Fibonacci Web Service

Table of Contents

[Architecture 1](#_Toc429592434)

[Design and Implementation 2](#_Toc429592435)

[Request and Response Format 2](#_Toc429592436)

[Fibonacci Web Server 3](#_Toc429592437)

[Fibonacci Library 3](#_Toc429592438)

[Cache 4](#_Toc429592439)

[Test 5](#_Toc429592440)

[Unit Test 5](#_Toc429592441)

[Functional Test 5](#_Toc429592442)

[Performance Test 6](#_Toc429592443)

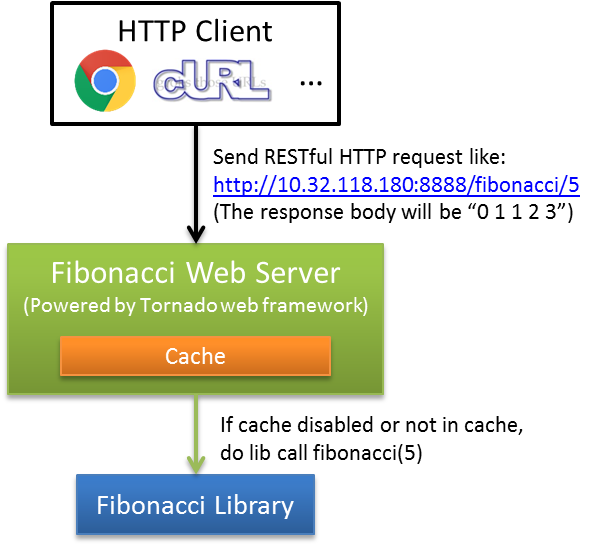
[Extendibility 6](#_Toc429592444)

[Scalability 7](#_Toc429592445)

[Further Work 7](#_Toc429592446)

# Architecture

The overall architecture and work flow of Fibonacci Web Service is illustrated by the graph below:



The component and work flow is described as follows:

* HTTP Client can be any kind of web browser or CLI, it send RESTful web service request like <http://10.32.118.180:8888/fibonacci/5> to web server.
* The Fibonacci web server is implemented under [Tornado](http://www.tornadoweb.org/en/stable/index.html) python web framework. To handle the request, it will first check if the result can be retrieved from cache in memory (if cache is enabled). If yes, construct the response from cache directly. Otherwise, call Fibonacci Library for further calculation.
* The Fibonacci Library provides the Fibonacci calculation interface, which accept an input length and return the list of the sequence.

# Design and Implementation

## Request and Response Format

The request URI is as http://<host>:<port>/fibonacci/<length>, using GET method.

The response body format is the length of Fibonacci sequence with Fibonacci number separated by one space. For example, when length is 5, the response body will be “0 1 1 2 3”, with response code as 200.

For invalid URL, or invalid length like negative number or non-number string, 404 Error will be returned.

## Fibonacci Web Server

There are many available [python web frameworks](https://wiki.python.org/moin/WebFrameworks/), among which we choose [Tornado](http://www.tornadoweb.org/en/stable/index.html) for this project. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for [long polling](http://en.wikipedia.org/wiki/Push_technology#Long_polling), [WebSockets](http://en.wikipedia.org/wiki/WebSocket), and other applications that require a long-lived connection to each user.

Based on Tornado framework, the Fibonacci web server implementation bin/**fibonacci\_web\_service.py** creates the FibonacciHandler by inheriting from RequestHandler, and implements the get() method for HTTP GET request (URL matches "/fibonacci/[0-9]+").

The usage of fibonacci\_web\_service.py is as follows:

|  |
| --- |
| ./fibonacci\_web\_service.py –h  SYNOPSIS  A RESTful web service server to handle Fibonacci sequence request with length specified  OPTIONS  Use the following options to specify a function:  -h, --help  Print this usage information  -p, --port <port>  Specify the port of this server to listen on.  Default port 8888 will be used if not specified this option.  -c, --cache <cache size>  Specify how many cache (keep first xxx number in memory) the server will use to speed up the request handling  Cache will be disabled if not specified this option. Below gives some cache size for reference:  \* Cache first 1000 number - need about 244KB additional memory  \* Cache first 2000 number - need about 840KB additional memory  \* Cache first 3000 number - need about 1756KB additional memory  EXAMPLES  # Start the server with default port 8888 and with cache disabled  ./fibonacci\_web\_service.py  # Start the server with default port 8001 and cache first 1000 number in sequence  ./fibonacci\_web\_service.py -p 8001 -c 1000 |

## Fibonacci Library

The Fibonacci sequence calculation is de-coupled from the web server implementation, by the lib/**fibonacci.py** python module.

We know recursive function is a classical implementation for Fibonacci calculation, while its efficiency may not be as good as its alternative iteration implementation, especially when recursive depth is large. And there is some max recursion limit set by default. You can check and change it as follows. And when max recursion limit is exceeded, Python RuntimeError will be raised.

|  |
| --- |
| >>> import sys  >>> sys.getrecursionlimit()  1000  >>> sys.setrecursionlimit(2000)  >>> sys.getrecursionlimit()  2000 |

There are two implementations of Fibonacci calculation interface in **fibonacci.py** module as follows. The first one is recursive implementation and the second is non-recursive implementation. Both of them accept a length input and return the corresponding Fibonacci sequence. The “base” parameter in second one will be discussed in “Cache” section below.

* fibonacci(length)
* fibonacci\_non\_recursive(length, base=None)

Based on the info above, the fibonacci\_web\_service.py calls the non-recursive implementation. And the two interfaces can also be used to verify the result of each other in Unit Test.

