Knowledge Base documentation

FromGold

November 2018

1 Quick Overview

This KB implementation is in typescript and offers a typescript library and a websocket server interface on node.js What follows is the list of components of our module:

src/kb.ts is exporting the typescript module of the knowledge base implementation.

src/matcher.ts contains the implementation of the module dealing with the matching of JSON objects.

src/server.ts is the websocket server that can run on node.js

src/inferenceStub.ts is a rough implementation of an inference engine with simple plain rules

test/ contains some function call to test the *kb.ts* and *matcher.ts* code

bindings/ contains websocket bindings, currently supported only for Python 3.6

cli/ contains command line scripts to interact with *kb.py*

package.json is the npm configuration file

tsconfig.json is the typescript compiler configuration file

tslint.json is a configuration file for a typescript linter

requirements.txt required packages to run the python binding

2 Webserver requests

All the requests sent to the webserver have to be made through a message consisting of a JSON file with the following properties:

method : string. A string containing the name of the method you want the KB to execute. Currently available methods are:

- **addFact**
- **addRule**
- **getAllTags**
- **getTagDetails**
- **query**
- **register**
- **registerTags**
- **removeFact**
- **removeRule**
- **subscribe**
- **updateFactById**

The first method the you want to call is **register**, which will return the *idSource* you will need to register tags and adding facts to the KB.

params : a JSON object whose keys are the name of the parameters required by the method, and whose values are the corresponding values.

token : a string used for security purposes. Its value has to be set to 'smartapp1819'

An example of a request could be:

```
{
    "method" : "addFact",
    "params":
    {
        "idSource" : myIdSource,
        "tag" : myTag,
        "TTL" : myTTL,
        "reliability" : 100,
        "jsonFact" : myJsonFact
    },
    "token": "smartapp1819"
}
```

Each of the methods will return a Response object defined as follows:

```
{
    "success" : boolean
    "details" : any
}
```

Success is a boolean specifying whether the operation was successful, while details is a property whose type depends from the method being invocated. Its type will be specified for each method in the next section.

Moreover, we wish you to be aware of the format we use for storing the facts in the KB, so that you can use it to query the data you need such as metadata. The format is the following:

```
{
          _id : integer ,
          _meta :
          {
                    idSource: string;
                    tag: string;
                    TTL: integer;
                    reliability: integer;
                    creationTime: Date;
          },
          _data : json object
}
```

The creationTime is a JavaScript Date object. It contains properties such as *year* and *month* that you can query if you're interested in retrieving a fact through its creationTime. A complete documentation of the Date object can be found here:

`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date`

3 KB methods

In this section we will list the currently available KB methods, parameters and semantics.

1. **ADDFACT**

   NAME
   addFact - add a fact to the KB

   SYNOPSIS
   addFact(string idSource, string tag, int TTL, int reliability, JSON jsonFact)

   DESCRIPTION
   The *addFact* function adds a new fact to the Knowledge Base, provided that the client has already registered itself through *register* and the corresponding tag through a call to *registerTags*. If so, the client has to provide, for the submitted jsonFact, also the desired TTL in hours and its reliability expressed as an integer from 1 to 100, which is how much the information has to be trusted. idSource is the string identifier returned by the *register* function. The jsonFact can have

any fields, but it's good practice to use always the same structure for objects having the same tag, and to specify this structure during the registration.

RETURN VALUE

The *getTagDetails* function returns a JSON object containing the following fields:

success : boolean value. Success will evaluate to false if either id-Souce is not a correct id, or the tag had not been registered yet. Otherwise, it will evaluate to true and the fact will be correctly added.

details : integer if success == true. Otherwise string or JSON object.
In case success is true, details will contain the idFact for the newly added fact. If the tag was not registered, it will be a string containing the non-registered tag. Otherwise, if idSource was not correct, it will be an empty JSON object.

