (Fig. 1) tries to find a recipe for a cocktail based on a list of ingredients that are required for their making. Cocktails are defined by the ingredients that are needed (amounts are not taken into account). The algorithm manages sets of available and unavailable (or discarded) ingredients. Initially, both sets are empty. Drinks that require the discarded ingredients are filtered out, while the rest form the set of possible drinks. From those candidates, the ones that can be made exclusively with the available ingredients are chosen (*ready*). If any drink can be made at this point, the algorithm finishes, and sends the recipe to the user. Otherwise, the user is asked whether he has the most common ingredient among the candidate drinks. Depending on the user's answer, the ingredient will be appended to the available or discarded list. After this, the algorithm repeats until a drink can be made.

4. ARCHITECTURE

Fig. 2 shows our framework architecture. There are three main components: a development middleware, a service directory and a Web user interface. Also, for clarity of the example, we included in the diagram an additional component (the development stage) that is provided by the developer itself, using whatever tool it is required for the creation of an algorithm, and it is not a part of our architecture.

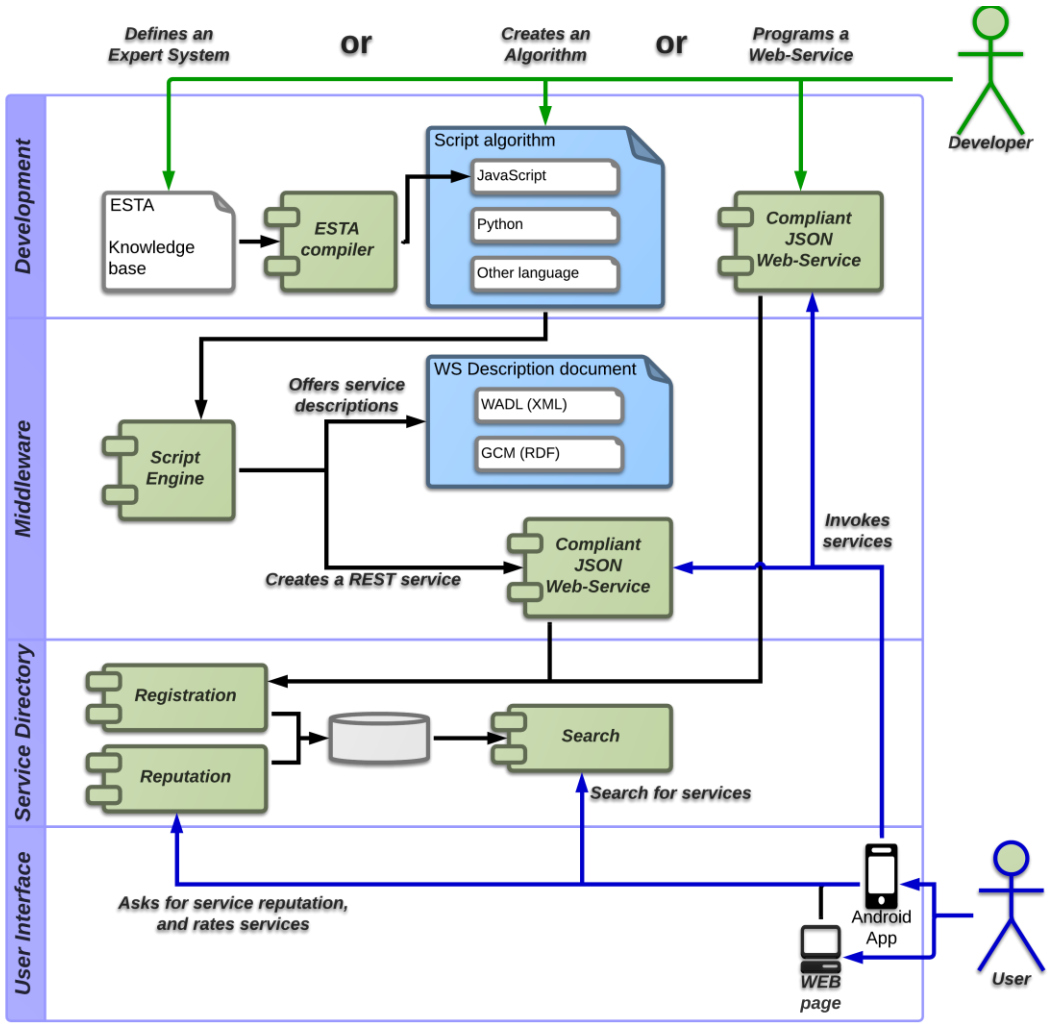

Figure 2. Framework components

The *Development support middleware* is a set of tools that facilitate the development of dialogue-based services. A *Script Engine* takes script code and generates a Web service implementation (*WS*) and its descriptions.

