DELIVERABLE

Project Acronym: **F4W**

Grant Agreement nbr: **000000**

Project Title: **FACTS4WORKERS**

RESTdesc demo use case

Revision:

Authors:

Joachim Van Herwegen (iMinds – Ugent – Multimedia Lab)

REVISION HISTORY

Revision	Date	Author	Organisation	Description

Table of Contents

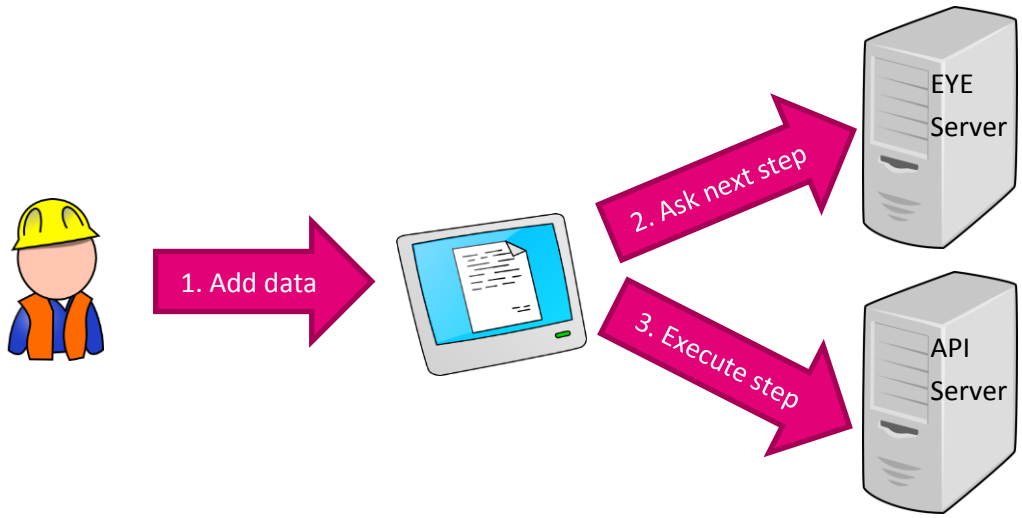
1 Introduction

This demo showcases how RESTdesc can be used in the context of FACTS4WORKERS. The demo is a basic implementation of a partial use case, as discussed during previous meetings. Although this is an incomplete use case, it is enough to show how RESTdesc works and how it can be used to describe APIs and their usage in any use case.

2 Use case

The use case is the following. A worker gets a machine setup from the system. He uses these settings to configure the machine and produces a single sample from the machine. The measurements of this sample are put back into the system. If the sample measurements are acceptable the machine is correctly calibrated. If they are outside of the tolerance values, the machine has to be recalibrated with new values returned by the API.

3 RESTdesc



RESTdesc[1] decides what steps have to be executed to reach a specific goal. These steps are based on provided API descriptions and the requested goal. To determine what the next step is RESTdesc makes use of the EYE reasoner[2][3]. This reasoner takes all of the API descriptions and all the information that was generated so far. Using that information the reasoner determines which information is still required to reach the end goal, and which APIs still need to be called to reach that end goal.

[1] http://restdesc.org/
[2] http://n3.restdesc.org/
[3] http://eulersharp.sourceforge.net/

3.1 RESTdesc for the use case

There are 2 APIs necessary for this use case: an API to request the initial machine information, and an API to validate the sample measurements. There are also 2 phases where input from the worker is required: once to input the calibration ID and once to input the sample measurements. Since the EYE reasoner only makes use of API descriptions, we model the possible worker actions as API descriptions, with the input being the necessary information required to undertake the action and the output being the information required from the worker.

This means we have 4 APIs for this use case:

- API1: Provides machine calibration information.
- API2: Validates sample measurements (and returns new machine calibration information).
- WorkerAPI1: Provides a calibration ID.
- WorkerAPI2: Provides sample measurements.

A sample full run of the use case would then be:

1. Call EYE with the initial API descriptions and the required goal. In this case the goal is an “ok” response from API2.
2. EYE returns a description of WorkerAPI1, since we need a calibration ID to start.
3. The application notices the next step is a WorkerAPI, so the worker receives an input box to enter the ID.
4. EYE gets called again with the additional information provided by the worker. It now returns a description of API1.
5. Since API1 is not a worker API, the application calls API1 without interrupting the worker.
6. EYE gets called with the new information from API1 (and all the information from the previous steps). The result is WorkerAPI2.
7. The client shows the results of the previous calls (e.g. machine parameters) to the worker and asks for the sample measurements.
8. EYE gets called with the additional information provided by the worker. The result is API2
9. API2 gets called by the application. This provides a result for the measurements, either “ok” or “recalibrate”.
10. EYE gets called with the information from API2. If the result was “ok”, the reasoner notices we found our goal and we are finished. If the result is “recalibrate” we go back to step 6 with the new machine parameters provided by API2.

As can be seen from the above example, every step consists of asking EYE what to do next and then doing that step. The result of that step then gets sent back to EYE to determine the next step. This continues until we reach our goal.

4 RESTdesc API descriptions

The EYE reasoner uses the N3 logic syntax to do its reasoning. This means the API descriptions have to be in the same syntax. We will show some examples of these descriptions as they were used in the demo.