EXAMPLE

Example of usage:

```
addFact('myid', 'rdf', 7, 100,
{
    relation: 'follows',
    subject: 'Michael FromGold',
    object: 'SmartApplication'
})
```

Example of return value:

```
{'success' : true, 'details' : 3}
```

2. **ADDRULE**

NAME

adds a rule to the KB

SYNOPSIS

addRule(string idSource, string ruleTag, string jsonRule)

DESCRIPTION

The *addRule* function adds a rule to the KB. It takes as parameters the idSource, a string rule tag and a string jsonRule made in the following way:

```
{head} <- {clause1} ; {clause2} ; {clauseN}; [pred1] [predN]"
```

Where **head** and **clause(s)** are json-like strings containing, respectively, the consequence of the rule and the N pre-conditions (clauses).

On the other hand, each **predk** is an array in the following form:

```
[pred] = [predicate, [predicateParams1], [predicateParams2]]
```

where **predicate** is one of the available predicates fully documented in Section 3.1. Semantically, a new fact matching with the **head** will be added on the KB if each of the clauses are satisfied. In addition to the explicit clauses, one can use predefined predicates to impose other clauses over the matches found for the explicit ones. The new added fact will have tag ruleTag. For example, the following rule will add a fact *professorOver50* : *$x* for each *$x* matching the two specified clauses and the predicate *isGreater*: foo es. greaterThan params 1 2 perché posso chiamare isGreater più volte con parametri diversi. Occhio che il placeholder deve matchare esattamente $age con $age

```
'{"professorOver50": "$x"} <-
 {"subject" : "$x", "relation" : "is a", "object": "professor"}
 {"subject" : "$x", "relation" : "age", "object": "$age"}
 ["isGreater", ["$age", "50"]] '
```

isGreater takes two parameters, that are the numbers to be compared.

RETURN VALUE

The *addRule* function returns a JSON object containing the following fields:

success : boolean value. Always true.

details : JSON object. Details will contain the id of the new inserted rule.

3. **GETALLTAGS**

NAME

getAllTags - get information about all registered tags of the KB

SYNOPSIS

getAllTags()

DESCRIPTION

The *getAllTags* function retrieves every tag that has been registered by any user and returns them grouped by user that registered it.

RETURN VALUE

The *getTagDetails* function returns a JSON object containing the following fields:

success : boolean value. Always true.

details : JSON object. Details will be an object whose keys are the idSource of the clients, and whose values are values are the registrated tags:

```
{idSource0 : tags0, idSource1 : tags1}
```

where tags are in the form:

```
{tag0 : {desc: "description0", doc: "documentation0"},
 tag1 : {desc: "description1", doc: "documentation1"}}
```

EXAMPLE

Example of usage:

```
getAllTags()
```

Example of return value:

```
{'id0' :
    {tag1 : {desc: 'desc1', doc: 'doc1'},
     tag2 : {desc: 'desc2', doc: 'doc2'}}
 'id1' :
    {tag3 : {desc: 'desc3', doc: 'doc3'}}
}
```

4. **GETTAGDETAILS**

NAME

getTagDetails - get the details for the given tags

SYNOPSIS

getTagDetails(string idSource, JSON jreq) The jreq parameter has to be in the following form:

```
jreq = {['tag1', 'tag2', 'tag3', ...]}
```

DESCRIPTION

The *getTagDetails* function retrieves the tag details previously provided by the user identified by idSource through a call to *registerTags.*Tags has to be provided as a string array containing the tags whose documentation has to be retrieved. In case some of them are not registered, they will be ignored. The operation will fail only when none of the input tags had been registered.

RETURN VALUE

The *getTagDetails* function returns a JSON object containing the following fields:

success : boolean value. Success will be true if any of the provided tags exists. Otherwise its value will be false.

details : JSON object. In case success is true, details will be a JSON object containing couples tag: details, where details is a json desc: 'desc', doc: 'doc'. Otherwise, details will be an empty objects.