The *Service Directory* acts as a mediator (yellow pages) among services and users. Agents advertise the services they provide by registering with the directory. A service registration includes (i) a *description* of its functionality, (ii) a *grounding* specifying the endpoint where the service can be invoked, and (iii) the agent/organisation that created or owns the service (for reputation management).

The *Web User Interface* is a generic Web application that provides a human interface to search and invoke services registered with the directory, as well as providing feedback about service use.

There are three options developers can choose to use our framework. They can create a web service on their own without using the development support middleware. They may register their services with the directory so as to be found by the users, or they just might not use the directory and somehow provide the users the endpoint. The second option is to write the script that represent the service logic and use the development support middleware to generate the web service and registration. Additionally, the framework includes a compiler to translate ESTA[9] knowledge bases into JavaScript, so the third option is to write the expert system in ESTA and let the framework to do the rest.

5. SERVICE DEVELOPMENT SUPPORT MIDDLEWARE

In order to ease the implementation and integration of Web Services using our framework, we have developed a middleware that deals with process workflow and message exchange. The advantage of this middleware is that it is possible to create a DBWS without implementing any Web functionality, since the communication part is isolated from the application itself. Also, this middleware offers a sandbox environment in which multiple applications can be run together isolated among them, and where errors are properly managed by the middleware.

The main characteristics of the proposed middleware are:

- *Isolate the communication layer from the application*. The middleware is divided in two parts: Transport and Script engine. The transport layer captures Web requests and translates them into a standard *model* object. Then, the script engine is invoked with that object, and the new state of the object is sent back to the requester as a response.
- *Transform Web requests into software objects used by the application.* Whenever a Web request is received the middleware transforms it into an object that contains the value of the parameters used in the dialogue. Also, the application can register parameters and their constraints (used in the dialogue). An advantage of this isolation is the possibility to create unitary tests without requiring access to the middleware.
- *Do not impose a programming language, or paradigm*. The script engine used is the standardized Java script engine (JSR-223). This script engine lets the developer include new parsers, while keeping a common API. We only require access to method invocation with a unique argument, the object *model*.
- *Avoid the use of special structures, or patterns, for dialogue management*. Thanks to the *model* object, applications do not require to apply a special pattern for dialogue management. Whenever an application requires access to a parameter, the script engine marks the parameter as missing and stops the execution of the script until the

[9] Expert System Shell for Text Animation

requester provides a value. This way of dealing with dialogue is similar to lazy programming, in the way that parameters are asked only when they are really needed by the program.

5.1 Interaction Protocol

In this section we describe a workflow process for dialogue-based services, and a format for message exchange.

5.1.1 Workflow

In order to use dialogue-based services, a record of the interaction has to be kept. Services could be invoked in two states: initialisation and resume. During initialisation a service communicates to the client which parameters must be provided. During resume, the service takes the parameters received and returns a message that may include additional information (parameters) required to continue the execution or the result. The message content is explained in next section.

Fig. 3 shows the interactions involved during a cocktail drinks' assistance. Green lines represent user to service messages, blue lines responses shown using the user interface, while other lines represent software level communication.

