File Integrity Monitoring

in

Cloud Environments

Alan O’Neill

National College of Ireland

[x16134427@student.ncirl.ie](mailto:x16134427@student.ncirl.ie)

[alan.oneill75@gmail.com](mailto:alan.oneill75@gmail.com)

*Abstract* - With the rapid adoption of cloud in everyday life from everything as mundane as shopping and leisure to critical functions like banking and transport, the infrastructure powering this adoption grows continually. The rise of tech giants such as Facebook, Amazon, Microsoft and Google with their immense data centres and public clouds push the number of live servers into the millions just from these players alone. Each server in turn serves a factor of virtualized environments continually spawning incomprehensible numbers of workloads that complete a function then die. A single compromised environment could lead to critical data loss and financial losses. The need for scalable security monitoring in such dynamic environments has become a critical need. File integrity monitoring is common in traditional computing and effective at reducing the attack vectors for threats such as Trojans and Ransomware, however integrity monitoring is resource intensive and complex to manage in large dynamic fast changing cloud based environments.

*Index Terms* - Cloud environments, Cyber threats, HMAC, File integrity monitoring, Python, Ransomware, Trojans

1. Introduction
2. background

This paper will describe the background, design and implementation of a scalable File Integrity Monitoring solution for use in dynamic cloud based environments.

In May 2017, cyber-attacks became mainstream news when the WannaCry ransomware attack infected in excess of 230,000 computers in over 150 countries [1]. WannaCry mainly targeted individuals and organizations running old operating systems by encrypting data. A monetary sum was demanded in order to provide the encryption key to unlock the file system. The threat of such an infection to large scale corporations is a real concern.

In my job as an integration architect in Watson Health, preventing exposure or loss of PHI (Patient Health Information) is of the upmost importance. We work with a constantly growing cloud footprint and ensuring that our environments are secure is constant.

In a scenario where a bad actor was able to gain unauthorised access and replace an everyday utility such as ‘ls’ or ‘ps’ with Malware, a then unsuspecting sysadmin could inadvertently compromise the system by running the infected utility. A need was identified for a secure light weight FIM (File Identity Monitoring) solution that could be certified and deployed in the Watson Health cloud to detect ransom-ware attacks or Malware infection.

1. aims

The main aim is to implement a production grade scalable FIM utility to address a gap within IBM Watson Health cloud environments. The FIM solution could be installed on cloud environments to establish a baseline of critical system files. The solution would ensure the integrity of files using hashes/signatures, private secret encryption keys and generated HMAC’s (hashed message authentication codes). The integrity monitor would continually scan and verify the integrity of the monitored files. If a compromised file is detected, the system would generate an alert which could then trigger some preventative action.

Additional aims are as follows:

* Performant i.e. build a solution that will consume resources within acceptable tolerances for public cloud environments.
* Secure i.e. passes internal IBM security review board.
* Deepen my personal understanding of hashing, encryption and key technologies.
* Increase my level of programming with Python.
* Expand my knowledge of new cloud technologies such as Docker, Kubernetes, NoSQL and Bluemix.

1. technology overview

The following technologies were used to develop the FIMpy FIM solution:



Figure 1: Technologies used

* Python 2.7, HTML, JavaScript, CSS
* Python Flask Web Framework – a popular and easy to use web framework. I am using this to drive a micro-service approach to expose REST end-points for clients to consume FIMpy functions via API calls and also provide a mechanism for FIMpy instances on the same network to all communicate with each other.
* Numerous Python libraries such as Slacker, pyLDAP, pyOpenSSL and APScheduler – Python has a vast community driven library of modules for every possible use-case. I had tried to reuse as much as possible.
* CloudFoundry and IBM Cloud (aka Bluemix) – IBM’s public cloud offering. FIMpy will run on Bluemix’ Python Flask runtime.
* Ubuntu – FIMpy Docker images run on Ubuntu 17.04 base image but can run on any platform that supports Python runtimes.
* IBM Containers (Docker & Kubernetes) – for the purpose of demonstrating the application and hosting, I am running a single pod Kubernetes cluster hosted on IBM’s Container solution. That allows me to run multiple instances of the application in the cloud and demonstrate how each instance can protect the integrity of its corresponding host and also communicate with any other instance running on the same network.
* DockerHub registry - for storing my FIMpy Docker images. Kubernetes searches DockerHub by default for required images.
* IBM Cloudant (CouchDB derivative Cloud NoSQL Db-as-a-Service) – a NoSQL database which is used to securely store baseline (hash + HMAC) and configuration information.
* IBM Key-Protect (Secure Key Management Service) – HMAC uses a crypto key to generate a digest which FIMpy will use to verify file integrity. KeyProtect is a secure key store solution in the cloud. This way the key itself does not need to be shared, only a reference to the stored key.
* LDAP authentication (JumpCloud LDAP-as-a-Service) – a freemium cloud based LDAP authentication service. Clients must authenticate with the app or when calling the API.
* GitHub Pipeline – source-code management and issue management,
* GitBook & Markdown for Documentation – all documentation is written in markdown. GitBook is a NodeJS library for building beautiful user friendly documentation based on Git Pages. It has lots of plug-ins such as keyword search, navigation etc.
* JetBrains PyCharm IDE – a popular community IDE for Python development.

1. scope

File Integrity Monitoring (FIM) is a complex software solution with a very wide scope especially where cloud based monitoring is a requirement. Numerous base features are required in order to deliver even a basic working prototype. Through planning, the following are all deemed lower priority and thus considered out of scope for MVP:

* + File attribute monitoring such as owner, group, file access timestamps etc.
  + Role based user access and accounting – currently, only one of the three AAA protocol is implement. Authenticated users are currently authorised to access all data and functionality. Post MVP, directory groups will be implemented.
  + Robust error handling and exception handling – dependant services are based on non-paid SLA’s and are not considered “five nines”. If services are inaccessible or credentials expire, the application may become unstable.
  + Activity logging to syslog and propagation of logs to a NewRelic or ELK stack (where necessary std-out logging has been implemented to aid development and troubleshooting).
  + Performance testing - due to the limitation of free cloud services, large volume stress testing is not feasible and thus out of scope. For example, Cloudant Free Tier has transaction per second SLA’s.
  + Database objects for configuration data (for MVP these are either hard-coded or read from local config files).
    - Penetration testing.
    - Locally encrypted key stores and CA signed certificates. SSL encryption is enabled using locally deployed RSA key and associated x509 self-signed certificate.
    - Encryption of credentials used in service to service calls (currently these are stored in plain-text in configuration files). These files are deployed with the app but are not accessible without direct server root access and thus the assumption is they are secure.

1. System
2. requirements
3. **Functional Requirements**

All key functional requirements’ use cases are shown in the use case diagram in *Figure 2*.

