Multithreaded static files http server:

Functional Overview

Multi-threaded web server for serving static files using thread-pools in Java.
Provide standard HTTP functionality for:

1. Cache-headers
2. E-tags
3. Last-modified
4. Mime-type
5. Http-range

Detailed requirements:
Web server to be able to serve list of available static files and to retrieve any specific file on the list. One should be able to start the server at any configured port as required based on the deployment environment.

Server needs to support HEAD requests and Http-range file retrieval GET requests as follows:
1] Client can submit a HEAD request for a file to identify, if the file is retrievable, file size, is client's cached content up-to-date and does the server support partial retrieval of file.
2] Server responds to HEAD request with all the file processing headers except the file content itself.
3] Response headers to be included in both HEAD and GET file retrieval request:

- *Content-Type* header / Mime-type indicating type of file request
- *Content-Length* header specifying file size in bytes. Based on the length, client can decide if file needs to be retrieved partially using Http-Range
- *E-tag* header is constructed based on file's last modification time and size. E-tag value can be used by client to compare it with previously retrieved file's E-tag. If they happen to be same indicates cached file is still good and client can decide to not retrieve again. Client can also choose to append "*If-Match*" request header with cached file's E-tag value and server would compare the 2 tags and return 304 response indicating to client that its cached version is good.
- *Last-Modified* header populated with last modification timestamp of the request file in RFC_1123_DATE_TIME format. Client can choose to use this response header from HEAD request and see if it has any cached version of same file and its modification time to decide if cached copy is still good. Client can also choose to append "*If-Modified-Since*" header with cached file modified timestamp in RFC_1123_DATE_TIME format. Server will compare the provided modified timestamp with the current file timestamp to return 304 if current file hasn't been modified since.
- *Accept-Ranges* header indicating that server accepts partial file retrieval requests specified using Http-Range request headers.
- *Cache-Control* header indicating retrieved files are good to be cached for x number of seconds (default defined to be 7 days) – this head is not included in HEAD request response.

4] Http-Range or partial file retrieval request with *Range* request header – This server implementation supports partial file retrieval request for files over 1MB size and client can append the range header formatted as below:

- "Range" header format: **bytes=<range-start>-<range-end>**

range-start: byte start offset based on 0^{th} indexing
range-end: byte index indicating end of range

Example:
The first 500 bytes of a file(byte offsets 0-499, inclusive): bytes=0-499
The second 500 bytes (byte offsets 500-999, inclusive): bytes=500-999
so on…..

Server Response for the partial requests:

- If file to be retrieved happens to be less than 1MB in size, server can choose to response with the complete file instead of partial and won't append *Content-Range* header to the response headers.
- If file to be retrieved > 1MB then partial request will be honored and server will respond with requested partial range file contents with these response headers:
  - *Content-Range* header: **bytes <range-start>-<range-end>/<file-length-bytes>**
  - *Content-Length* header: number of bytes returned as part of response
- Successful retrieval of file contents will result in response code 206 indicating partial contents
- Errors occurred due to incorrect requested range or any other reasons will result in response code 416
- If requested range happens to be greater than 5MB, it will result in response code 413

5] Simple file retrieval request: Server will respond with complete file contents, response code 200 given file size is <= 5MB otherwise results in 413 response code indicating error that request file is too big.

Server Design:

In very brief, designed this static file server based of **sun's simple implementation of HttpServer** and customizing server's handleRequest functionality for static file handling.