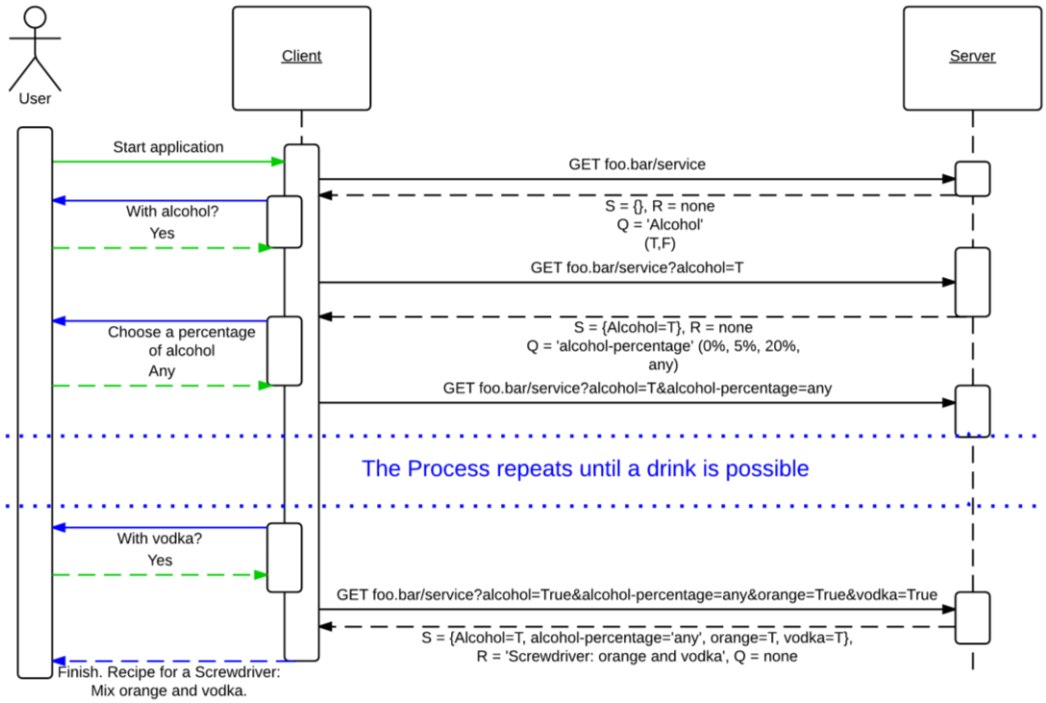

Figure 3. HTTP message exchange between a web client and the service

First, the server asks if the user wants to use alcoholic drinks. The user answers *Yes*. Then, the server tries to specify a limit for alcohol graduation in the drink (*an enumeration*). The user answers *'any'*. For clarity, we simplified the question field, omitting their description. Then, the service asks whether the user wants it with some orange (*true/false*) and the user answered *true* (this is not shown in the figure). Next, Vodka is offered as a possible ingredient (*true/false*), and the user agrees (*true*). As a result, a cocktail is found in the knowledge base, and the service closes the dialogue with the recipe as a message with no further values to be provided.

5.1.2 Message Format

Messages are quadruples <*S, R, Q, P*>, with:

- *S = State information.* This part includes a set of variables, with their current values, representing the service state. This information is used when interacting with stateless services and must be sent to the service again in order to keep a track of the dialogue. Examples of information include parameters asked, internal variables, session id, or a combination of them. They might not be important for the client, except that they must be included in subsequent requests.
- *R = Responses*. This is a set of messages that are sent to the client for its use. Each message can be, for example, a text, an HTML document, a picture, or an RDF document. Those messages are considered the output of the service. In general, those responses are expected to change in reply to subsequent requests, although it is not mandatory (e.g. the response is like an RDF document with additional triples).
- *Q = Question*. When a service requires more information, or asks the user to wait for a time condition to be reached, a question is sent to the client. That question has a textual condition (the *question*), a *motivation* (why it is needed, and/or some semantic information about the question), a *parameter name (id)* (used to send back a client response), and a rule of accepted values (*combination of type and values*) (e.g. an integer a range, an enumeration of possible values, or a class/type like a date). If that field is missing the dialogue is considered finished as the system does not need any further interaction with the client.
- *P = Properties*. As services are defined to be stateless, it must be possible to infer all the information of the algorithm in any state of the request. In our case, we decided to represent description, name and language of the service in this field. This information is used by user interfaces, and by the service directory.

We use JSON serialisation in our framework. Fig. 4 shows an example of message.

```
    {
      state_info: {
        state: "main",
        alcohol: "true",
        alcohol-percentage: "any-percentage",
        diary-products: "true",
        wine(red): "false"  },
      response: [
        ”Still requiring more info for a suitable recipe”
      ],
      question: {
        id: ”vodka”,
        question: ”Would you like to use vodka in the drink?”,
        motivation: ”http://dbpedia.org/resource/Vodka”,
        type: ”boolean”,
        values : null
      }
      properties: {
        description: "This algorithm has a database of drink recipes, and searches for a
suitable recipe
                      depending the available ingredients, asking one by one.”,
        name: "Cocktail advice",
        language: "en-us"
      }
    }
```

Figure 4. Example of a message sent by the Web service

5.2 Script Engine

The script engine relies on the implementation of the JSR-223[10] API present in the Java runtime. This API is capable of loading applications created in different script languages, offering an abstraction of the communication between Java classes and script applications. The advantage of this approach is that it is possible to access applications independently of the programming language as long as a parser for that language is available. In our case, we require the presence of a *setup* method, and specifying the initial method. Both methods must accept as unique argument a *model* object.