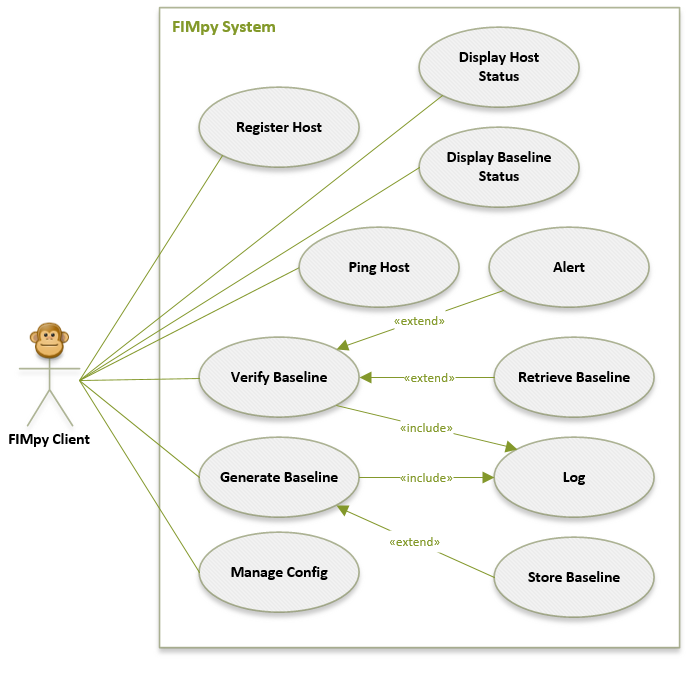


Figure 2: FIM Use case diagram

* 1. **Register Host** **–** the application will be able to register the host it’s running on when it’s starts. If the host is a new host (not already registered), the system will create a new baseline database for that host, create a registration and create the default configuration.
  2. **Generate Baseline –** the application must be able to generate a FIM “baseline” for the documents (files) to be monitored. The application will scan all documents, calculate its hash function (SHA-256), generate a hashed message authentication code (HMAC) using a secret.
  3. **Store Baseline –** the application will be able to store generated baseline data (see use case ii. *Generate Baseline* for reference) in the database.
  4. **Retrieve Baseline** **–** the application will be able to retrieve generated baseline data (see use case ii. *Generate Baseline* for reference) from the database.
  5. **Verify Baseline –** the application will be able to verify the baseline of a registered host by scanning the monitored documents (files) and comparing the hash and HMAC against the baseline stored in the database.
  6. **Ping Host** **–** the application must be able to ‘ping’ other registered hosts to determine if they are ‘alive’.
  7. **Display Host Status** – the application must be able to report the status of a registered host e.g. if the FIMpy app on that host is responding to pings as per use case ii. *Ping Host*. If the host is not communicating then it may be compromised e.g. an attacker may have stopped the FIMpy app instance on that host. Or it could just be powered off. If the host status is unknown then by extension the integrity of the monitored files cannot be guaranteed.
  8. **Display Baseline Status** – the application must be able to report the status of the baseline for all registered hosts. The status of a baseline is determine by use case v. *Verify Baseline*.
  9. **Alert** – the application will be able to alert if the integrity of a baseline for any registered host fails or if the status of a registered host cannot be determined.
  10. **Log Activity** – the application must be able to log all activity for auditing.
  11. **Manage Configuration** – the application must allow authorised clients to manage configuration e.g. add or remove monitored file paths, change settings e.g. disable push notification alerts etc.

1. **Data Requirements**

Relational SQL databases are not suited to scale and agility and therefore this project will use NoSQL databases. File integrity monitoring in general is not time or performant critical but suitable benchmark testing will be required to ensure that NoSQL databases do not throttle the overall performance. Performance considerations and testing are outside the scope of MVP.

The following data requirements are applicable:

* 1. All persistent data will be stored in NoSQL databases using IBM Cloudant.
  2. Each registered host will have its own database which will contain only the pertinent docs for that host (see *Figure 3*).
  3. All data updates will be real-time.
  4. The logical database structure will be modelled on the class diagram shown in Figure 3 below.
  5. JSON objects will represent the classes.
  6. *Document* database objects will have the following schema:

{

"\_id": "/app/test/10kfile",

"status": 2,

"modifydate": 1512961812,

"ipaddress": "172.30.91.226",

"createdate": 1512963287.028891,

"hmac": "dc54a021fdb7e23b9bdf71c06383f",

"host": "fimpy-app-1754247256-bsb74",

"hash": "84ff92691f909a05b224e1c",

"type": "doc",

"size": 10240

}

* 1. The *Host* database object will have the following schema.

{

"\_id": "host",

"status": 1,

"lastscandate": "11-Dec-2017 20:09:54",

"type": "config",

"host": "fimpy-app-1754247256-ftrr7",

"registerdate": 1513022874.068461,

"ipaddress": "172.30.91.234"

}

* 1. The remaining classes will not be implemented as persistent database objects for MVP and will instead be configured via config files at deployment time.
  2. The application will make use of NoSQL concepts such as key-value pairing, dynamic schemas and elastic scaling.

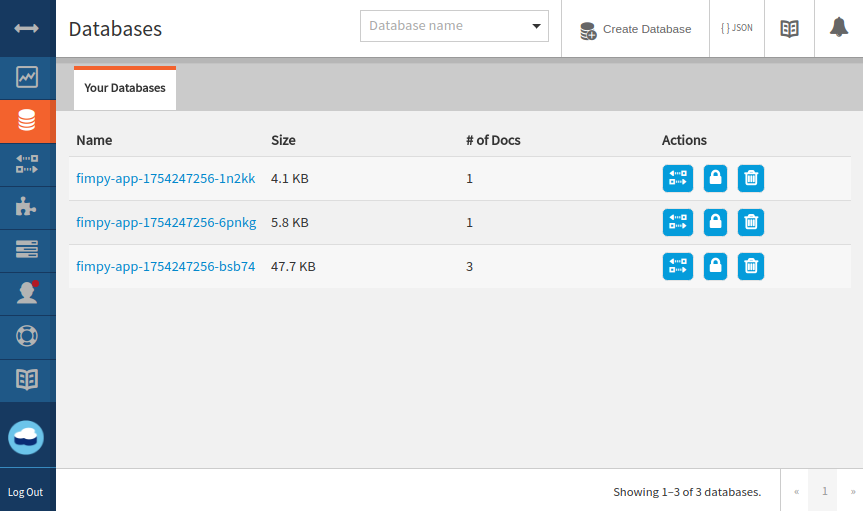


Figure 3: Cloudant Databases for Multiple Monitored Hosts

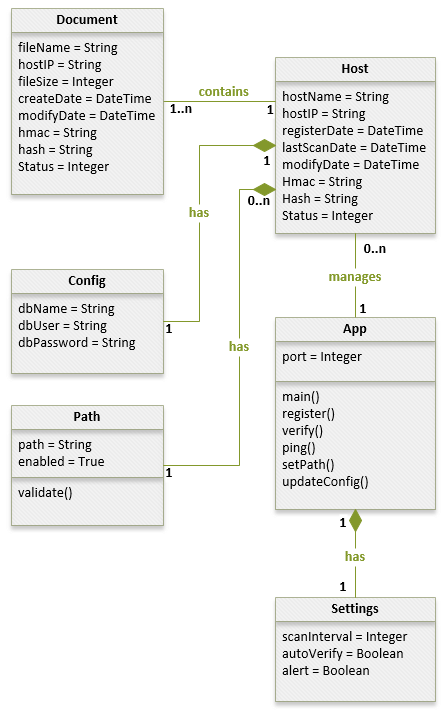


Figure 4: Class diagram

1. **User Requirements**
   1. The user requirements have been determined through research and analysis of features and function provided by other similar FIM solutions.
   2. The client/server design is unique to other FIM solutions. The concept was adapted from the authors experience with Guardium Data Encryption [2] solution. A single server node with user reporting (e.g. web console) capabilities with headless clients (agents) reporting into the central server with status. The FIMpy solution is similar except that all “node” or instances act as both server and client. This design feature may be changed in the future to represent the dedicated single server/multi-client architecture.
2. **Usability**
   1. Given the nature of FIM concepts, the application is designed to be used with minimal user interaction.
   2. Clients will interact with the application using a RESTful interface.
   3. Reporting e.g. logs, dashboards and alerts are implemented with minimal consideration to look and feel.
3. **Security**
   1. Clients access (via API call or web console dashboard) requires authentication using basic authentication. A more secure and modern authentication protocol such OAuth2 will be implemented as a future enhancement.
   2. Role-based authorisation to limit certain levels of functionality or access to different user groups.
   3. All communication will use TLS 1.2 AES encryption.
   4. Self-signed certificates will be used. CA signed certs are out of scope and therefore cert and hostname validation will be disabled for MVP.
4. Design & architecture

The system architecture was designed with cloud-native in mind. Traditional on-prem infrastructure such as application servers and SQL databases were deliberately avoided. The overall architecture design has continued to evolve as new technologies were discovered, evaluated and integrated. A conscious decision was made to use cloud-native services for key functions wherever possible and feature designs have iterated during development. The base application runtime consists of one or more instances of a Python Flask web application that runs headless. The application itself is built to a Docker image and containerised. Containers subsequently run as a pod in a Kubernetes cluster deployed on IBM Container Service.

External service were evaluated for necessary key functions i.e. Identity management and LDAP authentication, secure secret storage and NoSQL database storage. The advantages of cloud based services over locally hosted technologies are many. Most are freely available (with reduced functions or SLA’s) and require no maintenance. Inversely, third party services are subject to potential failures, maintenance or outages which add risk to the applications reliability (and also to the final evaluation and demo).

