Navid Malekghaini

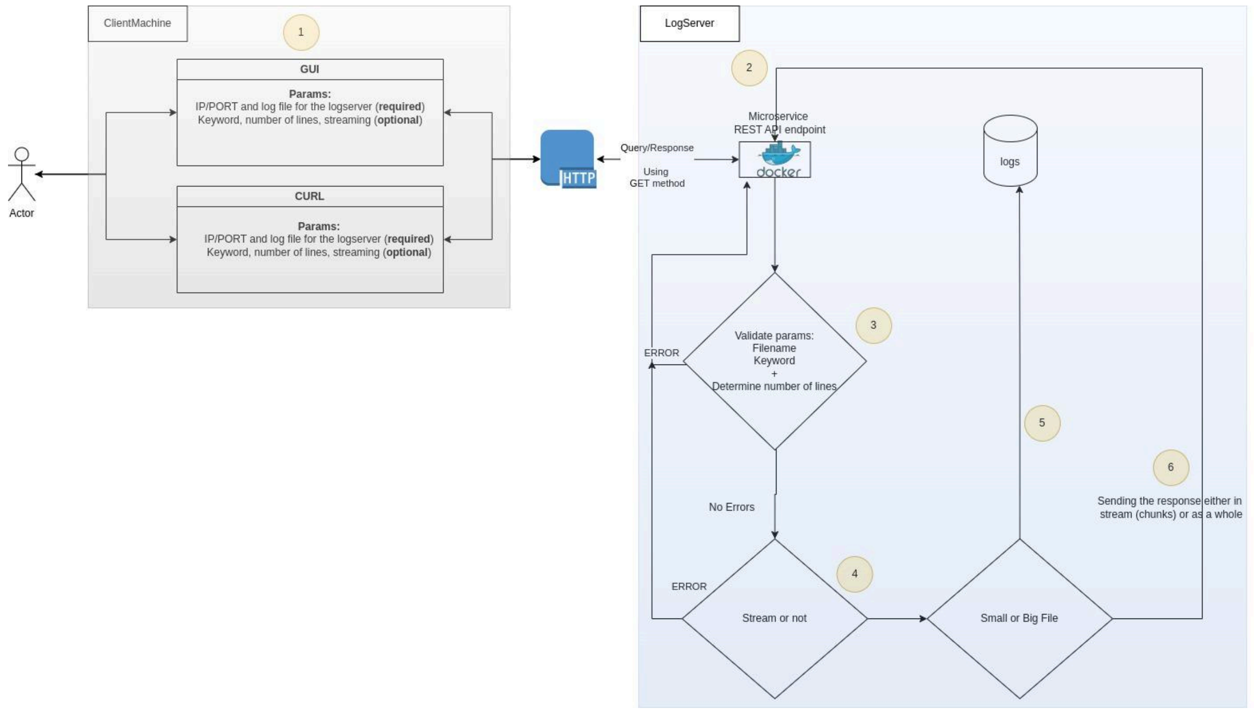
LogViewerApplication v1.1

Date: 25/01/07

Contents:

Design

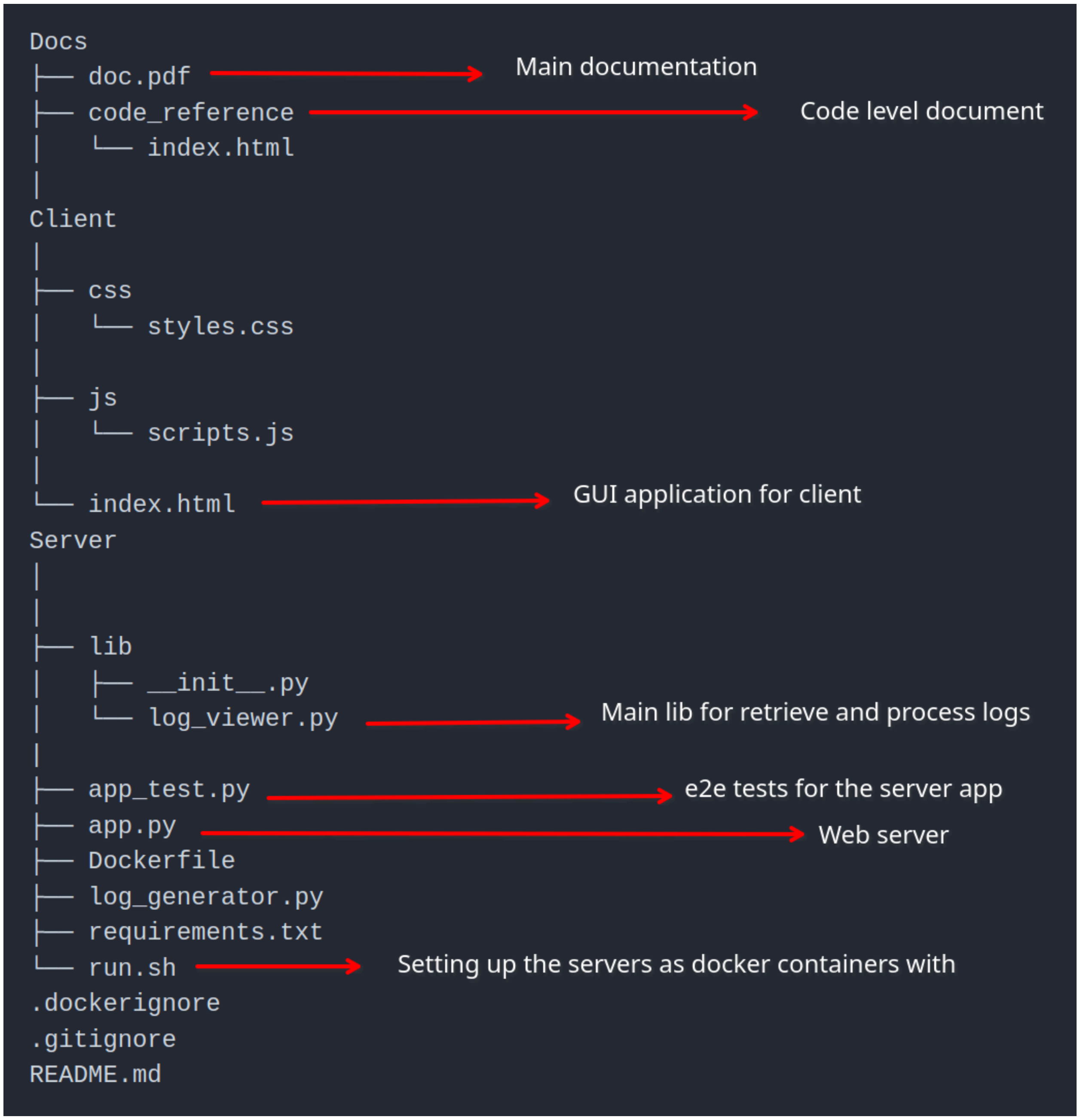
High-level design


ClientMachine
1
GUI
Params:
IP/PORT and log file for the logserver (required)
Keyword, number of lines, streaming (optional)
CURL
Params:
IP/PORT and log file for the logserver (required)
Keyword, number of lines, streaming (optional)
Actor
HTTP
Query/Response
Using GET method
LogServer
2
Microservice REST API endpoint
docker
logs
Validate params: Filename Keyword + Determine number of lines
3
ERROR
No Errors
5
6
Sending the response either in stream (chunks) or as a whole
ERROR
4
Stream or not
Small or Big File

1. User Interaction: The process begins with an actor (user) who can interact with the system through either a GUI or CURL. Both the GUI and CURL require the user to input the IP/PORT and log file for the log server (required). Optional parameters include a keyword, the number of lines, and streaming.
2. HTTP Request: The GUI or CURL sends an HTTP GET request to a microservice REST API endpoint, running in a Docker container.
3. Parameter Validation: The microservice validates the parameters, including the url path, filename, keyword, and the number of lines. If there are errors, the process terminates by sending back error to the user in a nice format (json) and human friendly (ie. nice error message).
4. Streaming Decision: If there are no errors, the microservice determines whether to stream the response or not based on the user request.
5. File Size Decision: The microservice then decides if the file is small or big. If the file is small, load the whole file into memory for processing. If not, use chunk based reading (load chunk by chunk in memory).
6. Response Handling: Finally, the response is sent back to the user either in chunks (if streaming) or as a whole.

