# Demonstrate REST API implementation using HTTP protocol

## Task 1

Create multithreading HTTP server that implements REST API for following methods:

* 1. **API-1: Return file size (optional)**  
     Returns file size (in bytes) for the file specified by the UNC file-path ([\\server\volume\path\file.ext](file:///\\server\volume\path\file.ext)) on Windows or local file path on Linux (/path/to/file.ext). If there is an error, server responses the error description (single line, short description of the error reason).
  2. **API-2: Return MD5 hash string based on the file content (required)**Returns MD5 hash value (as hexadecimal string) for the file content specified by the UNC file-path or local file path. If there is an error, server responses the error description (single line, short description of the error reason).
  3. **API-3: Create unique server side file copy** **(required)**  
     Creates a *server-side* local copy of the file specified by the UNC file-path or local file path, where source is the UNC file-path (specified by API call) and target file copy path is: the path prefix as defined in server XML configuration + unique file title + file extension based on source file.

The target file title is generated by the server automatically. Target file extension is the same as the source file extension. Target file title has to be different even for the same UNC source file-path across two (or more) threads. To put it another way: target file name has to be guaranteed to be globally unique across any number of threads. It has to be guaranteed that no two threads may overwrite the target files each other. When the copying finishes, the server responses the local file path, number of bytes copied and the time interval spent for copying the file (in milliseconds).

On error, the server responses the error message using the short description of error (single line message).   
Copy operation has to be implemented multithreaded that means two threads may copy the same source file into two target files in the same time. It depends on server load, but we may expect they should finish the copy operation about the same time if they have started within the same moment. To put it another way: there is no queue for copy operation. Copy operation algorithm should be as quick as possible.

* 1. **API-4: Print server runtime statistics** **(optional)**  
     Prints server runtime statistics: OS environment (Linux, Windows), server startup date and time, how long the server is up (in hours, minutes, seconds), what is the memory available to JRE (max/total/free in MB), how many requests have been served of each API-X type, what is the longest and shortest and average time for thread execution (in milliseconds).

XML configuration is loaded on server startup and it is specified by command line argument.

Server logs all output on console and inside single log file that is specified in XML configuration.

XML configuration defines the listening IP and port for the server. Only HTTP protocol is required but supporting HTTPS is a plus.

The arguments must be supporting Unicode file-paths and filenames, such as Chinese, German or Russian.

**Server startup call example**

C:\EclipseWorkspaces\RestApiServer>java sandbox/RestApiServer c:\config\myApiServer.xml

**API-1 example 1**  
HTTP call from browser: <http://localhost:8123/api-1?file=\\server\public\sample.docx>  
HTTP Response in browser: File size: 21275 Bytes

**API-1 example 2**HTTP call from browser: <http://localhost:8123/api-1?file=c:\japan-docs\ソフトウェア.pdf>   
HTTP Response in browser: File size: 33088 Bytes

**API-2 example 1**  
HTTP call from browser: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>   
HTTP response in browser: MD5: f6686e49f50820fcaa25ecddaca6cde3

**API-2 example 2**  
HTTP call from browser: <http://localhost:8123/api-2?file=/usr/home/test/linux.png>   
HTTP response in browser: Error: unable to open file: /usr/home/test/linux.png

**API-3 example 1**  
HTTP call from browser: <http://localhost:8123/api-3?file=\\server\public\sample.docx>   
HTTP Response in browser: File copied to: c:\local-storage\unique\_A1.docx, copied: 21275 bytes, time: 322 ms

**API-3 example 2**  
HTTP call from browser: <http://localhost:8123/api-3?file=\\server\public\sample.docx>   
HTTP Response in browser: File copied to: c:\local-storage\unique\_C8.docx, copied: 21275 bytes, time: 323 ms

**API-4 example**HTTP call from browser: <http://localhost:8123/api-4>  
HTTP Response in browser:

Server Statistics  
OS: Windows  
Startup: 2012-11-10 10:33:18.040  
Uptime: 01:23:58  
Memory: max: 989 MB, total: 247 MB, free: 179 MB  
Requests served: API-1: 183, API-2: 110, API-3: 168, API-4: 11  
Longest thread execution: 14242 ms  
Shortest thread execution: 2 ms  
Average thread execution: 539 ms

## Task 2

Create the multithreading client (stress-test) that sends chosen API-1, API-2 or API-3 concurrent requests for the given file path specified on command line.