The *model* object offers a proxy between the middleware and the script. There are different methods used to connect the script with the dialogue process, which are shown in Table 2.

Table 2. Methods of object *model*

Method	Description
name	a descriptive name for the application
language	the default language used in the messages
initialState	establishes which method must be invoked to process the information received
registerInput	parameters are registered in the setup method with id, question, motivation, and type (e.g.: boolean, string, enumeration)
get	used to access to the dialogue parameters. If a parameter is missing the execution is stopped and a response is sent to the requester asking for a value
info	used to send information to the user (*response* message field)
setState	it changes the method that will be invoked in next requests, for example to continue

[10] http://www.jcp.org/en/jsr/detail?id=223

	the dialogue process from a different method instead of the initial one
set	it assigns values to dialogue parameters. It may be used to keep a trace of a dialogue status when changing state. For example, the service asks (in its initial state) the requester's birthday and, if he is minor, that age is sent to another state focused on children
get_safe	Same as *get* method, but if a variable is not set, it returns a *null* value. Useful when decisions are made depending if a variable has been set or not, or if the algorithm must not stop at the moment.

An example illustrates the use of the *model* object in Fig. 5.

```
    function setup(domain) {
         domain.name = "Cocktail advice Algorithm";
         domain.language = "en-us";
         domain.initialState = "main";
         i = 0;
         // All ingredients are set
         for (;i < ingredients.length; i++){
                  ing = ingridients[i];
                  domain.registerInput(ing.name, ing.question, ing.dbpedia_url, ing.type,
ing.values);
         }
    }

    function main(env) {
         ingredients = {};
         i = 0;
         // Are ingredients available? (true, false, null=unknown)
         for (;i < ingredients.length; i++){
                  name = ingredients[i].name
                  // env.get_safe never stops the execution of the algorithm,
                  //    and assumes null as default value
                  ingredients[i] = env.get_safe(name)
         }
         // candidate: if no ingredient is set to false
         candidates = can_be_made_with(ingredients);
         // ready: if no ingredient is set to null
         ready = not_discarded(candidates, ingredients);
         if (ready.length > 0){
                  // end condition
                  env.info('Recipes found.');
                  env.info(ready);
                  return
         }
         // env.get of an unset variable reset the algorithm, until the user sets the
variable
         env.get( most_common_null_ingredient(candidates, ingredients) );
         return;
    }
```

Figure 5. Partial script of the cocktail algorithm

The Script Engine accepts different programming languages. In particular all JSR-223 compliant languages can be used, such as Java, JavaScript, Python, Scheme, Ruby, Lua, PROLOG, etc. In addition, we have built a compiler that transforms ESTA knowledge bases into JavaScript compliant with our middleware. ESTA is a rule-based language used to build Decision Support Systems (DSS).

The Script Engine also generates service descriptions. In particular, WADL and GCM descriptions are generated. GCM (*General Common Model*) is a unified language, which was used to integrate in a single framework different existing syntactic and semantic service description languages (OWL-S, WSMO, WSDL, SAWSDL, etc.). *GCM* descriptions include typical elements found in existing service description models such as inputs, outputs, preconditions, effects, category, keywords, tag cloud (weighted keywords) and text. Details about the *GCM* language can be found in (Fernández et al. 2012).

6. SERVICE DIRECTORY

The service directory is a key element to coordinate providers and clients. It keeps a database of service descriptions and provides the following functionalities:

- *Registration*: provider agents can advertise their services by providing a description of their functionality and access point.
- *Search*: client agents can query the directory to get a list of services matching the required functionality. Matching services are attached with a reputation value, which is obtained by a reputation module included in the directory.
- *Feedback*: clients can provide feedback to the directory by evaluating their experience with the services they have interacted.

In the rest of this section we detail those aspects.