At any stage, in case of exceptional/unforseen errors the HTTP server will catch an error (by using try/catch) and send it back to the user as an error with a message attached.

Code structure

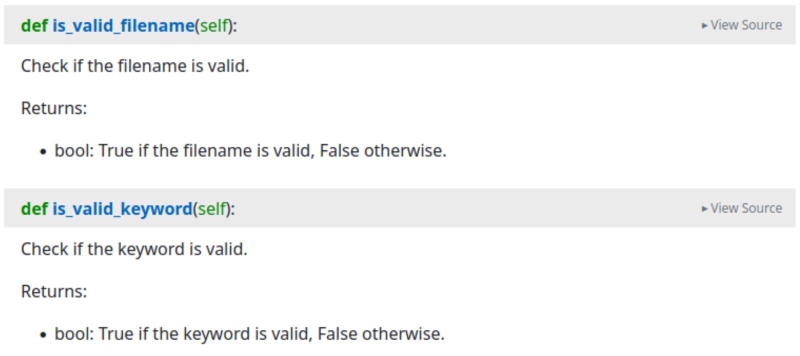


```
Docs
├── doc.pdf            →  Main documentation
├── code_reference     →  Code level document
│   └── index.html
│
Client
│
├── css
│   └── styles.css
│
├── js
│   └── scripts.js
│
└── index.html         →  GUI application for client
Server
│
│
├── lib
│   ├── __init__.py
│   └── log_viewer.py  →  Main lib for retrieve and process logs
│
├── app_test.py        →  e2e tests for the server app
├── app.py             →  Web server
├── Dockerfile
├── log_generator.py
├── requirements.txt
└── run.sh             →  Setting up the servers as docker containers with
.dockerignore
.gitignore
README.md
```

Discussion on low-level design

Backend:

- Do error checking: url is valid, usage response, valid file name, valid keyword (using regex)
  The keyword validates there are no wildcards or “/” these stuff (ie. regex). The reason behind is to make the server application robust (ie. prevent injection to some extent). Of course the current method is naive and this can be improved with a more advanced injection prevention that also supports special chars in the keyword.

  ```
  def is_valid_filename(self):
  ```

  Check if the filename is valid.

  Returns:

  - bool: True if the filename is valid, False otherwise.

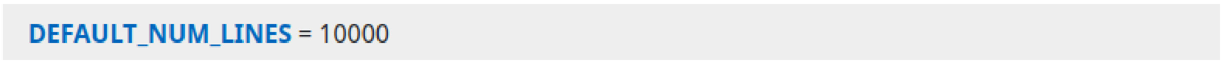
  ```
  def is_valid_keyword(self):
  ```

  Check if the keyword is valid.

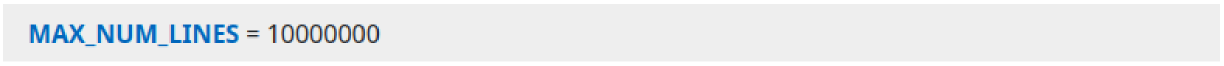
  Returns:

  - bool: True if the keyword is valid, False otherwise.

  Also, limit the number of lines returned.

  ```
  DEFAULT_NUM_LINES = 10000
  ```

  ```
  MAX_NUM_LINES = 10000000
  ```

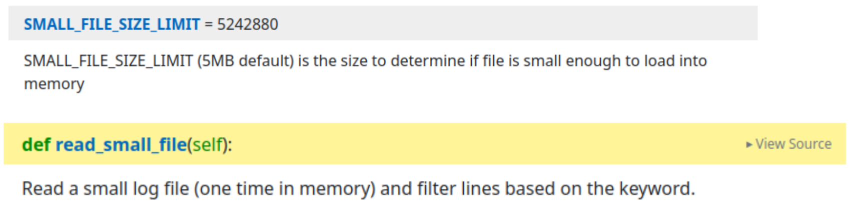
  The reason behind `MAX_NUM_LINES` is to limit the server. For example, not bombard the log server with an infinite number from the user (extremely huge number!). This will cause performance problems on the server if the resources for the process are not limited by OS. The user is responsible to break the query into smaller queries using the search method.

