Test Cases

# Asterix and olympic games

To implement and test whether our system can handle concurrent data fetch and update requests, we have implemented Threading in our clients as well. Threading ensures that the client sends concurrent requests after fixed intervals. **Thus, our clients(1-6) itself perform the job of high level testing**.

Apart from that we have made unit tests using Unittest and Mock. We have also documented web service responses to basic rest requests.

Threading at the Server ensures that all the requests received are handled sequentially to ensure data integrity. This also prevents race condition and deadlock conditions.

## Responses

**Scenario 1:**

Cacofonix sends updated score to Obelix which sends an instant update to Dummy client:

1. Message from Cacofonix sending the update:



1. Update sent to Dummy client



**Scenario 2:**

Dummy client requests get medal tally for Rome from Obelix:

1. Request received by Obelix



1. Response sent to Dummy Client



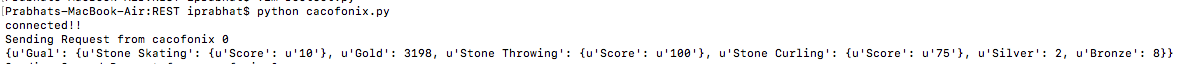
**Scenario 3:**

Cacofonix updates medal tally which reflects in database and Dummy client fetches the updated score:

1. Obelix receives update request from Cacofonix:

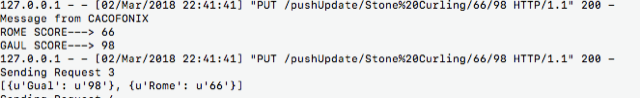


1. Dummy Client receives updated scores:



**Scenario 4:**

Dummy Client is requesting scores on a loop and reflects change in the scores when Cacofonix updates the score through Obelix. This shows data integrity in our system.



## Unit testing with unittest python and mock

Imports for our TestServices.py unit test file:

1. from unittest
2. import TestCase from mock
3. import patch from flask
4. import jsonify
5. import restGet
6. import unittest

Mocking makeJson() method which returns a JSON format of data string:

1. def mock\_getJSON(): #mock sum
2. function without the long running time.sleep json = {
3. "Team": {
4. "Rome": {
5. "Stone Skating": {
6. "Score": "20"
7. },
8. "Gold": 1542,
9. "Stone Throwing": {
10. "Score": "200"
11. },
12. "Stone Curling": {
13. "Score": "82"
14. },
15. "Silver": 8,
16. "Bronze": 7
17. },
18. "Gual": {
19. "Stone Skating": {
20. "Score": "10"
21. },
22. "Gold": 1547,
23. "Stone Throwing": {
24. "Score": "100"
25. },
26. "Stone Curling": {
27. "Score": "52"
28. },
29. "Silver": 2,
30. "Bronze": 8
31. }
32. }
33. }
34. return json

UnitTest for setScore method of restGet.py which is a PUT type of method:

1. @ patch('restGet.getJSON', side\_effect = mock\_getJSON)
2. @ patch('restGet.makeJSON',  side\_effect  =  mock\_jsonify)
3. @ patch('restGet.dumpJSON',  side\_effect  =  mock\_dumpJSON)
4. def  test\_setscore(self,  eventType,  rome\_score,  gaul\_score,  auth\_id):
5. self.assertTrue(restGet.setscore(eventType, rome\_score, gaul\_score, auth\_id))

Unittest for getScore method of restGet.py which is a GET type of method

1. @patch('restGet.getJSON', side\_effect = mock\_getJSON)
2. @ patch('restGet.makeJSON', side\_effect = mock\_jsonify)
3. def test\_getscore(self, eventType, val1):
4. self.assertTrue(restGet.getscore(eventType))

Results of the Unit testing

