**F**

Daniel Cosgrove – R00153272 – SDH4

**How to run the app**

To run the app, cd to the folder and run the command docker-compose up –build

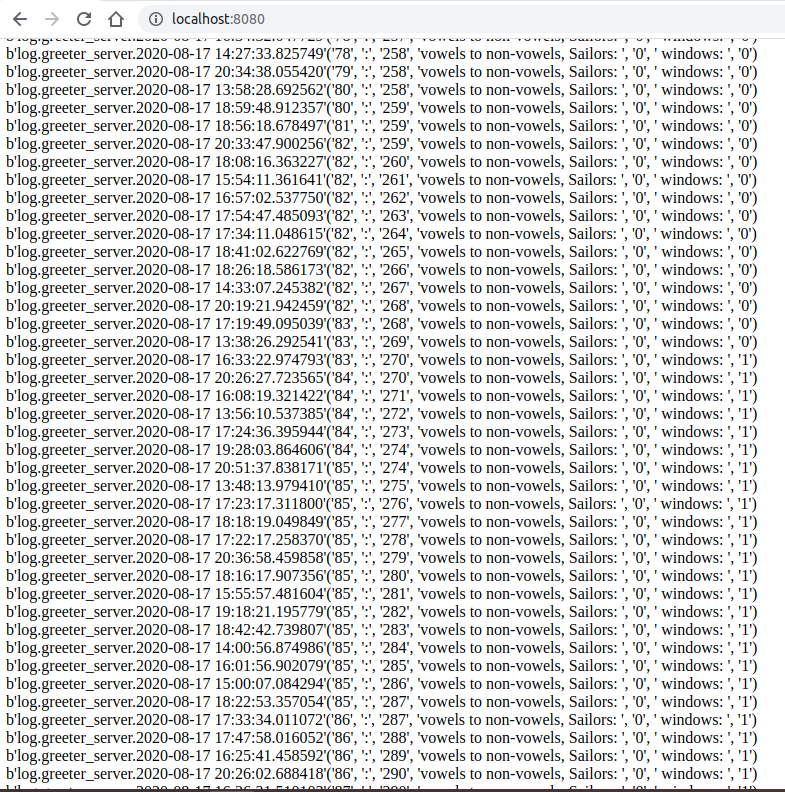
Here is the application running from the console.

A screenshot of a computer

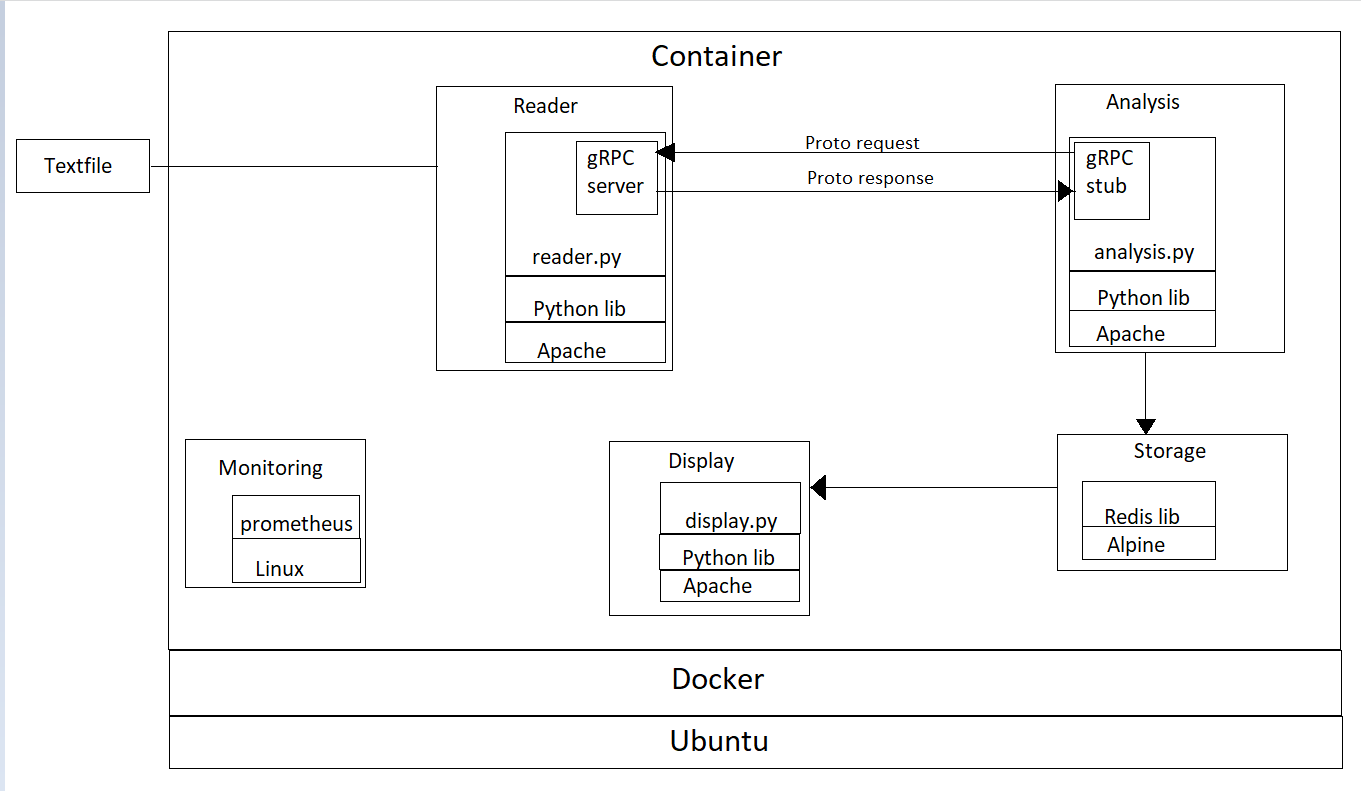
Description automatically generatedA screenshot of a computer