- The application server (ie. FLASK server) is decoupled from the LogRetriever for better feature development and modularization.

- Format of the response:
  If there is an error, use JSON (for better process at the client side). If there is no error, since there is not much processing on the client side (ie. just shows the lines retrieved) therefore using a lighter format such as binary is better. Because, json has the overhead of serialization/deserialization which can be a huge performance bottleneck.

- Small and large file check:

  Why the small file method?
  Depending on the system resources, loading a whole file in the memory would be faster to process it. This is because I/O operations are inherently slower, especially if not cached (ie. frequently accessed by OS). Therefore, if the file is small (fits into memory) it's preferred to load it into the memory once. Furthermore, it also diminishes the race condition if there are already logs being written to the file. This is because the file is already loaded into memory (ie. small file read) at that moment, and there is no worry to handle this situation.
  Then for implementation, we just read lines backwards and search for the keyword inside them up to the number of lines. Also if no keyword is there, it will just slice the lines and return them.

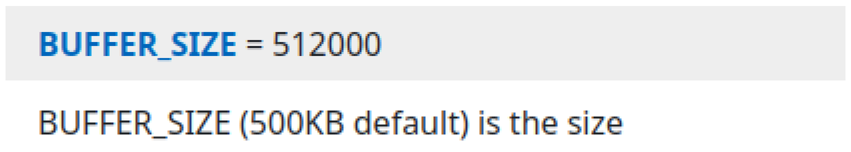

SMALL_FILE_SIZE_LIMIT = 5242880

SMALL_FILE_SIZE_LIMIT (5MB default) is the size to determine if file is small enough to load into memory

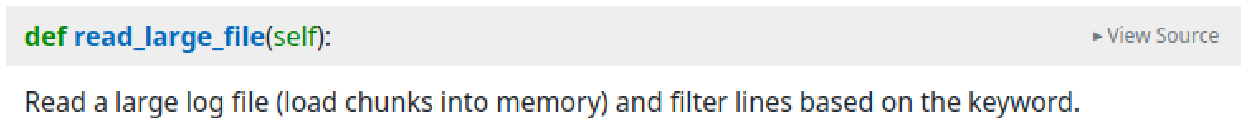
def read_small_file(self):

Read a small log file (one time in memory) and filter lines based on the keyword.

  Why the large file method?
  If the file is large, we cannot fit the file as a whole in the memory, also it takes a long time to fit it as well. Therefore, we have to read the file in chunks. Also we have to incorporate new logs that could be appended to the file. For this, we use pointers in the file: seek to the current EOF and start reading the file into memory backwardly, each time by `BUFFER_SIZE` and then decrementing the pointer by `BUFFER_SIZE` to read the next chunk.


BUFFER_SIZE = 512000

BUFFER_SIZE (500KB default) is the size


def read_large_file(self):

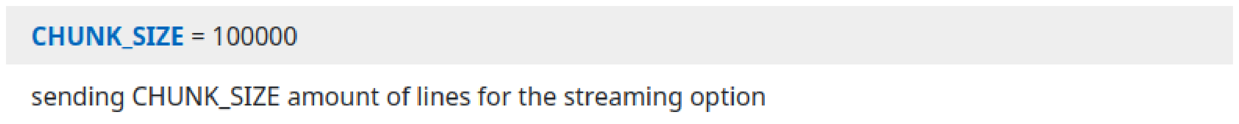
Read a large log file (load chunks into memory) and filter lines based on the keyword.

  - Why streaming option?

  In some cases there could be a streaming processing while the log files are being generated.

