For now, we are assuming that the server and the clients are running on the same machine.

Provenance Web Service:

Requests must be directed to the following URI: http://localhost:8080/provsimp/rest/ProcProv

Mainly supports POST requests (to the URI above). Supports GET requests too, but it is only used for testing purposes to see if the web service is up and running (just displays some sample HTML page).

JSON: The JSON payload to the POST requests for that web service must have an 'action' key. The provenance web service handles the following actions (mapped to the 'action' key in the JSON):

-save

-load

-addnodes

-copynode

-getnode

-getpaths

-getpathnodes

-getprovelems

-getprefnodes

-loadpartpref

Sample payload for requests to the Provenance Web Service:

```
1) "save"
{
"action":"save",
"nodes":[.....], //as in CISpaces
"user": username
 }

RETURN
{"prov":"...."}
```

2) "load"

```
{
"action":"load",
"nodes":[.....], //as in CISpaces
"prov": {.....} //prov saved in the file
 }
RETURN
{"response":"success"}
```

3) "addnodes"

```
{
"action":"addnodes",
"nodes":{ // nodes can be many or one of the following three types, indexed by the nodeID and switched according to the key prov,meta, or cisp
        "nodeID1":{"prov":{...}},//prov
        "nodeID2":{"meta":{"text":"..","source":"..","dtg":"..","stream":".."}},//as infobox
        "nodeID3":{"cisp":{"nodeID":..., "source":..., "dtg":.....,"text": ....}}//same as cispaces
        }
}
RETURN {"response":"success"} or {"response":"fail","nodes":[...]}//does that are not added
```

4) "copynode"

```
  {
"action":"copynode",
"from":"fromid123",
"to":"toid123"
}

RETURN
{"response":"success"}
or
{"response":"fail"}
```

5) "getnode"

```
{
"action":"getnode",
"user":"Joe",
"nodeID":"fromid123",
"obf":true
}
```

RETURN

```
{"prov":"...."}
```

6) "getpaths" (call 1 for analysis)

```
 {"action":"getpaths",
 "nodeID":ebfd5885-8749-432e-96b1-b1178b4dffb6,
 "user":"Joe",
 "obf":false}
```

RETURN

```
{"paths":[
                        {"hint":"CollectWaterData",
                        "pathid":"530fa8a5-77aa-4693-a692-d364f696073d",
                        "title":"P3 - Generation Pattern"
                         "arg":" Given the provenance chain, information *NGO Lab reports examined the contamination*
                         \n- was associated with NGOLabAssistant\n- was generated by LabWaterTesting\n-
                         was generated by using primary sources WaterSampleData\nthe stated provenance elements infer
information
                         *NGO Lab reports examined the contamination*\n=> Therefore, information *NGO Lab reports
examined the contamination* is credible"
                        }
                        .....]}
```

7) "getpathnodes" (call 2 for analysis)

{"action":"getpathnodes","nodeID":ebfd5885-8749-432e-96b1-b1178b4dffb6,

"pathID":cc388379-68df-45c9-b095-5bdbf1c44065,

"user":"Joe"}

RETURN

{"nodes":[....], "edges":[.....], "root":"774929b7-66c6-4b67-9e12-e024146a65a9"}

ERS Web Service:

Requests must be directed to the following URI: http://localhost:8080/ers/rest/WriteRules

Mainly supports POST requests (to the URI above). There is a GET request too, but it is only used for testing purposes to see if the web service is up and running (just displays some sample HTML page).

JSON: The JSON payload to the POST requests for that web service must have an 'action' key. The ERS web service handles the following actions (mapped to the 'action' key in the JSON):

-eval

-nlg

Sample payload for requests to the ERS Web Service:

For the eval action:

```
INPUT

{
"action":"eval",

"graph":{
"uncert":"Off"/"On",
"nodes":
 [
   {"text": "Claim2", "annot": "N/A", "eval": "N/A", "commit": "No", "input":
"CLAIM", "nodeID": "Q1","type":"I", "dtg": "2014/10/07 13:45:33", "source":
"user1"},
   {"text": "Claim1", "annot": "N/A", "eval": "N/A", "commit": "No", "input":
"CLAIM", "nodeID": "Q2","type":"I", "dtg": "2014/10/07 13:45:33", "source":
"user1"},
   {"text": "Info 2", "annot": "N/A", "eval": "N/A", "commit": "No", "input":
"INFO", "nodeID": "Q3","type":"I", "dtg": "2014/10/07 13:45:33", "source":
"user1"},
```

```
   {"text": "Info 3", "annot": "N/A", "eval": "N/A", "commit": "No", "input":
"INFO", "nodeID": "Q4","type":"I", "dtg": "2014/10/07 13:45:33", "source":
"user1"},
   {"text": "Info 1", "annot": "N/A", "eval": "N/A", "commit": "No", "input":
"INFO", "nodeID": "Q5","type":"I", "dtg": "2014/10/07 13:45:33", "source":
"user1"},
   {"text": "Con", "annot": "N/A",  "nodeID": "L1","type":"CA", "dtg":
"2014/10/07 13:45:33", "source": "user1"},
   {"text": "Pro", "annot": "N/A",  "nodeID": "L2","type":"RA", "dtg":
"2014/10/07 13:45:33", "source": "user1"}
 ],
 "edges":
 [
   {"edgeID":"E1","fromID":"L1","toID":"Q1","formEdgeID":null},
   {"edgeID":"E2","fromID":"Q5","toID":"L1","formEdgeID":null},
   {"edgeID":"E3","fromID":"Q3","toID":"L1","formEdgeID":null},
   {"edgeID":"E4","fromID":"L2","toID":"Q5","formEdgeID":null},
   {"edgeID":"E5","fromID":"Q2","toID":"L2","formEdgeID":null},
   {"edgeID":"E6","fromID":"Q4","toID":"L2","formEdgeID":null}
   ],
}
}


 OUTPUT
 Service returns a JSON STRING:
      -list of Problems of badly formed links (PROBS)
      -list of evaluation for nodes  (COLORS)
      either one of the two is empty
      - if uncertainty also a list of assignment is shown

{
  "probs": "",
  "cause": "",
  "fail": false,
  "colors": {
      "8a69f465-10b7-497a-8fbc-c0b6f732af01": "V",
      "827b5f40-3aeb-4b6f-933f-68739c079cdd": "V",
      "2726bf17-055d-4616-8c3f-1f87bae9f0e9": "V"
  }
    "assign-found": true,
    "uncert": {
    "8a69f465-10b7-497a-8fbc-c0b6f732af01": "Confirmed",
    "827b5f40-3aeb-4b6f-933f-68739c079cdd": "Confirmed",
    "2726bf17-055d-4616-8c3f-1f87bae9f0e9": "Confirmed"
  }
  "chunks": {
    "chunk-1": {"nodes":…, "edges":…}
    "chunk-2": {"nodes":…, "edges":…}
  },

}




List of problems:

 ER1="Link does not have a head"
 ER2="Link does not have a valid head"
 ER3="Link does not contain tails"
```

```
 ER4="Link has a double premise"
 ER5="Link has a tail that does not exist"
 ER7="Links not valid, pro/con at the same time"
 ER6="Same links"
 ER7="A CQ can only have one tail"

 PRB0="Sorry, you must have at least one node for evaluation, please try
again."
 PRB1="Sorry, the input was corrupted, please try again."
 PRB2="Sorry, the reasoner has given no response, please try again."
 PRB3="Sorry, the result was corrupted, please try again."
 PRB4="Sorry, there are problems with the structure of the links, please fix
them and try again."
 PRB5="Sorry, there are nodes with the same id, please delete and try again."
```

For the nlg action:

```
INPUT
{
   "action":"nlg",
   "graph": {"nodes":…, "edges":…, "uncert":"Off"}
}


OUTPUT
{
  "fail": false,
  "text": "Here is nlg text..."
}
```

VC Web Service

URI for a POST request that adds a node: http://localhost:8080/VC/rest/node

URI for a POST request that adds an edge: http://localhost:8080/VC/rest/edge

URI for a DELETE request that deletes a node: http://localhost:8080/VC/rest/node/{nodeid} where {nodeid} is the ID of the node to be deleted.

URI for a DELETE request that deletes an edge: http://localhost:8080/VC/rest/node/{edgeid} where {edgeid} is the ID of the edge to be deleted.

This web service will also support GET and PUT requests at a later stage of this project.

Sample payload for requests to the VC Web Service:

1) adding a new node to the database

INPUT: {"text":"LEO","dtg":"2014/09/07 21:00:46","annot":{"id":"LEO"},"nodeID":"d0f55def-96ff-4439-97cd-ee13ee525838","source":"Joe","type":"RA"}

OUTPUT: {“response”, “success”} or {“response”, “fail”} if the JSON is malformed

2) adding a new edge to the database

INPUT: {"toID":"6e26817f-998e-41ac-8ada-67b866354041","fromID":"4e83ad0f-3533-42ac-94d3-ad4d7bab015c","formEdgeID":null,"edgeID":"9ea6e1b8-e0c7-42fa-8a7f-bed660be4d03"}

OUTPUT: {“response”, “success”} or {“response”, “fail”} if the JSON is malformed

3) deleting an existing node

INPUT: "d0f55def-96ff-4439-97cd-ee13ee525838" (a correct node ID)

OUTPUT: {“response”, “success”} or {“response”, “fail”} if the node doesn’t exist in the db

4) deleting an existing edge

INPUT: "9ea6e1b8-e0c7-42fa-8a7f-bed660be4d03" (a correct edge ID)

OUTPUT: {“response”, “success”} or {“response”, “fail”} if the edge doesn’t exist in the db