6.1 Service Registration

Registration of services in the service directory module is done by using the information attached to the messages used in the protocol. Within those messages, there is meta-information about the service, like name, description, and language. In order to register a service, it is only required a URL to that service, as the directory will query the service itself the required information.

Once the service information is read, the directory will extract the most important keywords of its description, and measure their importance in the text. These weighted keywords will be used later in the search mechanism. Also, and attending to the web domain, services are clustered to be able to use reputation of related services when there is not enough reputation information available. For the keyword extraction, we rely on *AlchemyAPI Keyword Extraction API*[11].

6.2 Service Search

While *service registration* is used by service providers wishing to advertise their services, service search is a functionality for users that are looking for services.

Service matchmaking is based on the similarity of two texts (description of a service, and text query) based on its weighted keywords. For each service in the directory we measure their relevance with the text query, attending to the next equations:

[11] http://www.alchemyapi.com/products/alchemylanguage/keyword-extraction/

$$relevance(Service, Query) = ssim(Service, Query) * rep(Service)$$

$$ssim(Service, Query) = \frac{\sum_{k_s \in Service} weight(k_s, Service) * weight(k_s, Query)}{\sum_{k_q \in Query} weight(k_q, Query)}$$

$$weight(Keyword, Text) \in [0,1]$$

where *rep(Service)* represents the reputation as detailed in next section.

Then, services that are above an established threshold are offered to the user ordered by their score. In our case we offer a specific interface (Fig. 6) that displays both the service similarity with the query, as its rating (in our case, based on its reputation).

Figure 6. User interface for the web service directory

6.3 Reputation

Our framework includes a reputation module. Users can provide feedback to the system by evaluating their experiences with services. When a service search is launched the relevant set obtained by the *service search* module is attached with a reputation value ([0..1]) of each service based on past experiences.

We use a simplification of the reputation mechanism proposed by (Hermoso et al., 2006) for task oriented multi-agent systems. They propose a trust model for Virtual Organisations where agents play some roles in different interactions. In our case we have only two components, namely agents (organisations) and services. Our resulting reputation mechanism is as follows.

The reputation of a given service (*rep(s)*) is updated whenever a new evaluation (*eval(s)*) is obtained:

$$rep(s) = \alpha \cdot rep'(s) + (1 - \alpha) \cdot eval(s)$$

where *rep'* is the reputation value previous to the update and $\alpha \in [0..1]$ is a parameter (empirically adjusted) specifying the importance of the past reputation value. It may happen that a matching service has not been sufficiently evaluated (zero or very few evaluations). In those cases we take into account the reputation of other services run by the same agent following Hermoso's approach.

The recommendation value of a service *s* is given by the following equation.

$$rec(s) = \begin{cases} rep(s) & \text{if } r(s) \geq \theta \\ \dfrac{\sum_i rep(s_i) \cdot w(s, s_i)}{\sum_i w(s, s_i)} & \text{otherwise} \end{cases}$$

where *r(s)* measures how reliable *rec(s)* is, and $\theta \in [0..1]$ is a threshold. Reliability is based on the mean and standard deviation of user votes, assuming a Gaussian probability distribution. In case that the direct reputation is not reliable enough the value is calculated as a weighted sum of the reputation of similar services provided by the same organisation:

$$w(s, s') = \begin{cases} r(s') \cdot ssim(s, s') & \text{if } O(s) = O(s') \\ 0 & \text{otherwise} \end{cases}$$

where *O(s)* is the organisation that provides service s, and *ssim(s, s')* is service similarity (section 6.2).

7. WEB INTERFACE FOR WEB SERVICE INVOCATION

Since our framework defines a common interface for multiple services (the message protocol) it is possible to reuse a user interface to access different services. In our case, we have developed a user interface that covers the main aspects of our proposal: search, invocation, and feedback.

The web interface has been designed to be used in different devices like computers, tablets or smart phones.

7.1 Service Search

The user interface is presented through the list of services that are accessible by the server. This initial page lets users search for services attending to a free text query that uses the service directory component. Results are shown ordered by their relevance (rating) and, besides their name, their languages, reputations and descriptions are shown. It is possible to choose which information is shown. When the user selects a service, it is invoked, as described in the next section.