Description automatically generated

Here the results are being displayed on a webpage using flask.



**Architecture diagram**



**Integration and storage mechanisms**

With the micro-service model, each service is performing a specific function, therefore ‘breaking changes’ where a change to one function breaks another function are less likely to unexpectantly occur. The model is highly functionally cohesive allowing flexibility and reusability. The use of the docker container to hold the application with all it’s micro-services gives a high level of integration and portability. The docker container is hardware agnostic so can be moved easily and deployed on varied operating systems, as long as docker is installed.

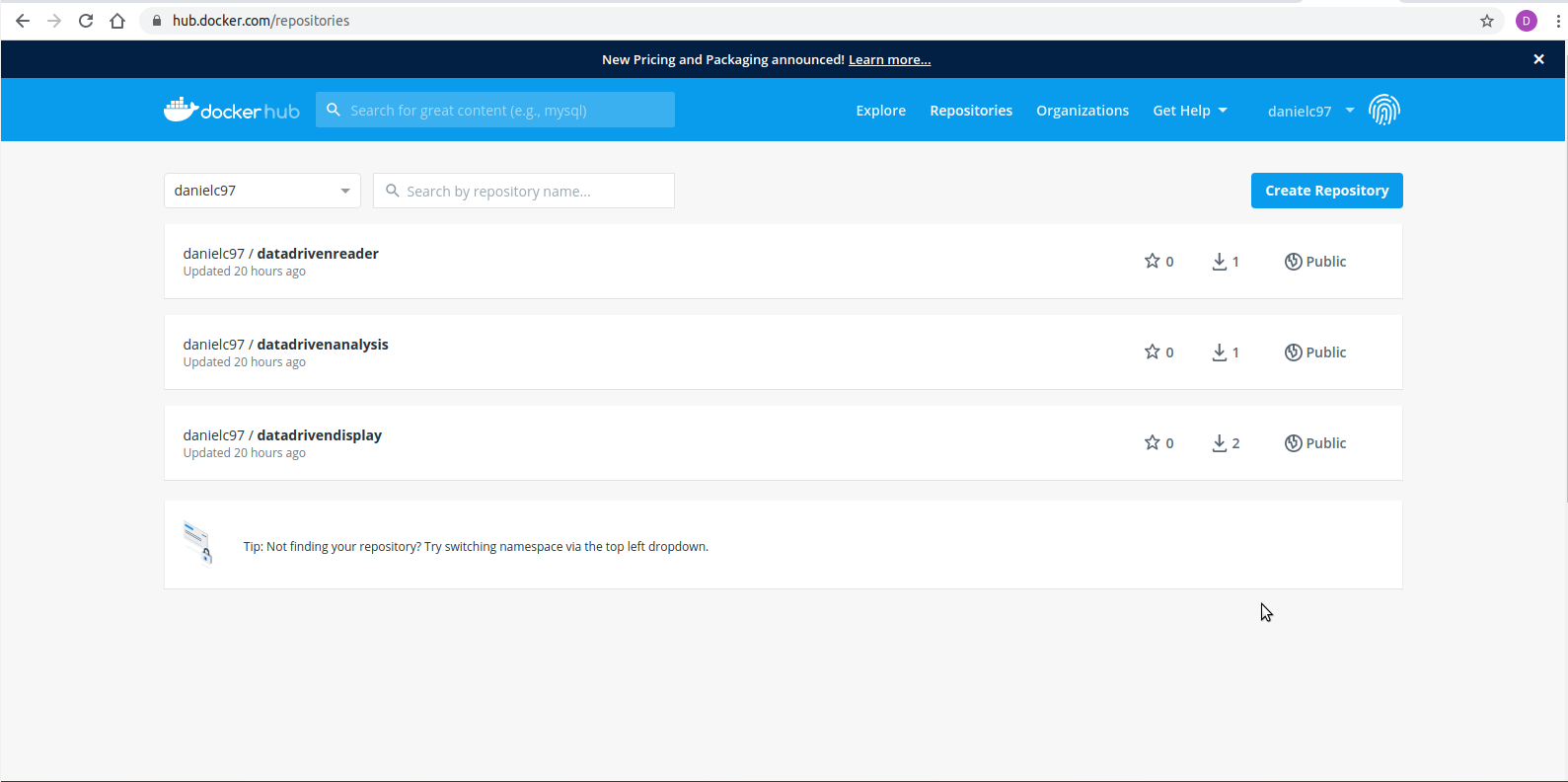
The technology chosen for inter-service communication is agnostic, for example the choice of gRPC means that any number of languages can be used, as opposed to, for example RMI where only Java can be used.