## Cache

Cache can help reduce the Fibonacci sequence calculation for calling Fibonacci Library for every incoming request, which can save CPU resource and shorten the response latency.

Given the trait of this Fibonacci request and required response, it does not need a traditional LRU cache mechanism to cache Fibonacci number or specified length of Fibonacci sequence. It just needs an initialized Fibonacci sequence to be cached in memory when server starts.

For example, when specifying option “--cache 1000” for fibonacci\_web\_service.py, it will initialize the Fibonacci sequence cache=[0,1,1,2,3,…] with length=1000 and cache it in memory. For any incoming request, the server will handle it as follows:

* The length is <= cache size (1000), fetch it directly from memory by cache[:length]
* The length is > cache size (1000), call fibonacci. fibonacci\_non\_recursive(length, cache) to calculate the Fibonacci numbers after 1000. Internally the lib call will do calculation based on the cache input, which does not start from scratch.

The table below shows the RSS (resident set size or real memory size) usage of the web server for different cache size setting.

|  |  |  |
| --- | --- | --- |
| **Cache Size** | **RSS (KB)** | **Additional Memory (KB)** |
| 0 | 12,484 | N/A |
| 1000 | 12,728 | +244 |
| 2000 | 13,324 | +840 |
| 3000 | 14,240 | +1,756 |

# Test

## Unit Test

Unit test is to test the interfaces in Fibonacci Library by specifying corresponding parameter and verify return value, raise exception, etc. It leverages the Python unit testing framework [unittest](https://docs.python.org/2/library/unittest.html).

For lib/**fibonacci.py**, there is a corresponding unit test file tests/**test\_fibonacci.py**, in which there are testing cases implemented for each interface in fibonacci.py. Similar unit test file can be added for other library module.

The tests/**unit\_test.py** script groups the test cases in test file like test\_fibonacci.py by test suite. Executing it will run all test cases included in the test suite. The example is as follows:

|  |
| --- |
| ./unit\_test.py  test\_fibonacci (test\_fibonacci.FibonacciTestCase) ... ok  test\_fibonacci\_no\_recursive (test\_fibonacci.FibonacciTestCase) ... ok  ----------------------------------------------------------------------  Ran 2 tests in 0.001s  OK |

## Functional Test

Functional test is to test the functionality of the Fibonacci web service by sending various requests and verify corresponding response. The tests/**functional\_test.py** script automates some normal cases as follows:

|  |
| --- |
| ./functional\_test.py  Host: localhost, Port: 8888  Test normal cases...  [SUCCESS] Fibonacci request (length=0) functional testing passed.  [SUCCESS] Fibonacci request (length=1) functional testing passed.  [SUCCESS] Fibonacci request (length=5) functional testing passed.  [SUCCESS] Fibonacci request (length=300) functional testing passed.  [SUCCESS] Fibonacci request (length=2000) functional testing passed. |

The host and port can be customized and the full usage is as follows:

|  |
| --- |
| ./functional\_test.py -h  SYNOPSIS  Functional testing for the fibonacci web service  OPTIONS  Use the following options to specify a function:  -h, --help  Print this usage information  -a, --host <host name or IP address>  Specify the hostname or IP that the web service is running  Default value "localhost" will be used if not specified this option  -p, --port <port>  Specify the port of that web service is listening on  Default port 8888 will be used if not specified this option.  EXAMPLES  # Do functional testing for localhost:8888  ./functional\_test.py  # Do functional testing for 10.32.118.180:8001  ./functional\_test.py -a 10.32.118.180 -p 8001 |

For negative cases like negative or invalid length, or invalid URL, 404 Error are expected, and they can be tested by web browser manually. They are not automated by functional\_test.py currently due to time limitation.

## Performance Test

Use [Apache HTTP server benchmarking tool (ab)](http://httpd.apache.org/docs/2.2/programs/ab.html) to do a simple performance testing by following command, which means use 5 concurrent threads, with thread sending 1000 request of http://10.32.118.189:8888/fibonacci/800. The testing result in the table shows cache helps the performance improvement very much.

* ab -n 1000 -c 5 http://10.32.118.189:8888/fibonacci/800

|  |  |  |
| --- | --- | --- |
| **Cache Size** | **Time taken for tests** | **Time per request** |
| 0 | 6.698 seconds | 33.492 [ms] |
| 1000 | 1.508 seconds | 7.540 [ms] |

# Extendibility

Although the Fibonacci web service is simple, the architecture of this project is designed and developed with high maintainability and with extendibility, which is represented as follows:

* For any new kind of request requirement, it can be simply fulfilled by implementing new RequestHandler like FibonacciHandler, and add corresponding URI mapping. (Of course the name fibonacci\_web\_service.py may needs be updated to reflect its extended scope)
* For any new kind of library, it can be added under lib/, by following the example of lib/fibonacci.py
* The unit test and functional test is also simply for extend for new library and RequestHandler.

# Scalability

If the request load is too high that exceeds a single node deployment’s capability. The Fibonacci web service can be simply and linearly scaled by deploying more server nodes, with some kind of load balancer (e.g. DNS) in front of them.

Currently the cache for FibonacciHandler is very small so each scaled node deployment can maintenance their cache copy. For future added RequestHandler which need large cache and complex cache mechanism, some 3rd party cache mechanism like memcached/redis can be leveraged.

# Further Work

To deploy the Fibonacci web service to a production server, we need a service init script to make it run as daemon, and control the service start/stop operation. And logging mechanism need also be added with log rotation enabled. This part of work is not included in this repository.