Project no. 826278

SERUMS

Research & Innovation Action (RIA)
SECURING MEDICAL DATA IN SMART PATIENT-CENTRIC HEALTHCARE SYSTEMS

Report on Initial Specification of Smart Patient Health Record Format
D2.2

Due date of deliverable: 31 October 2019

Start date of project: January 1st, 2019

Type: Deliverable
WP number: WP2

Responsible institution: Sopra-Steria Ltd.
Editor and editor's address: 30 Queensferry Road, Edinburgh EH4 2HS, United Kingdom

Version 1.0

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	30/10/19	AF Vermeulen	SOPRA	Version 001.000
2	30/10/19	E Blackledge	SOPRA	Version 001.000

Executive Summary

SERUMS proposes a unified Smart Patient Health Record (SPHR) that is capable of both securely storing data from any healthcare provider in a consistent manner, as well as the secure transmission of this data to only approved healthcare providers. Underlying this is the need for the patient to control who has access to their data, whilst still complying with all relevant legislation.

The SERUMS solution enables a patient to apply their consent to their own healthcare data using a range of role-base data sharing agreements via smart contracts stored on an identity block-chain. Any interactions with the system is stored for audit on the block-chain.

The system supports the future requirements for European wide healthcare by enabling the citizens to control their own healthcare data.

Contents

Chapter 1

Introduction

This deliverable is comprised of *T2.2: Storage and Access Control for Smart Patient Records, T2.3: Blockchain for Smart Patient Records, and T2.4: Serums Data Lake and Metadata Extraction*. Having already defined the format for the Smart Patient Health Record in T2.1, this deliverable is concerned with providing mechanisms for regulating access to it, providing mechanisms for tracking the lineage and provenance of the data using a blockchain approach, and develop new machine-learning mechanisms for structuring data into the patient records. This deliverable contains the initial versions of the software.

Chapter 2

Smart Patient Health Format

Our three (3) use case partners have their own proprietary formats for storing data within their own systems. With the ultimate goal of combining their data in a single record, it is necessary that we design a common format. As described in *D2.1 - Report on Initial Specification of Smart Patient Health Records* we have designed a format for the Smart Patient Health Record based on the data vault 2.0 framework.

The data vault is made up of three core table types in the database. These are Hubs, which hold the primary and foreign keys, Links, which join the Hubs via many-to-many relationships, and Satellites, which join onto the Hubs and contain the descriptive data.

In our format, we have chosen five types of Hubs. These are Time, Person, Object, Location, and Event (T-P-O-L-E). The Satellites are therefor all related to one of these five core categories hubs. The Satellites are then further grouped into smaller sets that contain closely related data.

As a way of demonstrating the process, we can use the example of the source NDC SMR01 as seen in figure 2.1 from the USTAN use case. We then abstract metadata out via a process described in the technical implementation section on data vault conversion 3.3.3. The output of this can be seen in figure 2.2.

The main advantage of this format is that as we add more data sources at any point in future and we can always join onto the relevant Hub. This allows the Satellites of a Hub to be easily checked for similar data. This can either be by a manual process or handled by machine learning. If data from two Satellites is deemed to be similar enough then these can then be grouped together into a single new Satellite to optimise the data. For instance we might have drug prescriptions from two health centres. When they are added the the SPHR they would join as two separate Satellites to the Object Hub. By combining them into a single Satellite we would ensure that we could deliver the complete record of prescription drugs for the patient as a universal summary view.

A further advantage of our data vault format is how it will compliment the use of tags in the smart contracts as describe in section 3.4. We enforce a standard

naming convention for the Satellites that follow the pattern *SAT HUB NAME - MAJOR GROUPING MINOR GROUPING* for example in figure 2.2 we can see the Satellites *sat_time_admissions_date* and *sat_object_operations_details*. When the patient or health centre design smart contracts, they add tags as part of the process. If for instance a tag for admissions or operations was added, the data vault would know immediately which data to package up and return.