The choice of micro-service architecture allows for a high degree of agnosticism, for example the choice of using python on an apache server for the reader function could be replaced with java on ubuntu. The use of Prometheus using Go can be replaced with Graphite on nginx server.

The choice of gRPC using protocol buffers allows cross-platform calls. In this case the reader could be written in java on a ubuntu image and the analyser could be in python on apache image and gRPC would work, therefore loose-coupling.

The choice of Redis is ideal as Redis is designed around the storage of key-value pairs, is in-memory, thereby very fast and suitable for streaming applications. Although the project could be implemented using an RDS, it would result in larger images, slower performance and overheads not required, for example the capability of more complex queries.

Use of docker hub.



**Testing**

As we have separated the functions into different micro-services we can test each individually. For example test of the reader micro-service with edge cases such an empty file (null case), a huge file (load test), a file with single word, a file containing “[ } >|\^” etc.

Like-wise testing of the analyser can be specific to it, e.g. analyse a file of all words being sailor, a file of no vowels in it, only the vowel ‘e’ etc. Ideally testing of one service having no impact on another, an advantage of high cohesion. There should be a high emphasis on testing of the interfaces, in this case the protocol buffers, their methods would have to cope with all possible inputs and also focus on the gateway, in this case the interface to the text file.

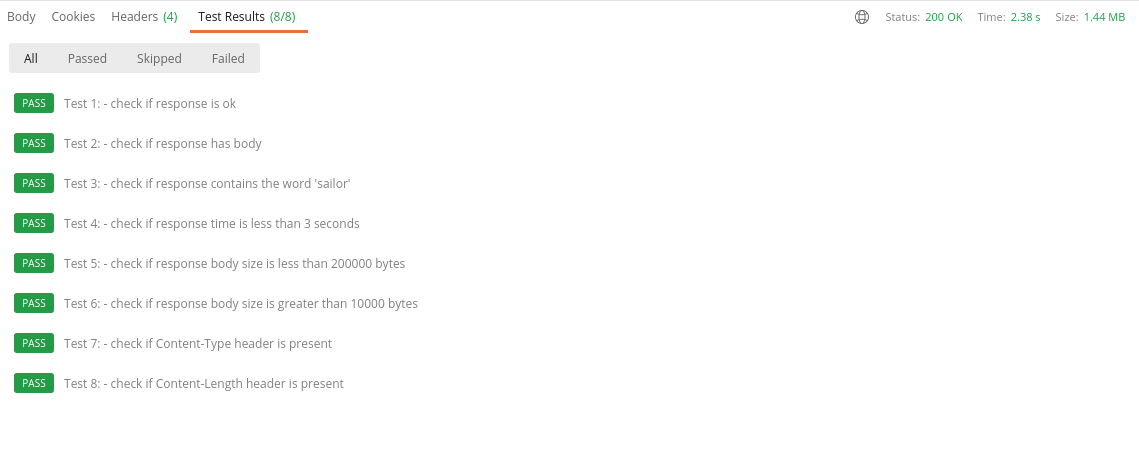
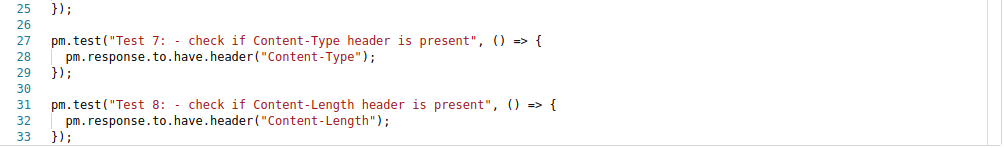
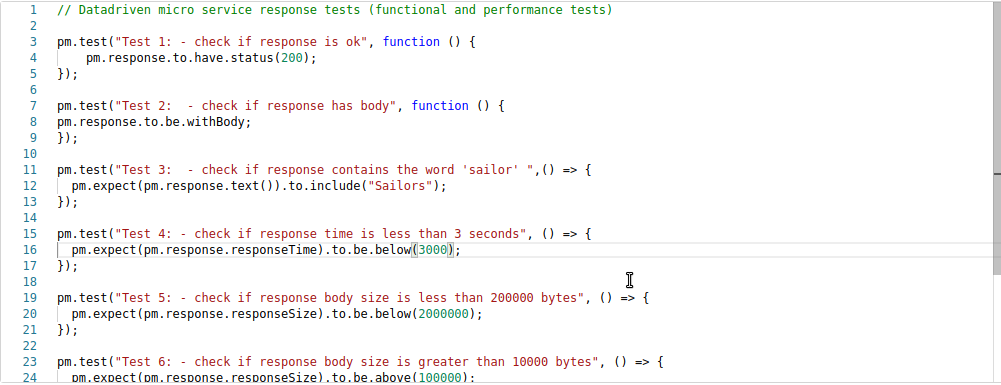
If the system is scaled-up, requiring thousands of files to be analysed, the container can be replicated, the complexity of say the reader and analyser do not change, but this will mean the gateway to the text files will be more complex. Multiple files in one location to be analysed and relayed individually to multiple containers. Therefore, gateway testing becomes more complex.

Consumer-driven contacts can be used, for example in the case of testing the analyser microservice. In this case the output from the reader microservice is not fed into the analyser, instead specific test code is used to run a battery of test sets against the analyser microservice.

A scaled-up system may include a load balancer, the failover feature of this could be tested by killing a service and testing if the system generates a replacement and how quickly.