The number of concurrent request is also specified on command line. Each thread logs on console its own REST API call and the output from server. When all threads are served and finished, clients print the response statistics for each thread, shortest, longest and average response (in milliseconds) for all threads.

OR

Use any Java based open source tool for stress testing the API-1, API-2 and API-3. Example: Apache jMeter™ (<http://jmeter.apache.org/>)

Following should be configured on stress-testing tool:   
- REST API URL that is called  
- number of concurrent requests

It should be visible from the statistic results of the tool how the server had performed for various API requests when using different number of concurrent requests.

Reason for stress testing: Try to find out how many concurrent requests for API-2 the server is able to handle, while keeping average response under 2 seconds when using Meßaufgabe.docx file.

**Example:** java sandbox/StressTestClient http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx 4  
Example output:   
 REST API Call: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>  
 Number of threads: 4  
 T1: sending request: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>  
 T2: sending request: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>  
 T3: sending request: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>  
 T1: server response: MD5: f6686e49f50820fcaa25ecddaca6cde3  
 T4: sending request: <http://localhost:8123/api-2?file=c:\temp\Meßaufgabe.docx>  
 T3: server response: MD5: f6686e49f50820fcaa25ecddaca6cde3  
 T2: server response: MD5: f6686e49f50820fcaa25ecddaca6cde3  
 T4: server response: MD5: f6686e49f50820fcaa25ecddaca6cde3  
 Statistics:  
 T1: 1023 ms  
 T2: 2328 ms  
 T3: 1852 ms  
 T4: 2629 ms  
 Shortest response: 1023 ms  
 Longest response: 2629 ms  
 Average: 1958 ms

Conclusion: We can send about 4 concurrent requests to keep the average server response under 2 seconds.

## Task 3

Create JUnit test that utilizes StressTestClient (or stress test tool) to send the responses to the server and evaluates some of its expected features and functions. Each API-X method has to be tested at least for one condition.

The task 3 is optional.

# General requirements

Client and server applications have to behave the same under Linux and Windows.

The server code must be robust enough to support at least 10 concurrent client requests for each API call. This should be proven by using the stress-test client and should be observed from the server log file. Such log file should be given to us by a candidate.

It is allowed to use only Oracle JRE 1.6 standard libraries.

The code should be clean and each public and protected method should be documented using JavaDoc style.

Neither server nor client should crash on wrong file paths, when files are unreadable (access denied), not found, etc.

Printing stack-trace is allowed only to server log file. In case of error, server should send to REST API client (some sort of) single-line error description based on its exception.

Also client should print some reasonable error message when error occurs and not stack-trace.

Avoid such blocks of code:

try {

// some code that does something

} catch (Exception ex) {  
 // exception ignored  
}

If possible and feasible, try to implement in the code at least one design pattern (such as Singleton). Also demonstrate using of some Object-Oriented Programming concepts, such as: interface, inheritance, polymorphism or setting private class members with public getter and setter methods.

Use IntelliJ, Eclipse or JDeveloper or similar Java IDE.

# Implementation Hints

I think only some parts of hints needs to be given. We need to know the skills of the candidate in application design. On server side we strongly rely on log file and its content, therefore it is important to make reasonable logging.   
Also it is important that he/she needs to think of some kind of synchronizing when logging into a single file from multiple threads and also for statistics he/she might consider to use java.util.concurrent.atomic classes.

Hiding the stack-trace on client side prevents black-box hacking of the server code by malicious client requests.