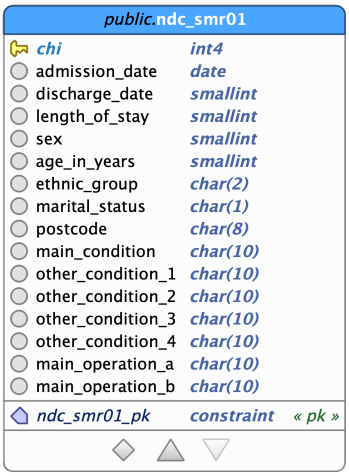

Figure 2.1: NDC SMR01 Source Table

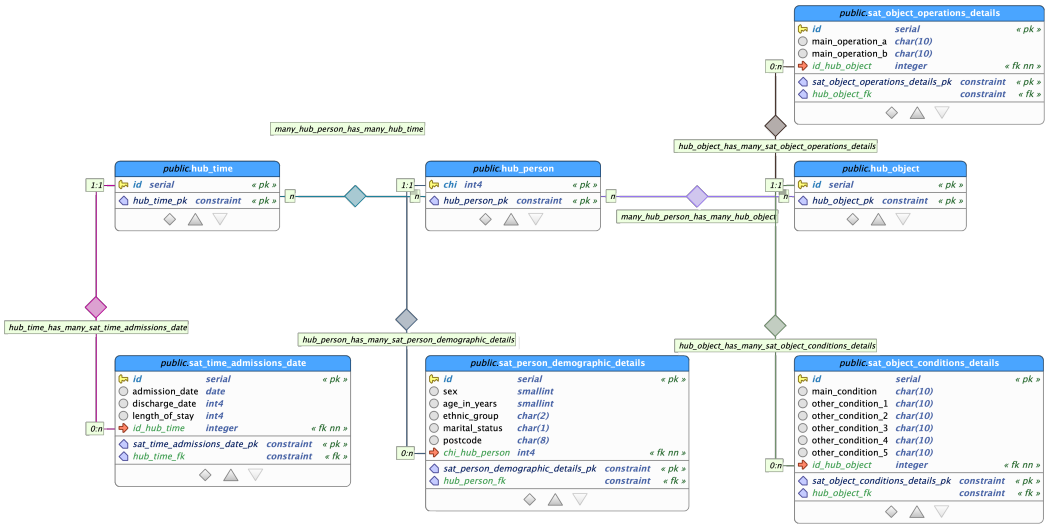

Figure 2.2: NDC SMR01 Data Vault

Chapter 3

Technical Implementation

3.1 Overview

In this chapter we will present an overview of the individual software components, as well as how they will combine to answer the requirements of this deliverable. The tasks will be broken down into four sections covering Storage, Access Control and Block-chain, Machine-Learning and Metadata, and how these combine into an integrated effective solution.

3.2 The St.Andrews Virtual Machine

3.2.1 Introduction to the Virtual Machines

The environment that has been chosen for the initial version of the software are virtual machines (VMs) running on servers inside of the St.Andrews campus. This has brought with it cost savings while we are still early in development. Within the St.Andrews VMs we have separate spaces set up for the Block-chain and Data Lake development.

The Data Lake development is taking place within a Docker image. Specifically it is one that has been obtained from Cloudera, a leader in cloud-based technologies. The image itself contains a highly robust Hadoop instance, giving us out of the box functionality and bypassing a lot of the difficulties associated with the manual setup of a custom Hadoop cluster.

This Docker image does come with some downfalls however. The OS is CentOS7 and the image is limited to Python34. This has meant that there is often some wrangling involved to get the necessary python packages to work, with a special focus involved in making sure the dependencies are correctly specified for legacy versions of the packages.

The future stages of the project will resolve some of these shortfalls to result into a more effective and efficient final solution.

The follows instructions is used in order to replicate the current setup.