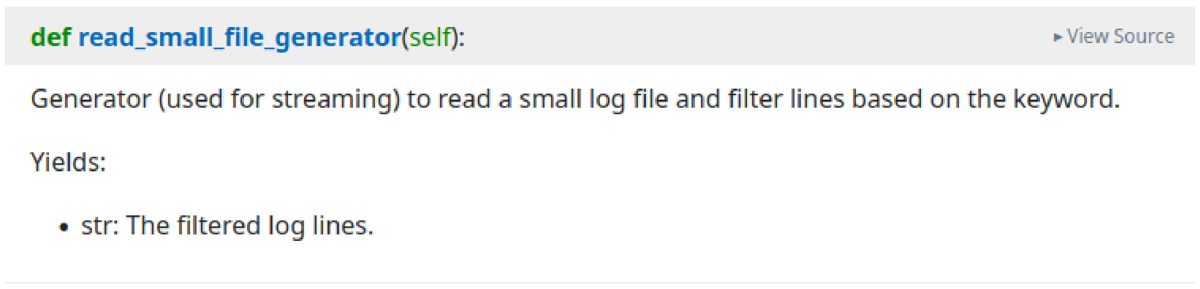
Although the REST API is not ideal for that, this can still have some benefits. For example, if we don't want to pressure the network with huge bandwidth, we can stream the response (ie. send small chunks of data as it's being read by the backend). Then the client can do stream processing as well (not reading huge amounts of data at once). Furthermore, since the logic is decoupled from the HTTP Server, then this part can be sent with more suited protocols for streaming while maintaining order such as QUIC.

For stream processing we can use `yield` option instead of `return` keyword in python. Meaning that when it reaches the `CHUNK_SIZE` it will send the result to the HTTP server and then continue processing the file where it's left. No dependency on external modules!



CHUNK_SIZE = 100000

sending CHUNK_SIZE amount of lines for the streaming option

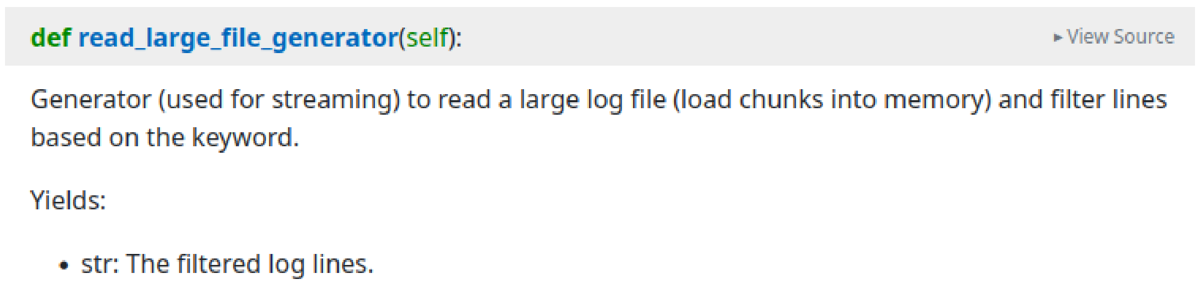


def read_small_file_generator(self): ▸ View Source

Generator (used for streaming) to read a small log file and filter lines based on the keyword.

Yields:

- str: The filtered log lines.



def read_large_file_generator(self): ▸ View Source

Generator (used for streaming) to read a large log file (load chunks into memory) and filter lines based on the keyword.

Yields:

- str: The filtered log lines.

Small file generator may not have any useful scenarios but just kept it in the code.

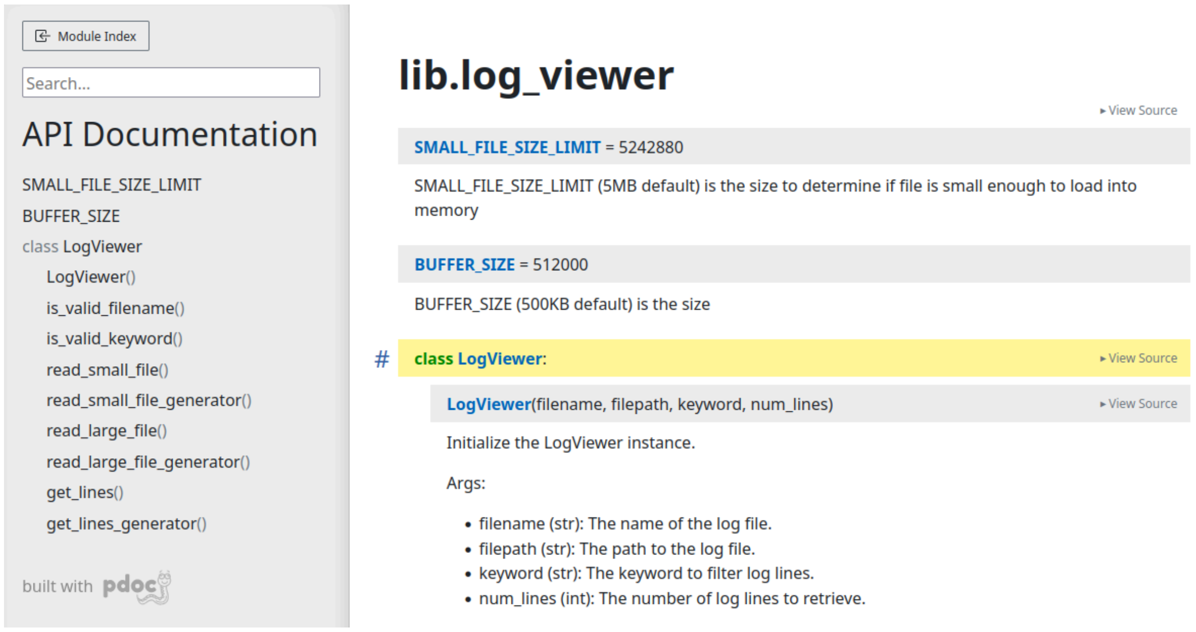
Front-end:

- There are error handlings done in the GUI to prevent sending malformed inputs to the backend and wait for the response.
- There is pagination support for the GUI, which will break the response into multiple pages for the user to browse within it. JavaScript is used for that. The pagination is also responsive to the size of the response and also the current page number.

Full-function reference

For low-level (ie. code level) documentation please open the `Docs/code_reference/index.html` in a browser after cloning the repository.

Example:



API Documentation

SMALL_FILE_SIZE_LIMIT
BUFFER_SIZE
class LogViewer
LogViewer()
is_valid_filename()
is_valid_keyword()
read_small_file()
read_small_file_generator()
read_large_file()
read_large_file_generator()
get_lines()
get_lines_generator()

lib.log_viewer

SMALL_FILE_SIZE_LIMIT = 5242880

SMALL_FILE_SIZE_LIMIT (5MB default) is the size to determine if file is small enough to load into memory

BUFFER_SIZE = 512000

BUFFER_SIZE (500KB default) is the size

class LogViewer:

LogViewer(filename, filepath, keyword, num_lines)

Initialize the LogViewer instance.

Args:

- filename (str): The name of the log file.
- filepath (str): The path to the log file.
- keyword (str): The keyword to filter log lines.
- num_lines (int): The number of log lines to retrieve.

Usage

How to use and setup

Please refer to README.md for this part.

Demo usage with video

You can watch them online.

CLI/CURL:
https://drive.google.com/file/d/1GX-HMiD0OoISi3cWBRp1MrYNMnLwFTGT/view?usp=sharing

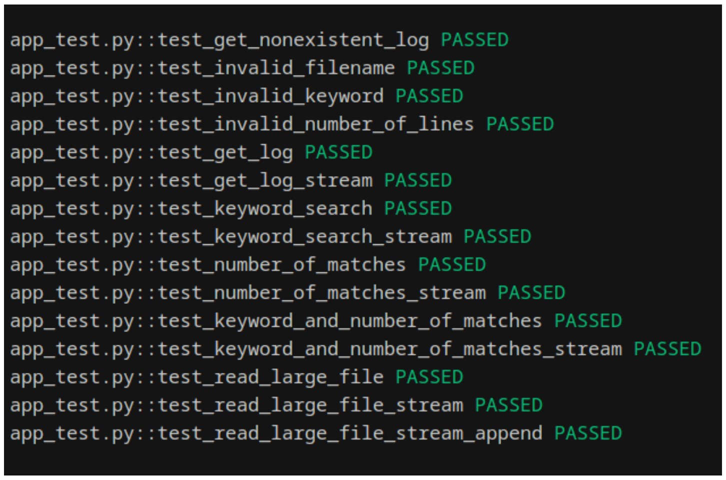
GUI:
https://drive.google.com/file/d/1V_FWIfHITb9Uc5TB201phIWwtyjiTUb-/view?usp=sharing

Test

Functional tests:

Please refer to README.md to run tests.