Documentation is very important; therefore at least each public method should be properly documented.

Good testing provides robust code. Therefore stress-test and JUnit sample is required.

**Singleton** pattern can be used for configuration class instance.

<http://en.wikipedia.org/wiki/Singleton_pattern>

**Inheritance** can be used for defining API base class and then each API-X can be a subclass of APIBase.

<http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>

**Polymorphism** can be used for calling API main method.

<http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>

**Example for server configuration (myApiServer.xml)**

<server>  
 <logFile>C:\EclipseWorkspaces\RestApiServer\server.log</logFile>  
 <targetPathPrefix>C:\Temp\unique\_</targetPathPrefix>  
 <host>localhost</host>  
 <port>8123</port>  
</server>

**Example for server starting**

C:\EclipseWorkspaces\RestApiServer>java sandbox/RestApiServer c:\config\myApiServer.xml  
2012-11-12 10:33:18.001 REST API Sample Server starting…  
2012-11-12 10:33:18.010 Environment detected: Windows  
2012-11-12 10:33:18.020 Loading configuration: c:\config\myApiServer.xml  
2012-11-12 10:33:18.030 Configuration loaded…  
2012-11-12 10:33:18.031 Printing configuration:  
2012-11-12 10:33:18.031 Log file path: C:\EclipseWorkspaces\RestApiServer\  
2012-11-12 10:33:18.031 Target Path: C:\Temp  
2012-11-12 10:33:18.031 Listening host: localhost  
2012-11-12 10:33:18.031 Listening port: 8123  
2012-11-12 10:33:18.040 Server listening startup time: 2012-11-10 10:33:18.040  
2012-11-12 10:33:18.041 Listening for client requests on: localhost:8123

**Example for logging two concurrent requests with multithreading**

2012-11-12 11:10:15.249 (Thread-1) Client request: GET /api-2?file=c:\temp\sample.docx  
2012-11-12 11:10:15.249 (Thread-2) Client request: GET /api-1?file=c:\temp\sample.docx  
2012-11-12 11:10:15.254 (Thread-2) Response: File size: 21275 Bytes  
2012-11-12 11:10:15.261 (Thread-1) Response: MD5: 6b2947b1074162c1467c20a8adc10ecc  
2012-11-12 11:10:15.263 (Thread-2) Finished, thread statistics: API-1 used count: 10, exec. time:…  
2012-11-12 11:10:15.263 (Thread-1) Finished, thread statistics: API-2 used count: 48, exec. time:…

**Example for logging for API-3 for the same source file**

2012-11-12 11:10:15.113 (Thread-1) Client request: GET /api-3?file=c:\temp\sample.doc  
2012-11-12 11:10:15.113 (Thread-2) Client request: GET /api-3?file=c:\temp\sample.doc  
2012-11-12 11:10:15.114 (Thread-2) Target file path: C:\Temp\unique\_1.txt, copying from:…  
2012-11-12 11:10:15.114 (Thread-1) Target file path: C:\Temp\unique\_2.txt, copying from:…  
2012-11-12 11:10:19.213 (Thread-2) Finished, thread statistics: API-3 used count: 10, exec. time:…  
2012-11-12 11:10:19.213 (Thread-1) Finished, thread statistics: API-3 used count: 11, exec. time:…

**Example of server statistics**

2012-11-12 11:57:16.068 (Thread-8) Client request: GET /api-4   
2012-11-12 11:57:16.068 (Thread-8) Sending text:  
Server Statistics  
OS: Windows  
Startup: 2012-11-10 10:33:18.040  
Uptime: 01:23:58  
Memory: max: 989, total: 247, free: 179 (MB)  
Requests served: API-1: 183, API-2: 110, API-3: 168, API-4: 11  
Longest thread execution: 14.242 ms  
Shortest thread execution: 2 ms  
Average thread execution: 539 ms  
2012-11-12 11:10:19.213 (Thread-8) Finished, thread statistics: API-4 used: 11 times