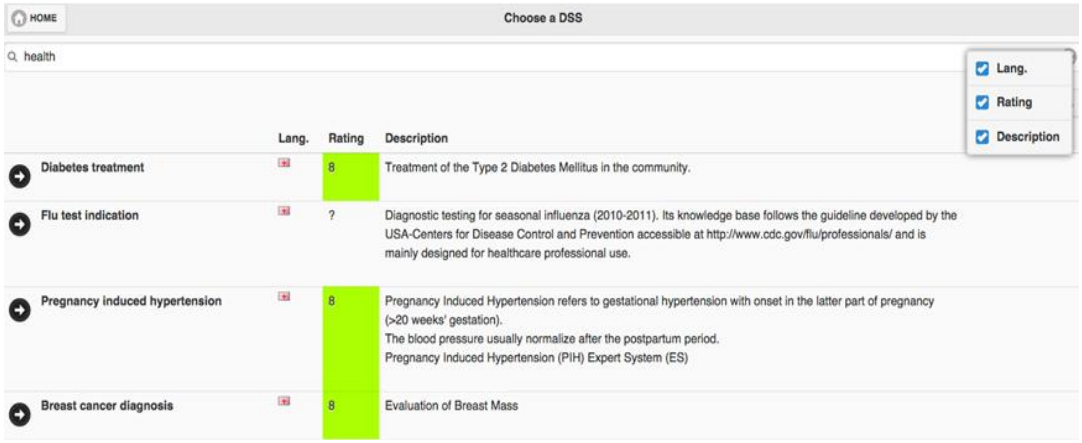

Figure 7. Web Service interface for the service directory

7.2 Service Invocation

The proposed protocol includes information needed for a dialogue stage, i.e. parameter required (question field) and response messages. The user interface (mobile UI in Fig. 8, desktop UI in Fig. 9) shows the response messages followed by the parameter question and by a log of previous responses in the dialogue. The parameter question contains two elements: the parameter question (enriched with motivation information) and the input field. The latter is created with the most appropriate HTML input, e.g. for a boolean or small enumeration a button for each option is shown, for long enumerations a drop list is used, etc. Some of the information available through mobile app is shown in Fig. 8.

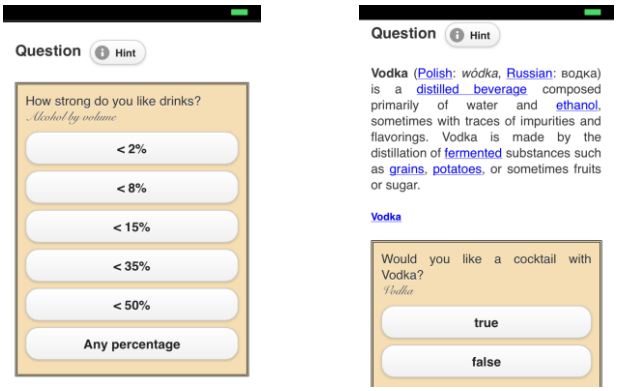

Figure 8. Mobile User Interface. Two different questions, the second with additional information gathered using the motivation field of the question.

7.3 Service Feedback

During the invocation process, the current reputation score is shown, and the user can submit a feedback about the service. The feedback can include a score, a text about the user's experience and the dialogue log (e.g. for debugging). Fig. 9 shows a snippet of the Web user interface. The panel on the right allows the user to rate the service as well as download the log and service descriptions.