The following SaaS services have all been integrated with FIMpy over HTTP:

* **Cloudant** – Cloud based NoSQL database storage
* **JumpCloud** – LDAP-as-a-Service
* **KeyProtect** – secure key management service
* **Slack** – real-time monitoring with push notifications

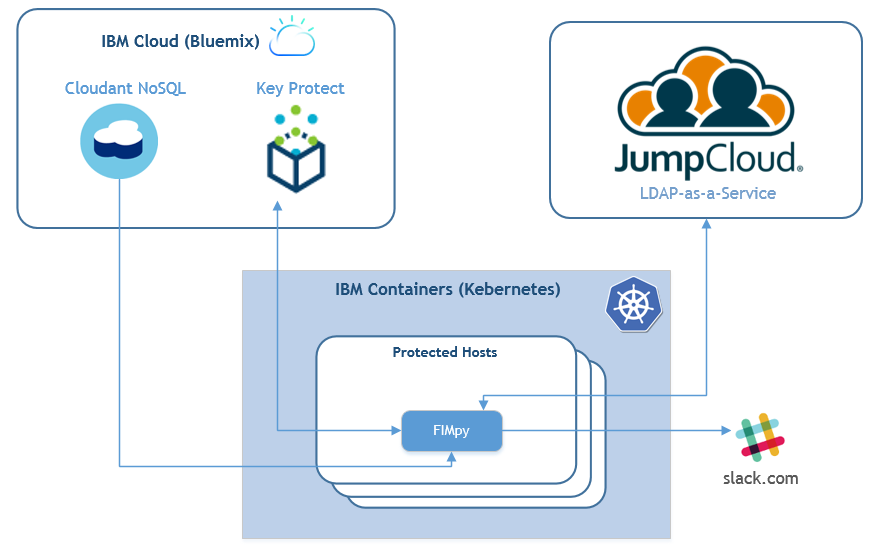


Figure 5: High Level System Architecture View

A key feature of FIM solutions is real-time alerting. Traditionally this has been via email with newer alerting platforms becoming available in more recent times. Two main target alerting solutions will be introduced into the architecture.

* **PagerDuty** – L1 support platform
* **Slack** – popular collaboration platform

For auditing, accounting and logging, the New Relic platform will be used in favour of a traditional ELK stack which typically requires heavy investment in infrastructure and maintenance.

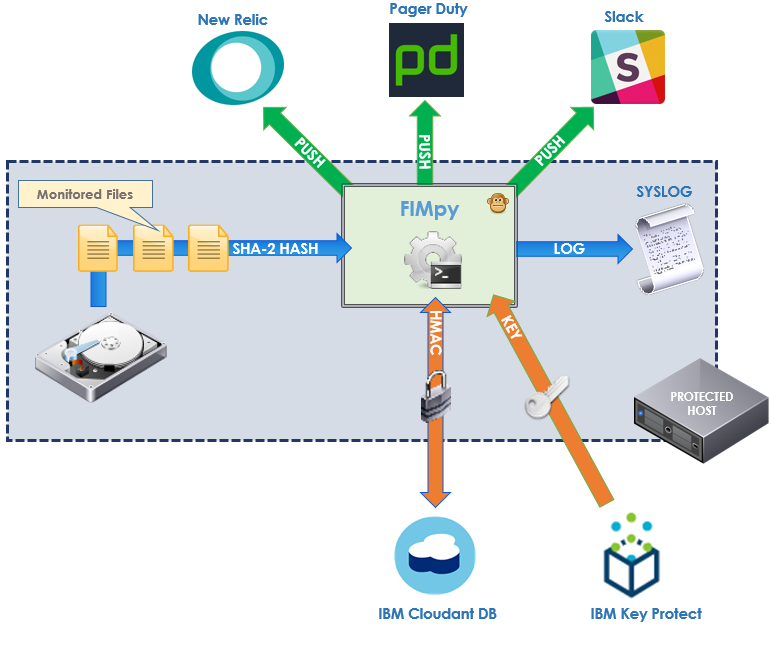


Figure 6: Detailed System Architecture View

The architecture also incorporates some of the tenants of a micro-service architecture. Functionality is developed as small consumable services that can be orchestrated to perform key functions. External clients can consume these services over HTTP. With Python Flask, even application instances themselves can consume their own services.

For example:

r1 = self.app.get('/api/v1/baseline')

r2 = self.app.post('/api/v1/verify')

Micro-services also allow application instances to communicate with each other which can be useful in a FIM scenario. If one instance goes down, the other instances can still monitor, detect and alert. Bespoke tools exist for this also such as Monit and Uptime but building this directly into the application removes unnecessary dependencies.

Of course the downside for this is that application instances can be compromised if the authentication credentials are exposed. The blast radius could be reduced by using unique credentials for each instance but this becomes difficult to manage. Chances are, if an attacker gains root access, the benefits of a FIM solution are negated anyway so this is probably a moot point in the design thinking.

The basis for the micro-services design and orchestration was determined via analysis of the user requirements and the development of a number of process flows. The main process flow for the application is shown in *Figure 7*.

In tune with an Agile development methodology, many aspects of the design have been continually iterated upon over the course of development.

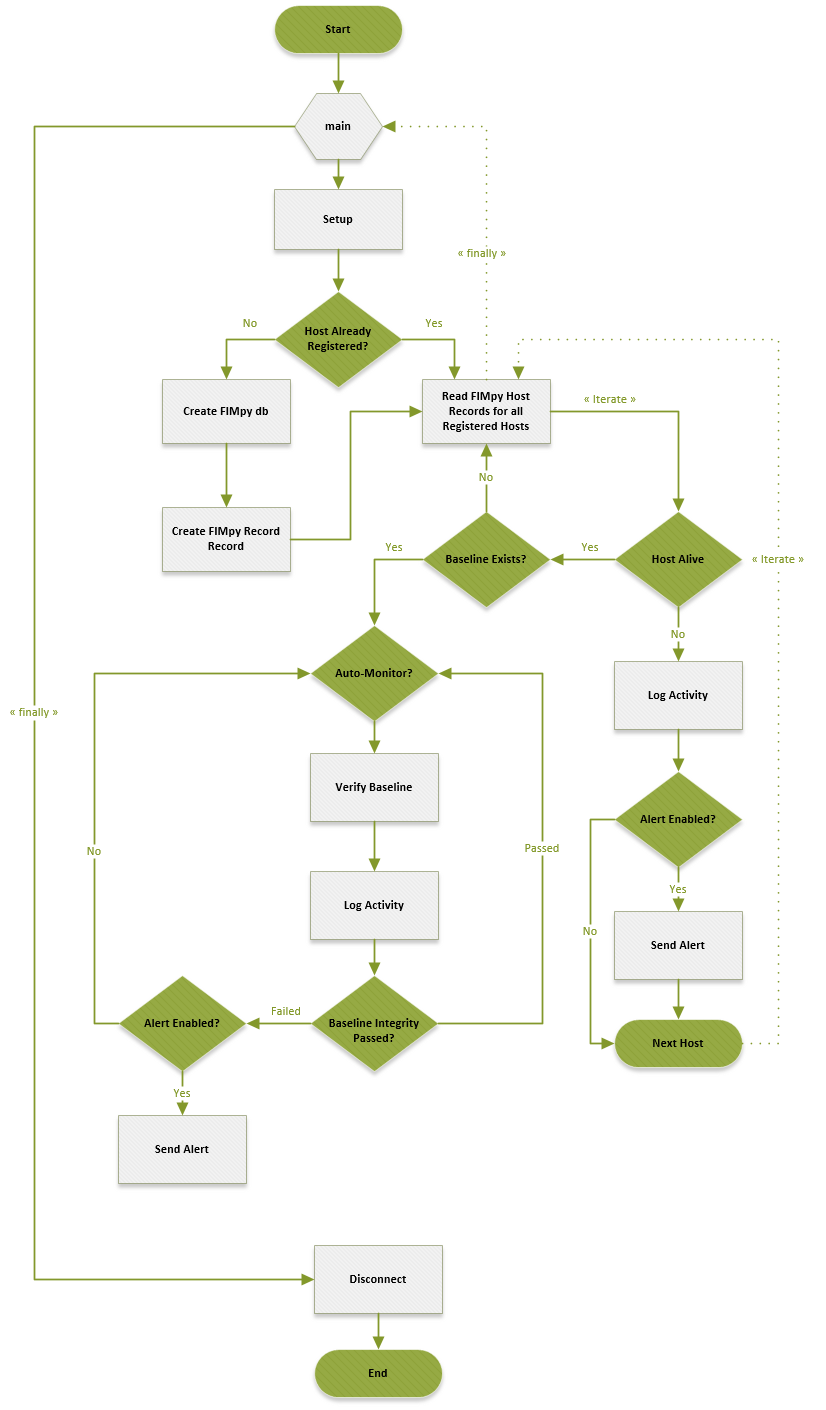


Figure 7: Process Flow Diagram

1. Implementation

The application was developed using an agile approach with continuous iteration of the base user requirements.