Scaled up systems will benefit from automated continuous integration testing using an application such as CircleCI or Jenkins. Any change to the repository will be detected and trigger a set of automated tests.

**Functional, feature and performance tests run via Postman, written in java script are shown below.**

****

**Monitoring**

In the early stages it’s important to monitor the system over a period of time and store this set, this is to establish a baseline to compare later results with.

Compared to a monolith the microservice model has a higher level of granularity, each micro-service can be monitored individually. Because a micro-service may feed into another, for example the reader feeds into the analyser, when troubleshooting this should be considered. A stub or source which produces known reliable results may be used temporarily in place of a microservice, then the micro-service under suspicion is re-monitored.

In scaled systems, it’s best to aggregate the metrics and logs. This allows for sorting/querying and comparing outputs. Tools such as Kibana monitor data from hosts and logs and produce graphics and allow trends and anomalies to be highlighted and isolated. With such tools it’s possible to customise monitoring via scripts and extend with plugins to add greater analysis such as machine learning etc. Alert managers, (e.g. a feature in Prometheus) can be setup to trigger and message if a service goes down or cpu usage jumps etc. Scaled systems will need a load balancer, the load balancer itself can be monitored giving metrics on allocation of resources among services, availability of services and alarms for severe events. In a scaled system the gateways become more complex, in this case evolving from a single file pulled to one service to many files fed to many services, so the gateway will now require individual monitoring.

The outputs via the monitoring micro-service, are available in console or graph form and are show below. Prometheus monitoring scripts are written in Go and new custom scripts can be written and executed within Prometheus.

**The monitoring screen shots below show the following metrics:**

* Http returns codes over a period of two hours, 161 are 200 ok and none are 500 or 503
* Memory heap usage over one hour
* Scrape duration in seconds over a one hour period
* Http codes returned, some errors but mostly 200 ok
* CPU usage over one minute
* Total worker threads, shown as 10, this was set to 10 in code.
* Http request duration separated into buckets over 1 hours. Also shows downtime after 21:00pm. This data can be shown in histogram form.
* Scrape duration in seconds
* Up time recorded at the start of the process in milli-seconds
* Shows target captured and scrape time duration and last scrape time.