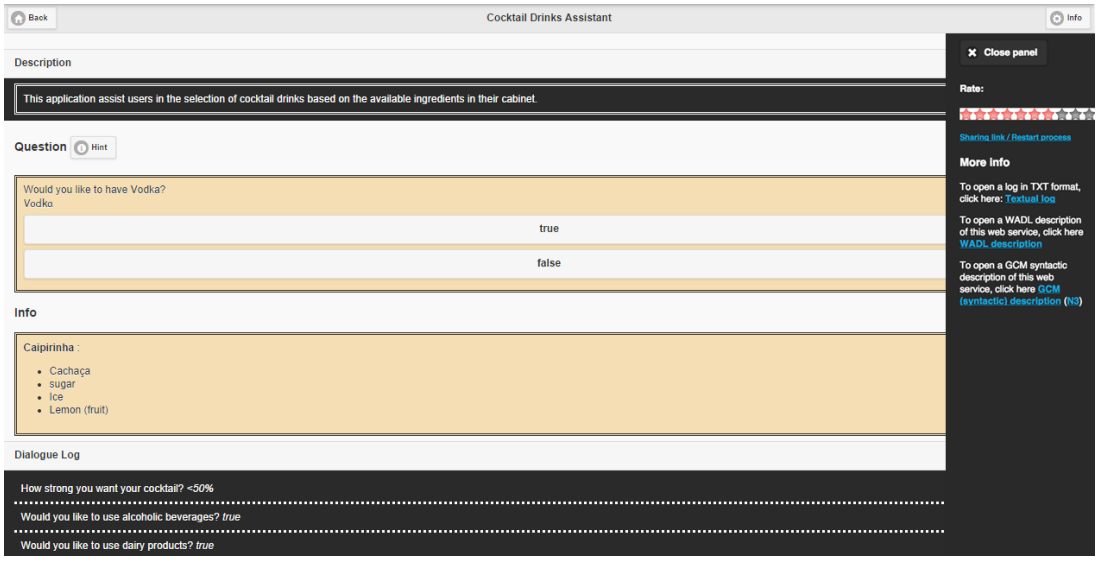

Figure 9. Desktop Web User Interface for the application

8. CONCLUSION

In this paper we have described a framework for developing and interacting with Dialog-Based Web Services. We have presented a service directory that manages service advertisement and search, extended with a reputation mechanism to take into account user's feedbacks. We have proposed a protocol for interacting with this kind of services. In order to support service construction we have developed a middleware that generates web services from scripting languages. We also have developed a generic Web interface to invoke such services using our framework, from either desktop computers, or mobile devices.

Web service developers and users can benefit from the proposed framework in different ways, using all or part of its functionality. (i) Developers can implement web services using the techniques they prefer and use only the directory functionality by registering their services. Alternatively, (ii) developers can implement the functionality of their services using a scripting language and the tool generates the Web services and directory registrations. In addition, (iii) developers (maybe not experienced programmers) can write an ESTA knowledge base and our compiler generates the script that can be used in (ii). Finally, the Web Interface is a tool that allows users to (iv) search services and/or (v) invoke them if wanted.

We have implemented a prototype to assist clinicians in their diagnosis. The system integrates knowledge-based medical decision support systems. Those systems are programmed in ESTA expert system, and its integration in our framework has been straightforward. We use the user interface presented in this paper to test that system.

In the future, we also plan to extend our approach to deal with asynchronous services, i.e. services that can pause their execution and resume it later (e.g. a diagnosis service requires blood analysis tests). Web service composition is another open issue we plan to tackle. We will also work on the automatic generation of web services provided by humans, i.e. they are not implemented from a script describing the algorithm but a human user (e.g. an expert) is on the server side using a Web based interface to provide the service.

ACKNOWLEDGEMENT

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness through grants CSD2007-00022 ("Agreement Technologies", CONSOLIDER-INGENIO2010), TIN2009-13839-C03-02 ("OVAMAH", co-funded by Plan E), TIN2012-36586-C03-02 ("iHAS") and RTC-2014-1850-4 ("SmartDelivery") as well as by the Autonomous Region of Madrid through grant P2013/ICE-3019 ("MOSI-AGIL-CM", co-funded by EU Structural Funds FSE and FEDER").

REFERENCES

Bartolini, C. et al, 2009 Ws-taxi: A wsdl-based testing tool for web services. *In Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pp 326–335.

Fernandez, A. et al, 2012. Bridging the gap between service description models in service matchmaking. *Multiagent and Grid Systems*, Vol. 8, No. 1, pp 83–103.

Guinard, D. et al, 2012. In search of an internet of things service architecture: REST or WS-*? a developers' perspective. *In Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pp 326–337.

Hermoso, R. et al, 2006. Effective use of organisational abstractions for confidence models. *Proceedings of the 7th international conference on Engineering societies in the agents world*, pp 368–383.

Kopel, M. et al, 2013 Automatic web-based user interface delivery for soa-based systems. In *Computational Collective Intelligence. Technologies and Applications, volume 8083 of Lecture Notes in Computer Science*, pp 110–119.

Lawton, G. 2008. Developing software online with platform-as-a-service technology. *Computer*, Vol. 41,No. 6,pp 13–15.

Pautasso, C. et al, 2008 Rest- ful web services vs. "big" web services: Making the right architectural decision. *In Proceedings of the 17th International Conference on World Wide Web, WWW '08*, New York, USA, pp 805–814.