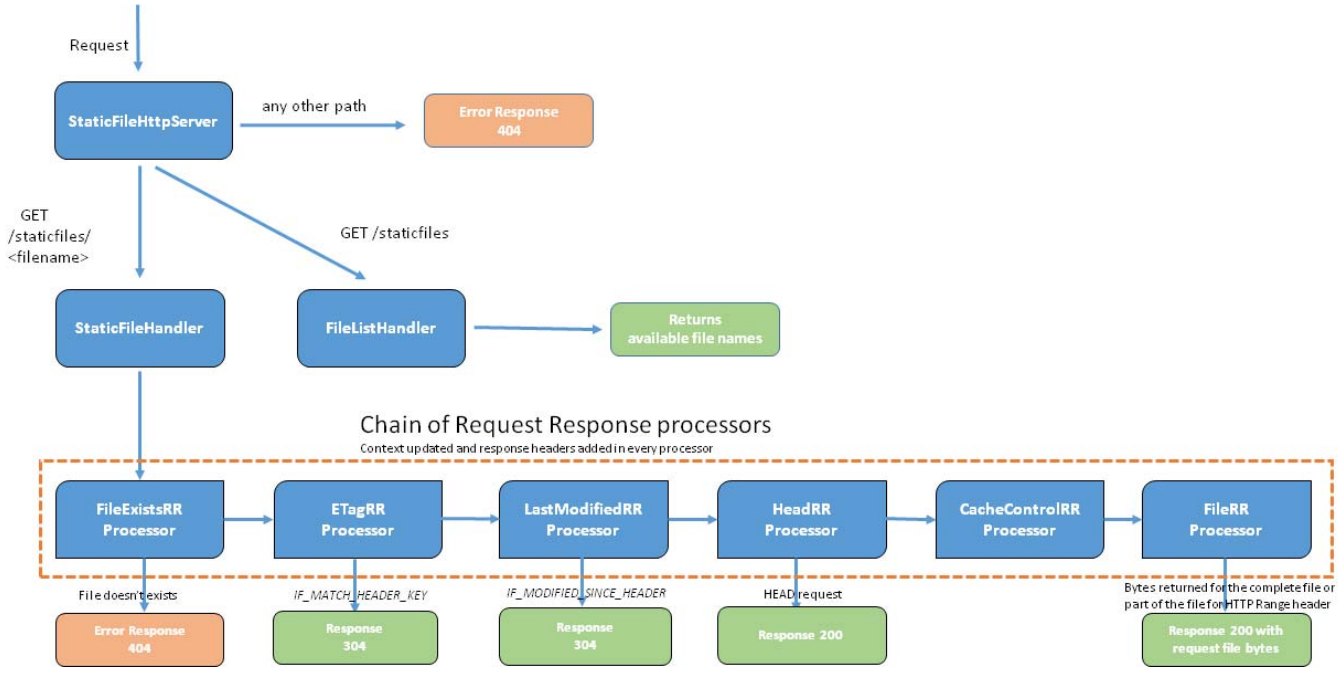
Using Java's **ExecutorService fixed threadpool** to support multithreading (One could define the pool size using an environment variable depending on where the server is deployed and cpu resources that are shared)

Used chain of responsibility pattern to process a request for each type of header and accordingly updating the processing context. Couldn't put together keep-alive header processor but it should be very easy to add another RequestResponseProcessor to the pipeline.

2 types of GET requests supported:

1] http://localhost:HOST_PORT/staticfiles or http://localhost:HOST_PORT/staticfiles/

2] http://localhost:HOST_PORT/staticfiles/<filename>



Some logic details on these **request response processors (similar to functional specs)**:

1] FileExists Processor: Checks if file exists, if it does set some basic content-type, content-length, accept-range headers otherwise pipeline is terminated with 404 response code.

2] ETag Processor: Calculates ETag header(hex(modifiedTime)-hex(size)) and also checks if there is an IF_MATCH_HEADER_KEY is provided and matches then we return 304 response code.

3] Last modified Processor: Gets the last modified time and sets it on the response header in the required format. IF_MODIFIED_SINCE_HEADER is part of request header then timestamps are compared to see if file has been modified, if not then return 304 response code.

4] HEAD request processor: Checks if it is a HEAD request then return with all the response headers we have appended so far in the pipeline with response code 200.

5] CacheControl processor: Appends the Cache control header depending on the max cache age that needs to be configured. Currently using default of 7 days.

