**SER421 Fall 2017 Lab2 NodeJS HTTP**

Assigned 9/13/17, due 9/26/17 at 11:59pm via online submission to Blackboard

**Objectives**:

1. Gain proficiency in NodeJS
2. Gain greater understanding of the HTTP protocol.

**Overview**:

*THIS LAB IS A TEAM ASSIGNMENT, ALL PARTS! You must work in groups of 3, and all 3 must participate in all parts. Failure of any individual to understand/explain all 3 parts will result in a downgraded score for ALL team members.*

For this lab you will construct a simple web proxy server in NodeJS. This is a near-direct port of Lab 1, but of course there are several notable differences between Java and Javscript, and some modifications to the requirements as we will assume real HTTP requests from a browser. This lab has multiple parts.

**Part 1: Implement a Web Server Proxy in NodeJS (40%)**

For this task, you will proxy content from a destination HTTP server, output that content back to a browser. Your requirements:

1. Your program must be in a file named lab2part1.js
2. Your program must allow the following configurations via an *options* object (as shown in your examples):

<local port> - the port your program will listen on

<server> - hostname or IP address of the target HTTP server

<port> - port number the HTTP process is running on the <server> machine

<delay> - a value in milliseconds representing an artificial delay to introduce on each request

1. Your program should accept an incoming request to retrieve a file, and if that file content is not already available in the in-memory cache, it should connect to <server>:<port> and retrieve the file, store the file in it's cache, and return the content to the client.
   1. You may safely assume the content is HTML for this part.
   2. The incoming protocol to your server should be an HTTP GET request. Unlike Lab 1, this is a "real" HTTP GET request, with the various headers.
      1. Note that use of a 3rd party library for server functionality outside of what has been discussed in class (http, url) is not allowed – you must write the server from scratch.
      2. You must NodeJS's http module.
      3. Any non-GET HTTP request must be rejected with the proper HTTP error response code sent back to the browser.
2. Handle errors returned from the HTTP server (400 and 500-level errors) by passing the error back to the client. Note this is slightly different than Lab 1; I do not expect you to wrap an error in HTML content, I expect you to pass the HTTP response in its entirety (code, headers, payload) back to the browser (passthrough).
3. For this part, you do not have to follow redirects (300-level messages); however you should report back to the browser the proper HTTP error response code. Note: again this is slightly different from lab 1.
4. Your program must support multiple concurrent requests. You need to have a correct and high-throughput design to your NodeJS functions.
   1. The <delay> command-line argument is meant to be an integer in milliseconds to pause that thread while processing the request. The intent here is to be able to artificially induce latency for testing. To do this in node, install the *sleep* package via npm (<https://www.npmjs.com/package/sleep>), put the line "var sleep = require('sleep');" at the top of your file, and have the first line of your request processor be "sleep.msleep(<delay>);".

Note that this part is just a proxy, not a caching proxy. Getting to that…

**Part 2: Extend your Web Server Proxy in NodeJS (35%)**

Your part 1 solution only supported retrieving text content and HTTP GET, and did not support redirects. Let's fix that:

1. Name this program lab2part2.js
2. Extend your solution by supporting Media Formats (MIME types) including all text types associated with XML and JSON, text/javascript and application/javascript, text/css, and all image/\* types.
3. Extend your solution to handle all HTTP response code 301, 302, 303, 307, and 308. Note that for several of these codes the behavior for the proxy will be the same. However you should still not handle redirects to an https target server, so filter these out and return the error message from Part 1, step 5.
4. Support HTTP POST requests. For this you will have to receive the POST content from a browser and forward it on to the target server.
5. Add a "<max-requests>" option to your program's options. Disallow a particular browser from making further HTTP requests after it has made greater than max-requests HTTP requests to your proxy in a 10-minute span.

**Part 3: Add Caching to your Web Proxy in NodeJS (25%)**

Name this part lab2part3.js. As before, we will add a LRU cache to your web proxy server. However, there are some significant changes:

1. You may use an outside implementation <https://www.npmjs.com/package/lru-cache>. Do not use other cache implementations, only this one (or of course you may stick with implementing your own, but this seems way easier). Add <cache-size> as a value in KB representing the max size of the in-memory cache.
2. In this implementation, there is a built-in staleness feature, but it is "lazy" in the sense that it removes a stale item only when it is accessed. Add a proactive staleness cleaner that checks the cache every <freshness> seconds, where "<freshness>" is a new value for your options.
3. Add a mini-API to your web server for cache manipulation. Specifications:
   1. POST /admin/reset – clears the cache
   2. DELETE /admin/cache?key=<key – removes the cache entry for <key> if it exists in the cache, returns a 404 if it does not.
   3. GET /admin/cache?key=<key> - returns the cache entry for <key> if it exists in the cache, returns a 404 if it does not. The response should use proper Media formats (MIME types).
   4. PUT /admin/cache?key=<key>&value=<value> - creates a new cache entry for <key> with value <value>. Note <value> is always a string and should be typed as "text/plain". If there was already a cache entry for <key> overwrite it with <value>.

**Extra Credit (10 points each)**

1. Add a feature to your web caching proxy server to rewrite URLs coming from the origin server. As it is, relative URLs should already come back to your proxy server from the browser. But any fully-specified URLs coming from the same host will not. Parse the response payload if it is HTML, and sub-in a URL to your proxy server if the fully specified target server URL is present (think of everywhere you have to sub – href, img, etc.). Name your solution lab2ec1.js
2. I will award the team that has the best performance and scalability, and has correctly completed all 3 parts of the assignment (but not including Extra Credit #1) 10 extra credit points. Performance will be measured as average response time to the browser, and Scalability will be measured by increasing the number of concurrent requests from 1 to 10 to 100 to 1000. The average of the performance times at these scalability levels will determine the winner.