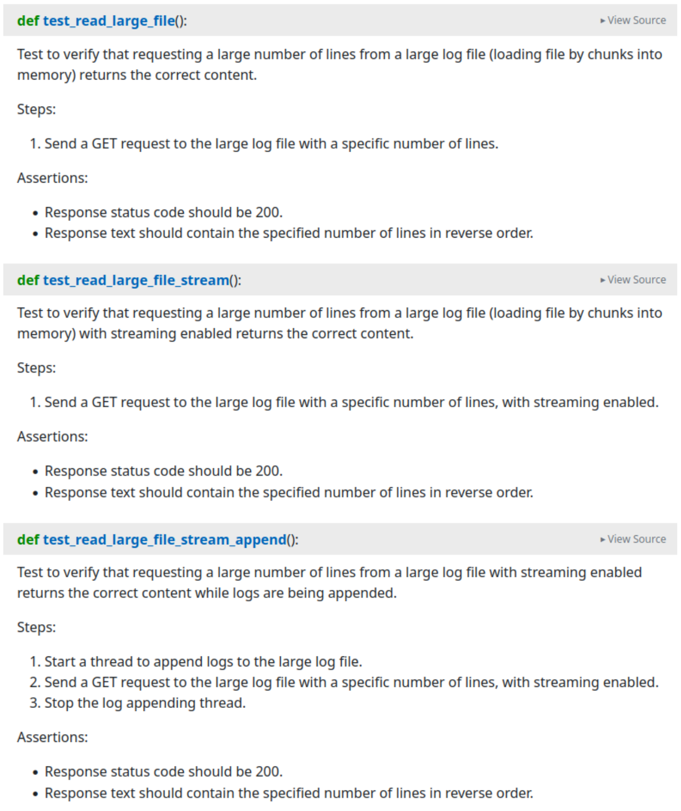
Added end-to-end tests using PyTest framework and sample log files. Why call it e2e tests? Because it uses real files and tests the microservice itself through real HTTP requests. Almost all the critical use cases are covered:



```
app_test.py::test_get_nonexistent_log PASSED
app_test.py::test_invalid_filename PASSED
app_test.py::test_invalid_keyword PASSED
app_test.py::test_invalid_number_of_lines PASSED
app_test.py::test_get_log PASSED
app_test.py::test_get_log_stream PASSED
app_test.py::test_keyword_search PASSED
app_test.py::test_keyword_search_stream PASSED
app_test.py::test_number_of_matches PASSED
app_test.py::test_number_of_matches_stream PASSED
app_test.py::test_keyword_and_number_of_matches PASSED
app_test.py::test_keyword_and_number_of_matches_stream PASSED
app_test.py::test_read_large_file PASSED
app_test.py::test_read_large_file_stream PASSED
app_test.py::test_read_large_file_stream_append PASSED
```

There is a setup and teardown function which creates actual test log files to read.

For the full reference of tests, visit:
`Docs/code_reference/index.html`
Example:



def test_read_large_file(): ▸ View Source

Test to verify that requesting a large number of lines from a large log file (loading file by chunks into memory) returns the correct content.

Steps:

1. Send a GET request to the large log file with a specific number of lines.

Assertions:

- Response status code should be 200.
- Response text should contain the specified number of lines in reverse order.

def test_read_large_file_stream(): ▸ View Source

Test to verify that requesting a large number of lines from a large log file (loading file by chunks into memory) with streaming enabled returns the correct content.

Steps:

1. Send a GET request to the large log file with a specific number of lines, with streaming enabled.

Assertions:

- Response status code should be 200.
- Response text should contain the specified number of lines in reverse order.

def test_read_large_file_stream_append(): ▸ View Source

Test to verify that requesting a large number of lines from a large log file with streaming enabled returns the correct content while logs are being appended.

Steps:

1. Start a thread to append logs to the large log file.
2. Send a GET request to the large log file with a specific number of lines, with streaming enabled.
3. Stop the log appending thread.