* 1. Python

I had very little practical Python programming experience before starting this course. Our team used it for our Semester 2 Secure Programming module project and I got to see how flexible and easy it was to pick up. There is a huge library of modules for any conceivable use case that can simply be imported and used by your Python application. Packages need to be installed locally before you can use them but again, that is simple to do.

apt-get install -y python-pip

Then just list your required packages in a file and pass that to pip as follows:

pip install -r requirements.txt

My application uses a number of modules:

Flask==0.12.2  
cf-deployment-tracker==1.0.4  
cloudant==2.4.0  
slacker==0.9.60  
pyopenssl  
python-ldap==2.4.28  
apscheduler

* 1. Python Flask

Flask is a Python web framework…it’s really simple to use and really powerful. The following lines of code are all that is required for a minimal web application (simply returning a static string to the browser).

from flask import Flask  
app = Flask(\_\_name\_\_)

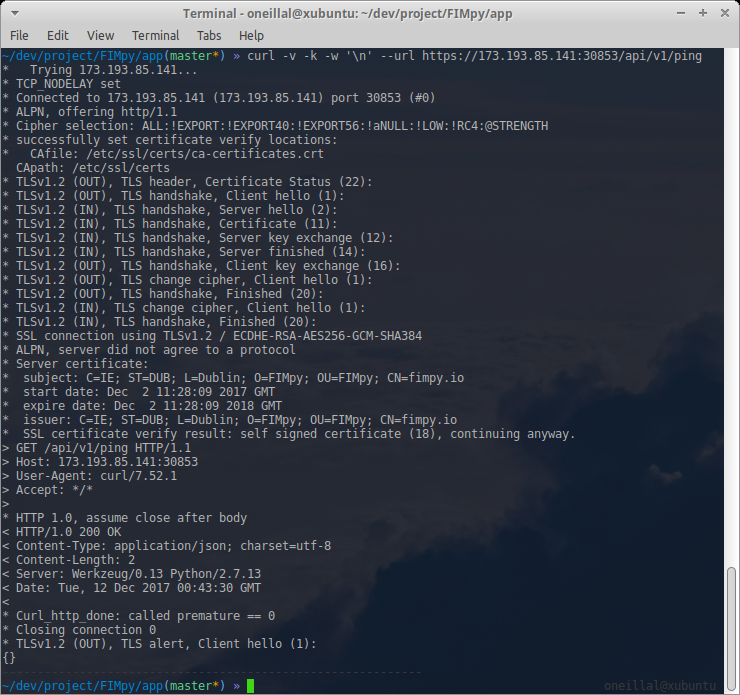
@app.route('/')  
def hello\_world():  
 return 'Hello, World!'

The following few lines of Python code enables a REST end-point that can be called over HTTP. In this sample code, I am providing an end-point that clients can call to check if the app is responding (commonly referred to heart-beats or health-checks). Here, I am just returning a HTTP response code 200 and empty JSON object but I have used this approach to create a number of different micro-service end-points to expose functionality.

@app.route('/api/v1/ping', methods=['GET'])  
def restgetping():  
 return json.dumps({}),   
 200, {'Content-Type':   
 'application/json; charset=utf-8'}

A client can then call this end-point once the application is running (output is shown in *Figure 8*).

$> curl -v -k -w '\n' --url https://173.193.85.141:30853/api/v1/ping

Figure 8: Calling a FIMpy end-point with curl

* 1. Identity management & Authentication

Using a RESTful interface to implement a micro-service architecture, core features would require some sort of authentication. I considered hard-coding authentication details but given how quickly things were progressing thanks to Pythons excellent library of modules, I looked at trying to hook up an LDAP. I first thought about running OpenLDAP in the same container. Then I looked at what was available on Bluemix. The AppID service doesn’t yet support federated identity provider support (only OAuth Google Authentication). Inversoft Passport offer a federated identity provider service on Bluemix but it’s not free. A quick Google search led me to a freemium LDAP-as-a-Service offering by JumpCloud. I signed up and read the docs on how to setup a directory and some users.

Using Python’s LDAP module was simple to integrate. Just import the module, construct the DN and pass the credentials for authentication:

import ldap

def check\_ldap(user, pw):

# TODO store LDAP org-id in IBM Key-Protect

dn = "uid=" + user +   
 ",ou=Users,o=59fee5a1d7fed97c192c12ce,  
 dc=jumpcloud,dc=com"

server = "ldaps://ldap.jumpcloud.com:636"

try:

l = ldap.initialize(server)

try:

l.bind\_s(dn, pw)

return 1

except ldap.INVALID\_CREDENTIALS:

return 0

except ldap.LDAPError, e:

print e

return -1

finally:

l.unbind()

* 1. NoSQL & Cloudant

Cloudant is very straight forward using the client module. Pass connection parameters to the constructor class to get a handle.

Then I create a database. If the database already exists, we supress the exception with throw\_on\_exists=False and the code simply continues.

import Cloudant  
  
client = Cloudant(user, password, url, connect=True)  
db = client.create\_database(dbName,throw\_on\_exists=False)

Query selectors make working with documents easy. This query returns all documents with a key ‘type’ a value that equals ‘doc’.

query = Query(db, selector={'type': {'$eq': 'doc'}})  
 for document in query.result:  
 file = document['\_id']

Manipulating documents is just as simple. The code below updates the document in the database with the id ‘host’

and sets the value of the key ‘status’ to a value of 1. A handy thing to note here also is that if a document with an id ‘host’ does not already exist, then it will be first created.

with Document(db, 'host') as host:  
 host['status'] = 1 # protecting

* 1. HMAC

It took a while to fully understand the concepts around HMAC. My Python code uses hashlib [3] and hmac [4] modules.

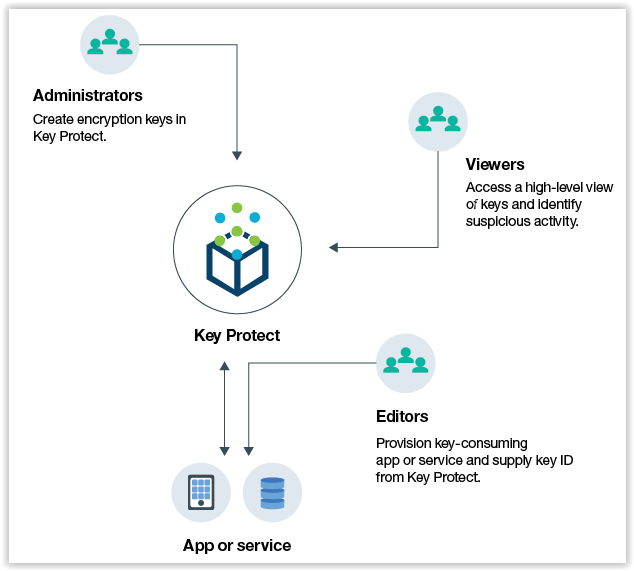
f = open(file, 'rb')  
try:  
 digest = hmac.new('key', '', hashlib.sha256)  
 while True:  
 block = f.read(BUF\_SIZE)  
 if not block:  
 break  
 digest.update(block)  
finally:  
 f.close()

With hmac and hashlib, you use the update() method because it's advisable to only read a certain number of blocks at a time and not read the whole file in one go.

* 1. Secrets Management

In order to ensure that HMAC cannot be compromised, I need to secure the key used to generate the digest. Initially, I hard-coded a key ‘mysecret’. Obviously this is completely insecure.

digest = hmac.new('mysecret', '', hashlib.sha256)



*Figure 9: System architecture*

I was eventually able to integrate IBM KeyProtect with FIMpy to use securely stored secrets for generating and verifying HMAC digests. Using the KeyProtect console, I manually generated a secret key. Then using the key reference id, I can pull the secret to be used to securely generate the HMAC by calling KeyProtect suite of API.

GET /api/v2/keys/{id} HTTP 1.1

A Bluemix IAM bearer token is needed to authenticate with the KeyProtect API. Dynamically requesting tokens for the KeyProtect service is out of scope for MVP. A valid token is accessed from a config when needed. When tokens expire, a new one will need to be generated and the app redeployed.