EXAMPLE

Example of usage:

```
getTagDetails(['emo', 'crawl'])
```

Example of return value:

```
{'success' : true, 'details':
    {
        'emo': {'desc': 'emotion', doc : 'emotion_doc'},
        'crawl': {'crawler': 'crawler_doc'}
    }
}
```

5. **QUERY**

NAME
query - query the KB and retrieve the matching facts and the corresponding bindings

SYNOPSIS
query(JSON jreq)

DESCRIPTION
The *queryFact* function will query the KB and find fact matching the provided jreq. If you need to match the metadata, you can do so by addressing the '_meta' field of the jreq. The following is an example of a jreq used to query the KB for all the facts with tag 'NLP':

```
{_meta : {'tag' : 'NLP'}}
```

Alternatively, one can query the KB for a specific fact by knowing its id, previously returned by a call to *addFact*, by filling the '_id' field of the jreq, as follows:

```
{_id : 'my_fact_id'}
```

RETURN VALUE
The *query* function returns a JSON object containing the following fields:

success : boolean value. Success will be true if any facts matching the query have been retrieved.

details : array of JSON objects or JSON object. If success is true, details will be an array of JSON objects with the following fields:

- **object** : a JSON object matching the query. It will contain, as usual, the properties _data, _metadata and _id.
- **binds** : an array of the bindings found. If any of the values of the query have been defined with a wildcard value (e.g. a value starting with $), this array will contain all of them.

EXAMPLE
Example of usage:

```
query({"subj":"$sub"})
```

Example of return value:

```
{"success": True, "details":
[{"object": {
        "_data": {"relation": "is", "subj": "Pollo", "obj": "student"},
        "_id": 2,
        "_meta": {"idSource": "id0", "tag": "nlp",
        "creationTime": "whatever", "TTL": "100", "reliability": "1"}
    },
"binds": [{"$sub": "Pollo"}]
},
{"object": {
        "_data": {"relation": "is", "subj": "Fromme", "obj": "student"},
        "_id": 3,
        "_meta": {"idSource": "id0", "tag": "nlp",
        "creationTime": "time", "TTL": "100", "reliability": "1"}
    },
"binds": [{"$sub": "Fromme"}]
}]}
```

6. **REGISTER**

NAME
register - register to the knowledge base

SYNOPSIS
register()

DESCRIPTION
The *register* function registers the client to the KB and returns a string id. Registering is mandatory for any call to *registerTags* and *addFact*, and so it should be called as first thing when interacting with the KB.

RETURN VALUE
The *register* function returns a JSON object containing the following fields:

success : boolean value. Success evaluates to true.

details : JSON object. Details will contain your personal idSource.

EXAMPLE
Example of usage:

```
register()
```

Example of return value:

```
{'success' : true, 'details': 'id5'}
```

7. **REGISTERTAGS**

NAME

registerTags - register tags and corresponding documentation in the knowledge base

SYNOPSIS

registerTags(string idSource, JSON jreq)

The jreq parameter must have the following form:

```
{'tag1' : {desc : 'desc1', doc : 'doc1'},
 'tag2' : {desc : 'desc2', doc : 'doc2'}, ...}
```

DESCRIPTION

The *registerTags* function registers the provided tags in the knowledge base, but only if the user has previously registered through a call to *register*. Tags has to be given as a JSON object whose keys are the tags to be registered, and whose values are JSON objects containing a short tag description and a complete tag documentation. In case some of the given tags have already been submitted by the same user, they will be overwritten.

RETURN VALUE

The *registerTags* function returns a JSON object containing the following fields:

success : boolean value. Success evaluates to false only if idSource is not a correct id returned by a call to *register*.

details : string array or JSON object. If success is true, details will be an array containing the registered tags. Otherwise, details will be an empty object.

EXAMPLE

Example of usage:

```
registerTags('myid',
{'emo': {'desc': 'emotion', 'doc': 'emotion_doc'},
'crawl': {'desc': 'crawler', 'doc': 'crawler_doc'}})
```

Example of return value:

```
{'success' : true, 'details': ['tag1', 'tag2', 'tag3']}
```

8. **REMOVEFACT**

NAME

removeFact - remove a fact from the KB

SYNOPSIS

removeFact(string idSource, JSON jreq)

DESCRIPTION

The *removeFact* method will find and remove facts matching the jreq object. The latter, to match the data present on the KB, must have at least one of the following fields:

_id : used to identify a fact given an idFact returned by a call to *addFact*. For an exact removal of a fact, use only this field.