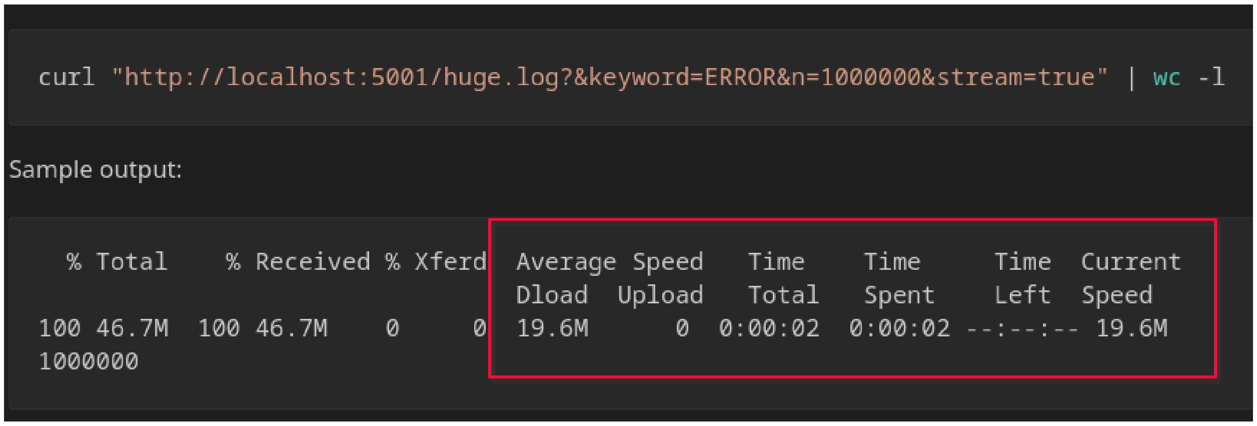
Assertions:

- Response status code should be 200.
- Response text should contain the specified number of lines in reverse order.

Non-functional tests:

For this part I haven't really had time to implement a sophisticated pressure test. I have just sticked to CURL which gives the amount time needed to get the full response. Also tested it with the user requirement of test files bigger than 1GB which is the `huge.log` that's created (refer to README.md for more info).

Example:



```
curl "http://localhost:5001/huge.log?&keyword=ERROR&n=1000000&stream=true" | wc -l
```

Sample output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 46.7M  100 46.7M    0     0  19.6M      0  0:00:02  0:00:02 --:--:-- 19.6M
1000000
```

Open-discussion for improvement

This part I just added some ideas from top of my head.

Performance

1) Implementation in a language that is faster especially with data. For example, C/C++. There are two ways:
   1 - use C/C++ for both the HTTP server and log processor API
   2 - in real world, the HTTP server could do more functionalities and it's already in another language (eg. Express/Node.js, SpringBoot/JAVA). Then the data intensive part (ie. log processor) can be re-written in C++ for major performance improvement. However, I don't recommend this approach as much as possible, because it makes the end-to-end debugging so hard.

2) Caching:
   a) Client side: it would query the backend, the backend first sends last time the file was changed, if the results are already cached on the client side, then don't bother backend again to fetch new results and just fetch results from the local.

   In other words, using “fetch through cache” from the client side, if not cached then fetch.

   b) Intelligent caching: which the GUI detects most used patterns (ie. log file, keyword, number) and then cache those in the background every couple of hours or based on the user behaviour in the background. Instagram kind of uses that for the image and videos.
   c) Backend side: the same as above but for the processing part. It could do the processing on a schedule based on the user behaviour and when the server is not so busy, then cache it.

3) Tune the configuration: for example, the chunk size, the number of limit, small file size, etc. according to the usage/load/resource availability.

4) Pagination send by the client and backend: for example, if we request last 10K lines but a page can show 200 lines, then if we’re on the first page in GUI, the backend can have this parameter as well and just send the last 200 lines and process the rest in the background and cache it. Then if the user clicks on a new page on GUI, then it grabs the rest from the backend, etc.

5) Multi-thread backend: the web server can be multithreaded (ie. thread per connection) for faster processing and handling more simultaneous connections. However, I would prefer
6) Multi-thread backend/front-end background tasks: the intelligent caching (number 2-B 2-C above) can be done as a separate thread(s).

Edge cases

1) Cover regex pattern or more sophisticated keyword search

Testing

1) The unit-tests are not covered here due to timing limit, but they are necessary for feature development in the real world. Just like end-to-end tests
2) Better non-functional testing is needed for performance tests.
3) Not all the edge cases are covered right now due to the time limit.