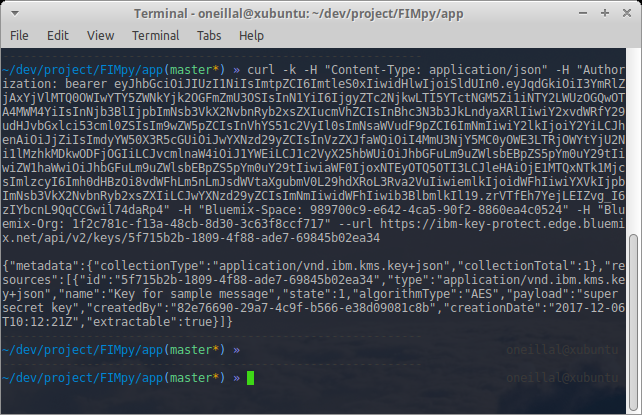


Figure 10: System architecture

Using sample code Python from IBM [5], I was able to generate HMAC with a key stored in KeyProtect.

get\_key\_request = requests.get(key\_url, headers=headers)  
get\_key\_status = get\_key\_request.status\_code  
response\_key = get\_key\_request.json()

if get\_key\_status >= 400:  
 raise Exception(get\_key\_status, response\_key)

payload\_key = response\_key['resources'][0]['payload'].encode('UTF-8')

* 1. Docker & Kubernetes

One of the earliest considerations during the implementation phase was how to approach hosting and demoing the application for the final review. I had decided early on to host on Bluemix but a limitation was that I could only have one instance running. One of the main key design aspects was that the architecture should support a multi-instance client-server implementation. That would require running multiple instances together on the same network. Another key design aspect was that the application needed to be “containerised” so that it could be deployed on Watson Health Cloud in the future.

So I knew I needed to start developing the application as a Docker “containerised” application sooner rather than later. This used up a lot of time extra time learning the [Docker Basics](https://oneillal.github.io/FIMpy/monthly-journal/mj2-containers.html) and [documenting how to build my images](https://oneillal.github.io/FIMpy/references/docker.html) and getting my [Dockerfile](https://github.com/oneillal/FIMpy/blob/master/app/Dockerfile) working. I had only a little experience using Docker up until now.

Below is the Docker file used to build the FIMpy image.

FROM ubuntu:zesty

MAINTAINER "alan.oneill75@gmail.com"

RUN apt-get update -y

RUN apt-get install -y python-pip python-dev build-essential libldap2-dev libsasl2-dev libssl-dev

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

ENV PYTHONUNBUFFERED=0

ENV TZ=Europe/Dublin

RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

EXPOSE 5000

ENV SLACKTOKEN=xoxp-263593032944-263593033264-285370171619-af047513158f7673de13a587fad919b3

#ENTRYPOINT ["python"]

#CMD ["main.py"]

CMD ["python","-u","main.py","--alert","--auto"]

The next challenge was how to deal with multiple Docker instances which wold need to run on different ports. This requires complex networking and I didn’t have enough time. The default port is 5000. If I wanted multiple Docker instances running simultaneously then every instance would need a random port. The problem then is how to determine which port other instances are running on. I could store them in the db but I tried a different approach.

I decided to go the whole way and try and run multiple instances using a Kubernetes cluster. That way, I could replicate a private network of hosts each with a FIMpy application instance running on the default port. It took a lot of extra research trying to learn how to even get started setting up a Kubernetes cluster on IBM’s new Containers implementation. I’ve documented all the reference info in my [Monthly Journal](https://oneillal.github.io/FIMpy/monthly-journal/mj2.html) and Kubernetes [Reference](https://oneillal.github.io/FIMpy/references/kube.html) info.

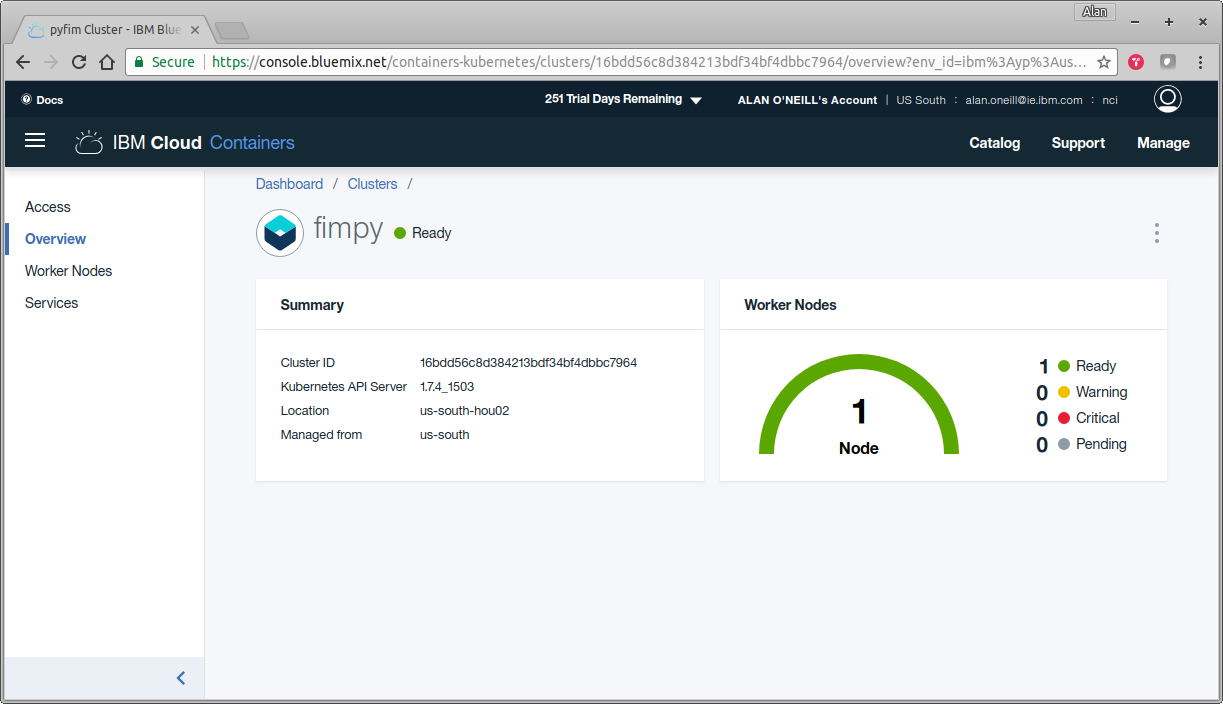
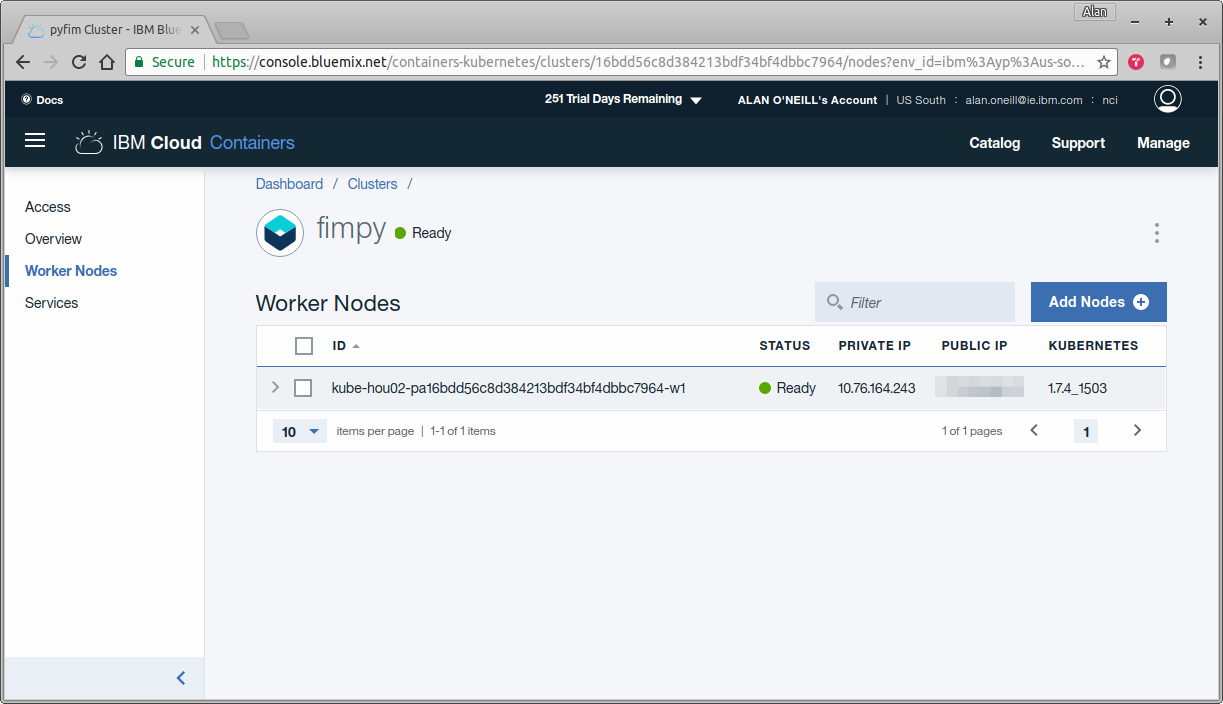