_metadata : metadata of the fact. It can contain the properties 'idSource', 'tag', 'TTL', 'creationTime' and 'reliability'. Any of them can be left out if not needed for the match. For example, the following call will remove all the facts with tag 'tag1':

```
removeFact('myid', { '_meta' : {'tag': 'tag1'}})
```

_data : actual data submitted through a call to *addFact*

RETURN VALUE

The *removeFact* function returns a JSON object containing the following fields:

success : boolean value. Success will evaluate to true if any facts have been deleted. In case no facts match the jreq object, it will be false.

details : integer array. If any fact has been matched and removed, details will contain their ids listed in an array. Otherwise, an error string will be returned.

EXAMPLE

Example of usage:

```
removeFact('_meta': {'idSource': myIdSource, 'tag': 'nlp'})
```

Example of return value:

```
{'success' : true, 'details' : [43, 12, 542]}
```

9. **REMOVERULE**

NAME

SYNOPSIS

DESCRIPTION

RETURN VALUE

The *removeRule* function returns a JSON object containing the following fields:

success : boolean value.

details :

EXAMPLE

Example of usage:

Example of return value:

10. **SUBSCRIBE**

    NAME

    SYNOPSIS

    DESCRIPTION

    RETURN VALUE

    The *subscribe* function returns a JSON object containing the following fields:

    success : boolean value.

    details :

    EXAMPLE

    Example of usage:

    Example of return value:

11. **UPDATEFACTBYID**

    NAME

    updateFactByID - update an already added fact by its id.

    SYNOPSIS

    updateFact(string idSource, int id, string tag, int TTL, int reliability, JSON jsonFact)

    DESCRIPTION

    The *updateFact* function updates a fact given an id previously returned by a call to *addFact*. Both data and metadata will be updated with the freshly provided ones.

    RETURN VALUE

    The *updateFactById* function returns a JSON object containing the following fields:

    success : boolean value. success will evaluate to true if idSource is a correct id returned by *register*, and if the KB contains a fact identified by id.

details : integer or JSON object. In case success is true, then details will be the id of the correctly updated fact. Otherwise, if no facts are identified by id, details will be the missing tag; finally, if idSource is not a valid user id, details will be an empty JSON object.

EXAMPLE

Example of usage:

```
updateFact(myid, 412, 'rdf', 7, 100,
{'relation': 'follows',
'subject': 'Ferrante Francesco', 'object': 'SmartApplications' }
```

Example of return value:

```
{'success' : true, 'details' : 412}
```

3.1 Predicates

What follows is a list of the currently available predicates, divided in semantic groups:

1. NUMERIC PREDICATES

   isEqual (integer op1, integer op2)

   isGreater (integer op1, integer op2)

   isGreaterEqual (integer op1, integer op2)

   isLess (integer op1, integer op2)

   isLessEqual (integer op1, integer op2)

2. DATE PREDICATES

   isEqualDate (string op1, string op2)

   isAfterDate (string op1, string op2)

   isBeforeDate (string op1, string op2)

3. STRING PREDICATES

   isEqualString (string op1, string op2)

   isGreaterString (string op1, string op2)

   isLessString (string op1, string op2)

The semantic is very intuitive. As an example, *isGreater*("0", "5") returns false.

Note that even though the parameter type may semantically not be string (as in *isGreater*), the clients calling addRule are nonetheless supposed to write every parameter as string.

4 Python Bindings

In case you're using Python, we provide some bindings that let you create a connection to the webserver and interact with the KB straight away.
First of all run the command **pip install -r requirements.txt** in the *SmartApp.KB/* folder to install the python required packages.

The bindings file is located under the path *SmartApp.KB/bindings/python/kb.py*. It is sufficient to import the class **KnowledgeBaseClient**, contained inside kb.py, in your python code, create an object and call its methods.

Note: **KnowledgeBaseClient**'s constructor takes in input a boolean that lets you specify whether the connection should be persistent or not. By default the connection is persistent.

5 How to run

In case you want to run the KB on your local machine, open a terminal in the *SmartApp.KB/* directory and issue the following commands:

- npm install
- npm start

npm install will install the packages needed to execute; **npm start** will start the server on port 5666. In case you lack the node packet manager, you can install it with sudo apt install npm node-typescript