4.1 API1 example

```
# API 1
{
    ?x a :calibration;
       :id ?n.
}
=>
{
    _:request http:methodName "GET";
              tmpl:requestURI ("http://pacific-shore-4503.herokuapp.com/calibrations/" ?n);
              :root ?x;
              http:body [ :TODO ?x ];
              http:resp [
                  http:body[
                      :operator _:o;
                      :part_number _:p;
                      :tolerances ( (_:min1 _:max1) (_:min2 _:max2));
                      :machine_parameters (_:p1 _:p2 _:p3 _:p4)
                  ]
              ].
    ?x :operator _:o;
       :part_number _:p;
       :tolerances (( :min1  :max1) ( :min2  :max2));
       :machine_parameters (_:p1 _:p2 _:p3 _:p4).
}.
```

The listing above is the complete description of API1. Each description has 2 main parts: the requirements and the result of the API call. Here the requirements are that we have a calibration object with a stored id.

If we have the required data, we can make a GET request to the specified API (with the id appended). If the request is successful, we will receive a response with the extra information as indicated in the description. This information then gets appended to our knowledge the next time we make an EYE request.

Note that we never describe in what order this API has to be executed. We simply define the input and the output of the API. The EYE reasoner then determines what the necessary order is to make sure there is a working API execution order.

4.2 WorkerAPI2 example

```
# API "worker" measure
{
```

```
    ?x a :calibration;
       :part_number ?n;
       :machine_parameters (?p1 ?p2 ?p3 ?p4).
}
=>
{
    _:request http:methodName "GET";
              tmpl:requestURI ("http://askTheWorker/" "doMeasurement");
              :root ?x;
              http:body [
                  :message "Please measure a new part with the following settings.";
                  :part_number ?n;
                  :machine_parameters (?p1 ?p2 ?p3 ?p4)
              ];
              http:resp [
                  http:body[
                      :geometrical_dimension (_:d1 _:d2)
                  ]
              ].
    ?x :geometrical_dimension (_:d1 _:d2).
}.
```

This description corresponds to the second worker API. As can be seen, this is quite similar to the description of a normal API. We use http://askTheWorker as request URL to indicate that this information should be obtained from the worker. The response then shows the information that is expected from the worker. Since the descriptions are similar, the EYE reasoner just interprets this description as a normal API description.

5 JSON

As can be seen in the examples above, EYE only works with the N3 syntax, while the APIs work with JSON syntax. For this demo we have implemented a method that easily converts between the two. Take the following N3 triples for example:

```
:c1 :id 101;
    :operator "Gianni";
    :part_number "123";
    :machine_parameters ( 1200.25 0.0024 13.7 270 ).
```

This can easily be converted to the following JSON:

```
{
    "id": 101,
    "operator "Gianni",
    "part_number": "123",
    "machine_parameters: [1200.25, 0.0024, 13.7, 270]
}
```

Our demo implementation always changes between the two formats depending on which is needed. The only thing missing from the JSON representation is the subject (:c1), which is why we include it with the :root predicate in the API descriptions seen above, so the application knows which subject to update with the new information.

6 Input/output

```
{
    "http:body": {
        "message": "Please input the starting information."
    },
    "root": "c1",
    "http:methodName": "GET",
    "http:requestURI": "http://askTheWorker/start",
    "output": "...",
    "extra": []
}
```

The listing above is the response of the application after an empty POST call to demo/next has been made. These values correspond to the values that are generated by the API descriptions an can be used to do the next API call, or in this case to request information from the worker. The output field provides some extra information of what the application did which can help in understanding the inner workings of RESTdesc.

For the next call, the following JSON should be posted to demo/next:

```
{
    "json": {"id": 1},
    "root": ":c1",
    "extra": []
}
```

The ‘root’ and ‘extra’ fields need to be exact copies of the fields provided by the last application call. This is because the application is stateless, so some state information needs to be sent to continue the API flow. The json field contains the values entered by the worker. If the application is finished, the result will be a JSON object with a single field status, containing the value “DONE”.

7 Calling EYE

We also provide a simple way to test the EYE reasoner without using our rule set. This can be accessed at demo/eye. The expected input (as POST) is a JSON with the input field containing all input rules and the goal field containing the required goal. To simulate the first call in the application, EYE can be called with the contents of demo/n3/api1.n3, demo/n3/api2.n3 and demo/n3/in.n3 as input (an array with 3 strings corresponding to the input) and the contents of

demo/n3/goal.n3 as goal. Note that the application currently always returns “DONE” if no next API could be found, so this does not always mean the run was successful.

8 Known issues

There are currently still some problems with the RESTdesc demo, which will be fixed in the future:

8.1 No recalibration

If API2 return “recalibrate” instead of “ok” it will be impossible to reach the “ok” state (without restarting). As can be seen in the output dialog, the new machine parameters actually get calculated, but there is still a problem with the generation of a new calibration object, causing the reasoner to not detect that there is a new calibration.

8.2 Formatting issues

Due to some of the requirements in our rules, not all API descriptions get accepted. It is for example always necessary to have a body for the call, which explains the `:TODO ?x` data in the API1 example. Similarly, it is also necessary to always have some requirements in the API description. Empty requirements provide wrong results. This is also why we need to already define a calibration object in demo/n3/in.n3. There is also a problem with multiply nested JSON objects, which is why we internally change the tolerances to an array of arrays, instead of an array of objects.

8.3 Data format changes

The description format used here is not completely final yet. Obviously the problems mentioned above need to be changed. Besides that some other parts of the description might change or get removed.

The current statelessness of the system might also change to reduce the amount of data that needs to be resent every time.