Figure 11: Console for FIMpy Kubernetes cluster running in IBM Container service  
Figure 12: IBMers can avail of a single worker node cluster

Below is my Kubernetes deployment file. It deploys a container using the alanoneill/fimpy FIMpy image published at <https://hub.docker.com/r/alanoneill/fimpy/>

apiVersion: apps/v1beta1

kind: Deployment

metadata:

name: fimpy-app

labels:

app: fimpy-app

spec:

selector:

matchLabels:

app: fimpy-app

template:

metadata:

labels:

app: fimpy-app

spec:

containers:

- name: fimpy-app

image: alanoneill/fimpy

env:

- name: SLACKTOKEN

value: "xoxp-9999"

ports:

- containerPort: 5000

The following command is used to create the deployment.

$> kubectl create -f ~/dev/project/FIMpy/kube-deployment.yaml

After a few seconds the application will be running in a kube pod.

$> kubectl get pods -o wide  
NAME READY STATUS RESTARTS AGE IP NODE

fimpy-app-1754247256-rf1vz 1/1 Running 0 26m 172.30.91.247 10.76.164.243

1. TESTING

FIMpy is a headless web application built using an micro-service architecture. The main types of testing identified as required for the project are:

* functional unit-tests
* integration testing
* security testing
* performance testing

I used GitHub during the development phase for defect management and tracking issues and enhancements.

<https://github.com/oneillal/FIMpy/issues?utf8=%E2%9C%93&q=is%3Aissue+sort%3Aupdated-desc>+

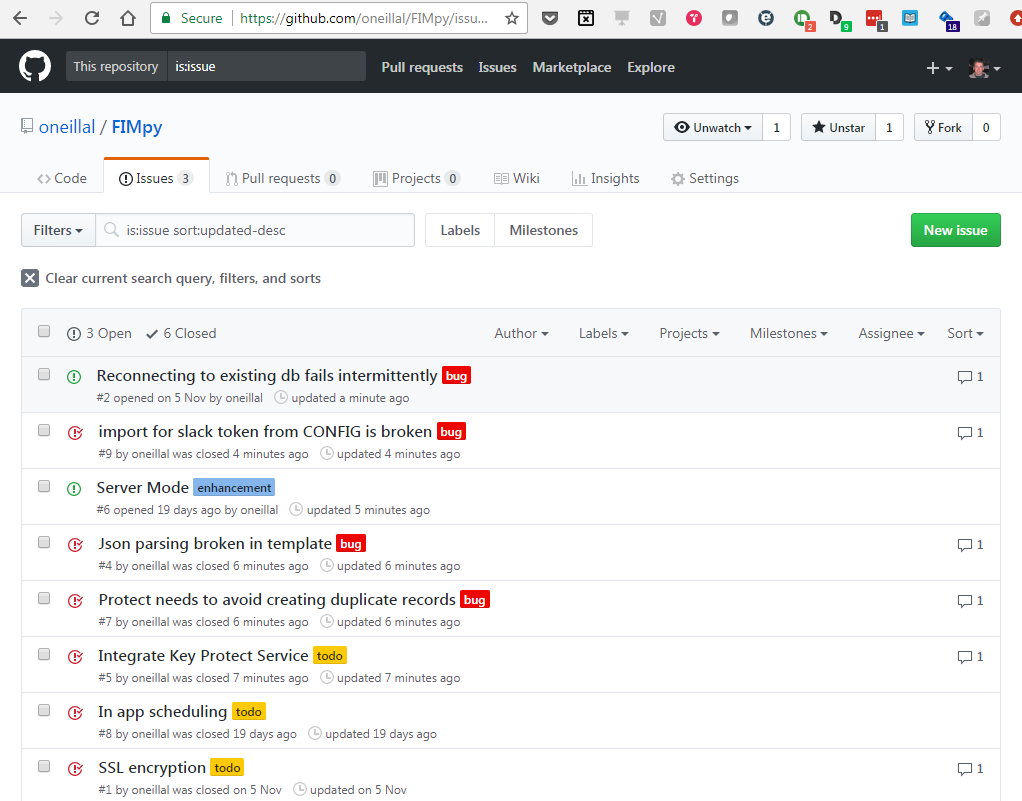


Figure 13: Issue management using GitHub

1. Unit Testing

Python has a number of testing frameworks available and I used the default unittest package that comes installed with Python.

Using reference and examples in the Python Flask documentation [6], I was able to create a unit-test suite and unit-tests

class TestCases(unittest.TestCase):

…  
…  
 def testRestGetPing(self):

# sends HTTP GET request to the application

# on the specified path

response = self.app.get('/api/v1/ping')

# assert the status code of the response

# should return 200 - OK

self.assertEqual(response.status\_code, 200)

# run test suite

unittest.TestLoader().loadTestsFromTestCase(TestCases)

unittest.TextTestRunner(verbosity=2).run(suite)

Below are some unit-test cases I am running to test the rest end-points and authentication. My overall code coverage is quite low however. There are many test cases required for a project like this and I have only implemented a small number in order to show that I have researched how to deliver effective unit testing for Python applications.

As I am using multiple databases, more time and test planning will be needed to figure out a better way to manage, create and tear down databases for unit testing. I did not have enough time to fully handle additional test cases. In the final code, I will be commenting out test-cases which involve manipulating database records in order to demo the project clearly and also with future evaluation by external examiners in mind.

import unittest

class TestCases(unittest.TestCase):

@classmethod

def setUpClass(cls):

pass

@classmethod

def tearDownClass(cls):

pass

def setUp(self):

# creates a test client

self.app = app.test\_client()

# propagate the exceptions to the test client

self.app.testing = True

def tearDown(self):

pass

def testRestGetPing(self):

# sends HTTP GET request to the application

# on the specified path

response = self.app.get('/api/v1/ping')

# assert the status code of the response

# should return 200 - OK

self.assertEqual(response.status\_code, 200)

def testRest404NotFound(self):

# sends HTTP GET request to the application

# on the specified path

response = self.app.get('/api/v1/dummy')

# assert the status code of the response

# should return 404 - Not Found

self.assertEqual(response.status\_code, 404)

def testRestAuthenticationUnauthorised(self):

# sends HTTP GET request to the application

# on the specified path

response = self.app.get('/api/v1/config')

# assert the status code of the response

# should return 401 - Unauthorized

self.assertEqual(response.status\_code, 401)

def testRestAuthenticationAuthorised(self):

# sends HTTP GET request to the application

# on the specified path

response = self.app.get('/api/v1/config', headers={'Authorization': 'Basic ' + base64.b16encode(testsuiteuser + ':' + testsuitepassword)})

# assert the status code of the response

# should return 200 - Authorized

self.assertEqual(response.status\_code, 200)

# run test suite

unittest.TestLoader().loadTestsFromTestCase(TestCases)

unittest.TextTestRunner(verbosity=2).run(suite)

During start up, the application code above will run the TestCases test suite and all its associated test-cases.

testBluemixCloudantDbConnect (\_\_main\_\_.TestCases) ... ok

testBluemixKeyProtectConnect (\_\_main\_\_.TestCases) ... ok

testRest404NotFound (\_\_main\_\_.TestCases) ... ok

testRestAuthAuthorised (\_\_main\_\_.TestCases) ... ok

testRestAuthUnauthorised (\_\_main\_\_.TestCases) ... ok

testRestGetPing (\_\_main\_\_.TestCases) ... ok

-------------------------------------------------------

Ran 6 tests in 0.328s

All unit-tests OK

\* Running on https://0.0.0.0:5000/ (Press CTRL+C to quit)

1. Debugging

The PyCharm IDE has an excellent integrated debugging client which I used extensively during development to resolve issues but also to understand a lot of the technologies involved e.g. examining responses from dependant services.