6] File processor: This processor is the primary one that supports HTTP Range headers to read the requested file partially and return the results. If Range header is not specified then returns the complete file. Supports partial file requests for files over 1MB and returns at the most 5MB of contents in one request. If a file happens to be >5MB or request range >5MB, it will be returned with error response code 413 for file too large.

7] All the processors implement an interface, are package private and are instantiated using a factory.

How to build deploy/run/execute the server:

Project is built using jdk15 and maven3.6.3, though one should be able to use jdk version>=11 and maven version 3.x.x

Once project is cloned from here: https://github.com/Pkasliwal/staticfile-httpserver, cd into com.namo.dist.server and run the following steps:

1] Build jar (jar will be generated in target directory with this command):

mvn package

mvn package -DskipTests (to skip test execution)

2] Execute unit tests:

mvn test

3] Execute a specific test:

mvn test -Dtest=<TEST_CLASS_NAME>

4] Clean, Build, Deploy, Test, for all:

mvn clean install

5] Run/Execute the server on port 8500 (set env variable SERVER_PORT to customize port# and env LOG_LEVEL if need to be different than INFO before running the following command):

mvn exec:java

6] Run/Execute the server using jar file available in target directory (Xmx configuration can be changed as needed)

java -XshowSettings:vm -Xmx1024m -jar target\com.namo.dist.server-2.0.0.jar

7] One can build docker image as well as follows (given docker is installed in the environment)

docker build -t IMAGE_NAME:VERSION .

IMAGE_NAME and VERSION to be used for further commands:

pkasliwal/com.namo.dist.server.staticfileserver:2.0.0

8] Docker image execution

I have dockerized the server and is available on docker hub with image name:

pkasliwal/com.namo.dist.server.staticfileserver:2.0.0

Command to run the server (-m parameter is optional to configure memory allocation for the server container)

docker run -m3GB -p HOST_PORT:8500 pkasliwal/com.namo.dist.server.staticfileserver:2.0.0

docker run -m3GB -p HOST_PORT:8500 -e LOG_LEVEL="FINE" pkasliwal/com.namo.dist.server.staticfileserver:2.0.0

Above server has some static example files in-built and can be accessed using following urls:

http://localhost:{{HOST_PORT}}/staticfiles/Serverless.pdf

http://localhost:{{HOST_PORT}}/staticfiles/light_green.jpg

http://localhost:{{HOST_PORT}}/staticfiles/rhythm.mp3

http://localhost:{{HOST_PORT}}/staticfiles/simple-test.txt

9] Command to mount other files:

docker run -p HOST_PORT:8500 -v ABSOLUTE_FILES_DIR:/usr/src/myserver/files pkasliwal/com.namo.dist.server.staticfileserver:2.0.0

Unit test framework: included as part of the code base covering good percentage of codebase

Postman test Cases

Have configured some very basic postman test cases which are available at tests/postman directory. They test following happy path, example urls:

http://localhost:{{HOST_PORT}}/staticfiles/Serverless.pdf
http://localhost:{{HOST_PORT}}/staticfiles/light_green.jpg
http://localhost:{{HOST_PORT}}/staticfiles/rhythm.mp3
http://localhost:{{HOST_PORT}}/staticfiles/simple-test.txt

I have a readme populated for details. With postman, one could easily automate testing of request and response headers. Due to time crunch I couldn't add those testing scenarios and negative testcases.

Design patterns used:

1] Chain of responsibility pattern to chain request response processors to process a request.

2] Factory pattern to instantiate all request response processors from the RRProcessorFactory and keep processors package protected.

3] Singleton pattern to instantiate RRProcessorFactory instance.

Enhancements/improvements that I couldn't get to:

1] To make it a secure server using HTTPS.

2] FileRRPRocessor to be enhanced to support multipart range in one request.

3] Enhanced testing to include more complex scenarios and negative testing.

4] Minor improvements like using String constants, String buffers.