An unusual feature of the Python debugger is an automatic reloading function that will restart your application when it detects a change in the source code. This is very useful but a side effect of its use is that it needs to invoke two separate instances of the application. This lead to duplicate records and conflicts in the early phases of development. A google search indicated the use\_reloader=False to disable this feature:

app.run(host='0.0.0.0', port=CONFIG['port'], ssl\_context=context, threaded=True, use\_reloader=False, debug=True)

1. Integration Testing

For integration testing, I used curl to good effect for testing the Flask REST end-points. I could have also automated this via scripting but this was not necessary.

There are many articles on integration testing strategies in Python such as the use of Context Managers [REF] but this is outside the scope of this project.

$> curl -k -w '\n' -X POST -u "user:pass” --url <https://173.193.85.141:30853/api/v1/baseline>

{

"docs": [

{

"createdate": 1513050342.792403,

"file": "/app/test/11kfile",

"hash": "5309e677c79cffae49a65728c61b43",

"hmac": "37c34ae7c13d66abfb7f47558827dd",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"modifydate": 1509359232.0,

"size": 11264,

"status": 1,

"type": "doc"

},

{

"createdate": 1513050342.792403,

"file": "/app/test/10kfile",

"hash": "84ff92691f909a05b224e1c56abb48",

"hmac": "dc54a021fdb7e23b9bdf71c06383ff",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"modifydate": 1512961812.0,

"size": 10240,

"status": 1,

"type": "doc"

}

]

}

$> curl -k -w '\n' -X POST -u "user:pass” --url <https://173.193.85.141:30853/api/v1/verify>

{

"docs": [

{

"file": "/app/test/10kfile",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"status": 2

},

{

"file": "/app/test/11kfile",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"status": 2

}

]

}

1. Graphical User Interface (GUI) Layout

The original MVP scope of the application was intended to be all headless e.g. all API driven and no UI. I decided to implement a web console style dashboard that displays status information about registered hosts and base-line integrity.

The host serving the dashboard will “ping” all other registered hosts and determine if they are “alive”. For every responding registered host, a baseline status is retrieved and displayed.

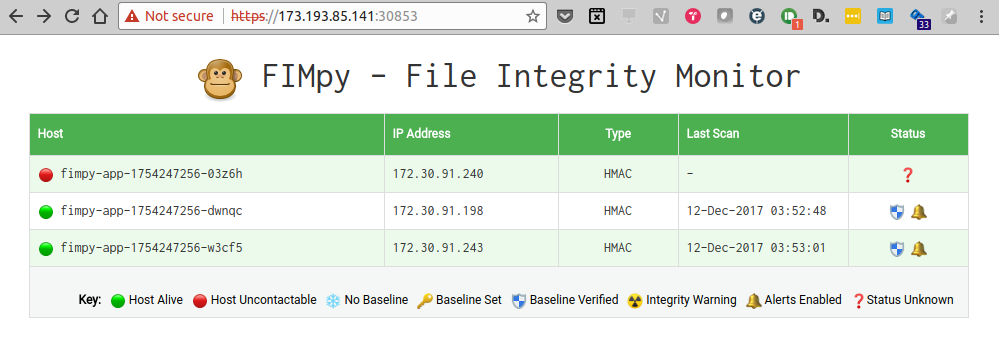


Figure 14: FIMpy Status Dashboard

|  |  |
| --- | --- |
| **FIMpy Dashboard Status**  Determine in Real-time  Determine in Real-time  Host[‘status’] == 0  Host[‘status’] == 1  Host[‘status’] == 2  Host[‘status’] == 3  Config[‘alert’] = True  True if Host Uncontactable |  |

Figure 15: FIMpy Dashboard Legend

For authentication for the REST end-points and dashboard, a simple trick I have used in the past to pop-up the login dialog when using a browser to access FIMpy endpoints e.g. ‘/’ ‘/api/v1/baseline’ etc. (see *Figure 16).*

def authenticate():

# Send a 401 response to force BasicAuth

return Response(

'Authentication failed', 401,

{'WWW-Authenticate': 'Basic realm="'Authentication required"'})

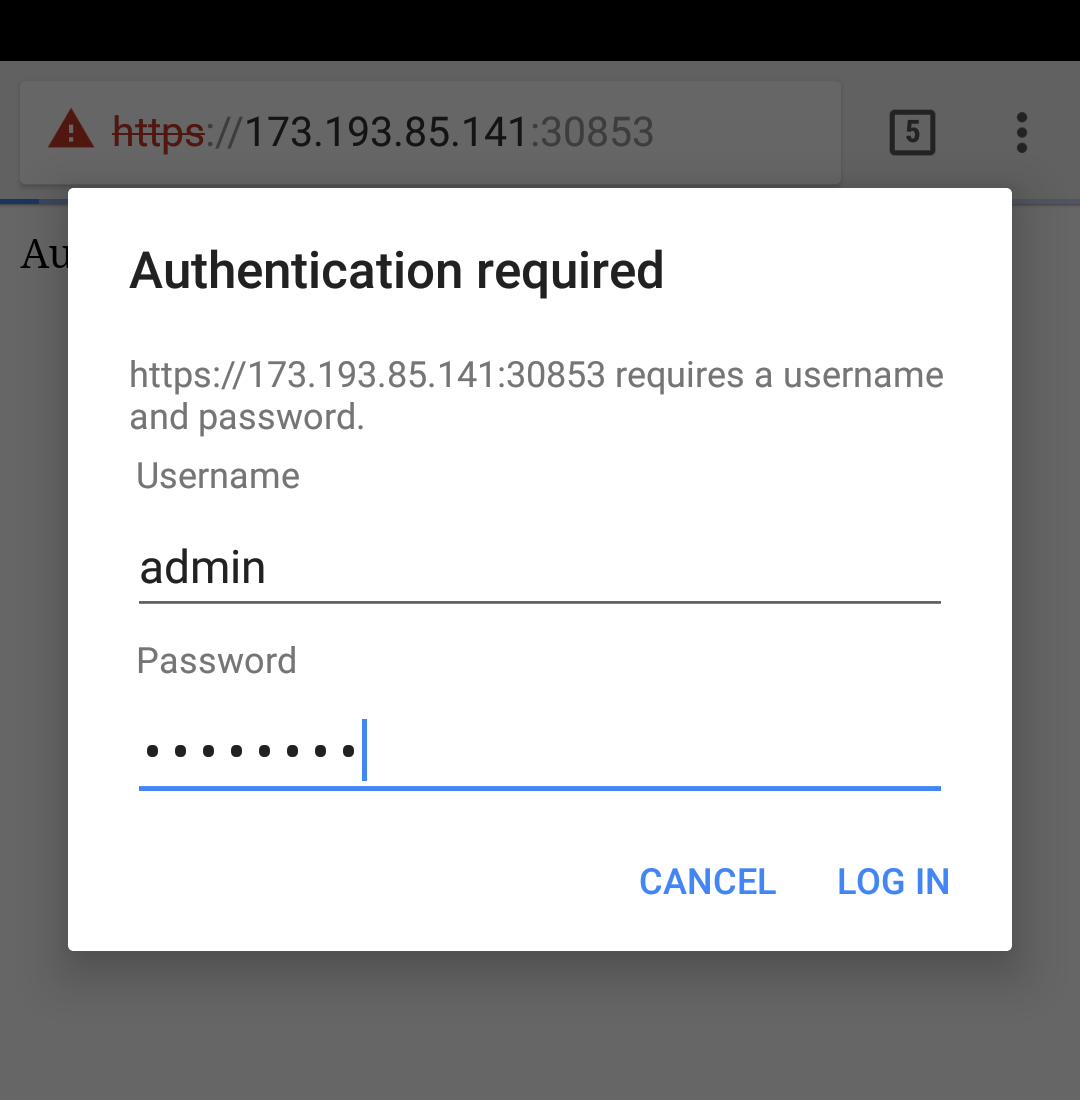


Figure 16: Dashboard browser login

Outside of the status dashboard, clients of the system are consuming data via API. I attempted to ensure that API’s follow RESTful API development best practice and that payloads are returned as simple, easy to consume JSON objects.

{

"docs": [

{

"file": "/app/test/10kfile",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"status": 2

},

{

"file": "/app/test/11kfile",

"host": "fimpy-app-1754247256-w3cf5",

"ipaddress": "172.30.91.243",

"status": 2

}

]

}

Another cosmetic user element are slack alerts generated by FIMpy. The slacker [7] client provides full rich formatting support for slack alerts. The below Python code generates the alert shown in *Figure 17*.

slack.chat.post\_message('#alerts', '', 'FIMpy', 'false', '', '', '[{"color":"#FF0000","title":"FIMpy Alert","title\_link":"https://' + host + '","text":"FIMpy has detected a potential integrity compromise with the following asset:","fields":[{"title":"Host (IP)","value":"' + host + ' (' + ipaddress + ')"},{"title":"File","value":"' + doc + '"},{"title":"Type","value":"HMAC"}]}]', '', '', '', ':face-monkey:', '')

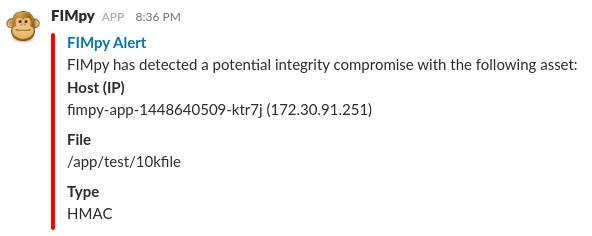


Figure 17: FIMpy alert in slack

1. Customer Testing

The application has not undergone any customer testing to-date. There a number of additional deliverables needed in order to bring the app up to a level suitable for initial security evaluation by Watson Health’s security review board i.e. security hardening, 3rd party signed CA certificates, penetration, stress and performance testing.

1. Evaluation

Upon personal evaluation, I believe the application can provide the basis for a viable FIM solution that could be further developed and enhanced into a commercial product.

On initial observation, performance is poor. Python is not a very typical language for FIM. Most other solutions are developed using lower level C derivatives. Python can be compiled to Cython [8], a superset that support C functions. This could improve performance but needs further research and evaluation.

Overall, FIM solutions are only secure as the underlying host infrastructure and OS security. If an attacker can gain root access then FIM protection can simply be disabled. The FIMpy design ensures that unless all FIMpy instances (assuming there is at least one other instance configured to auto verify integrity) are disabled simultaneously, then the application should be able to send an alert about the other uncontactable instances.

1. Further Development & Research

I will present the MVP application to my local security and if it receives positive feedback, I will seek approval from my senior management to take the project further.

1. Conclusions

Overall, I am very happy with the outcome of my project. I realised after a few weeks that I had set myself a difficult challenge with my chosen topic of File Integrity Monitoring. My design evolved into a complex one overtime and the decision to use multiple third party services of which I had little to no experience with meant many extra hours of learning. Some of the services were also unstable from time to time which hindered progress. There was always an element of doubt that things may go wrong but it worked out in the end and I am very happy with the outcome.

This project also presented me with the opportunity to expand my technical skills and gain experience with many new technologies. I am especially happy with the level of understanding I have gained around Docker and container orchestration using Kubernetes. This will stand to me in my day to day work as we migrate our Watson Health applications to the IBM Cloud which is underpinned by Docker and Kubernetes. I am also pleased to have exposed myself to NoSQL which I am very intrigued about.

I am also very much impressed with the flexibility and power of the Python programming language and I have already begun to use it in day to day work. Also, Python Flask is great choice for web application and API development.

1. References
2. BBC News. (2017). Cyber-attack 'unprecedented' in scale. [online] Available at: http://www.bbc.com/news/world-europe-39907965 [Accessed 12 Dec. 2017].
3. Ibm.com. (2017). IBM Guardium for File and Database Encryption - Overview - United States. [online] Available at: https://www.ibm.com/us-en/marketplace/guardium-file-and-database-encryption [Accessed 12 Dec. 2017].
4. Pypi.python.org. (2017). hashlib 20081119 : Python Package Index. [online] Available at: https://pypi.python.org/pypi/hashlib/20081119 [Accessed 12 Dec. 2017].
5. Pypi.python.org. (2017). hmac 20101005 : Python Package Index. [online] Available at: https://pypi.python.org/pypi/hmac/20101005 [Accessed 12 Dec. 2017].
6. GitHub. (2017). IBM-Cloud/key-protect-helloworld-python. [online] Available at: https://github.com/IBM-Bluemix/key-protect-helloworld-python [Accessed 12 Dec. 2017].
7. Flask.pocoo.org. (2017). Testing Flask Applications — Flask Documentation (0.12). [online] Available at: http://flask.pocoo.org/docs/0.12/testing/ [Accessed 12 Dec. 2017].
8. GitHub. (2017). os/slacker. [online] Available at: https://github.com/os/slacker [Accessed 12 Dec. 2017].
9. Cython.org. (2017). Cython: C-Extensions for Python. [online] Available at: http://cython.org/ [Accessed 12 Dec. 2017].
10. Anon, (2017). [online] Available at: http://eigenhombre.com/introduction-to-context-managers-in-python.html [Accessed 12 Dec. 2017].
11. Anon, (2017). [online] Available at: http://bobthegnome.blogspot.co.uk/2007/08/making-ssl-connection-in-python.html [Accessed 12 Dec. 2017].
12. Apscheduler.readthedocs.io. (2017). User guide — APScheduler 3.4.0.post17 documentation. [online] Available at: https://apscheduler.readthedocs.io/en/latest/userguide.html [Accessed 12 Dec. 2017].
13. Console.bluemix.net. (2017). IBM Cloud Docs. [online] Available at: https://console.bluemix.net/docs/services/Cloudant/getting-started.html [Accessed 12 Dec. 2017].
14. Docs.python.org. (2017). 15.5. optparse — Parser for command line options — Python 2.7.14 documentation. [online] Available at: https://docs.python.org/2/library/optparse.html [Accessed 12 Dec. 2017].
15. Docs.python-requests.org. (2017). Quickstart — Requests 2.18.4 documentation. [online] Available at: http://docs.python-requests.org/en/master/user/quickstart [Accessed 12 Dec. 2017].
16. Easyicon.net. (2017). face monkey PNG/ICO/ICNS Multi Size Icons Download,face monkey ICON DOWNLOAD\_easyicon.net. [online] Available at: https://www.easyicon.net/language.en/541685-face\_monkey\_icon.html [Accessed 12 Dec. 2017].
17. Flask, S. (2017). Scheduling a function to run every hour on Flask. [online] Stackoverflow.com. Available at: https://stackoverflow.com/questions/21214270/scheduling-a-function-to-run-every-hour-on-flask [Accessed 12 Dec. 2017].
18. Flask.pocoo.org. (2017). How to serve HTTPS \*directly\* from Flask (no nginx, no apache, no gunicorn) | Flask (A Python Microframework). [online] Available at: http://flask.pocoo.org/snippets/111/ [Accessed 12 Dec. 2017].
19. Flask.pocoo.org. (2017). HTTP Basic Auth | Flask (A Python Microframework). [online] Available at: http://flask.pocoo.org/snippets/8/ [Accessed 12 Dec. 2017].
20. GitHub. (2017). IBM-Cloud/get-started-python. [online] Available at: https://github.com/IBM-Bluemix/get-started-python [Accessed 12 Dec. 2017].
21. Hellmann, D. (2017). hmac – Cryptographic signature and verification of messages. - Python Module of the Week. [online] Pymotw.com. Available at: https://pymotw.com/2/hmac/ [Accessed 12 Dec. 2017].
22. Jinja.pocoo.org. (2017). Template Designer Documentation — Jinja2 Documentation (2.10). [online] Available at: http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters [Accessed 12 Dec. 2017].
23. Packtpub.com. (2017). Configuring and securing PYTHON LDAP Applications | PACKT Books. [online] Available at: https://www.packtpub.com/books/content/python-ldap-applications-part-1-installing-and-configuring-python-ldap-library-and-bin [Accessed 12 Dec. 2017].
24. Appendix

|  |  |
| --- | --- |
| Initial Project Proposal |  |
| Mid-Term Presentation |  |
| Agile Project Plan |  |
| FIMpy User Documentation | <https://oneillal.github.io/FIMpy/> |
| Monthly Journals | <https://oneillal.github.io/FIMpy/> |