3.2.2 Signing into Fracas

- We need to make sure our ssh key has been added to both Bigmill and Fracas
- Now we can sign into Bigmill:

```
$ ssh -X -Y euanblackledge@bigmill.cs.st-andrews.
   (cont.)ac.uk -v
```

- Now we can sign into Fracas:

```
$ ssh -Y -X -p 2203 euanblackledge@fracas.cs.st-
   (cont.)andrews.ac.uk -v
```

The port 2203 is the endpoint for the Data Vault

3.2.3 Install Docker

- First we grab the password:

```
$ cat passwd.txt
```

- Now we need to make sure we are up to date:

```
$ sudo apt-get update
```

- Now we install dependecies:

```
$ sudo apt-get install \
apt-transport-https \
ca-certificates \
curl \
gnupg-agent \
software-properties-common
```

- Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/
  (cont.)ubuntu/gpg | sudo apt-key add -
```

- Verify fingerprint:

```
$ sudo apt-key fingerprint 0EBFCD88
```

- Install stable repository:

```
$ sudo add-apt-repository "deb␣[arch=amd64]␣https
  (cont.)://download.docker.com␣linux/ubuntu␣$(
  (cont.)lsb_release -cs) stable"
```

- Now install Docker and its tools then verify it works:

```
$ sudo apt-get install docker-ce docker-ce-cli
  (cont.)containerd.io
$ sudo docker run hello-world
```

If this runs successfully then it is time to get the Cloudera CDH image

3.2.4 Install Cloudera CDH Image

- Run this to pull the image:

```
$ docker pull cloudera/quickstart:latest
```

- Check the image ID for the Cloudera Docker image:

```
$ sudo docker images
```

This will return something similar to the table 3.1.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cloudera/quickstart	latest	914bab9e5b49	3 years ago	6.34GB

Table 3.1:

- Use that image id in the following command:

```
$ sudo docker run --hostname=quickstart.cloudera
   (cont.)--privileged=true -t -i -p 8888:8888 -p
   (cont.)7180:7180 -p 8088:80 914bab9e5b49 /usr/
   (cont.)bin/docker-quickstart
```

In this example "914bab9e5b49" is the image ID

3.2.5 Python and Pandas

Out of the box Python2 is installed however for ease of development across platforms we need to install Python3 and Pandas. Note the versions required for this setup. Currently the VM is only capable of running Python3.4 which is no longer supported by Pandas

```
$ sudo yum install python34-setuptools
$ sudo yum install python-devel
$ sudo easy_install-3.4 pip
$ pip install numpy==1.7.0
$ pip install pandas==0.19.0
$ cat pip install mysql-connector-python
```

3.2.6 PyHive

In order to run PyHive we need to first install the dependencies:

```
$ sudo yum install gcc
$ sudo yum install python34-devel
$ sudo yum install libevent-devel
$ pip install thrift
$ pip install sasl
$ pip install thrift_sasl
$ pip install pyhive
```

This enables the software to support access to Hive directly from our python code to utilise for processing the source data and data vault as a single data lake.

3.3 Storage

3.3.1 Choice of Technologies

For the initial version of storage, a combination of Hadoop running alongside MySQL has been chosen. These two technologies have been chosen for two key reasons:

Firstly, they are both available for free. During this initial stage, where we are still very much researching and testing different tools and techniques, it is important that we keep the associated costs to a minimum. Towards the end Serums we will aim to move to an at-scale cloud solution, like AWS or Azure, in order to improve speed, scaling, security, and improved access.

Secondly, they are natural bedfellows. Hadoop allows us to store just about anything, from csvs to images. However, in order to combine these into the SPHR, we can make the process a lot easier if we utilise a more traditional relational database like MySQL.

All software code are developed in Python3 to open it to more developers for ease of extension in future.

3.3.2 Data Lake Setup

For each of the use case partners, we have set up their own data lake. This allows for each partner to have their own space for the storage and processing of data. This process is handled by a piece of code named *0100-SERUMS-RIF-DL-Hadoop-Setup*.

This code defines the structure of the data lake 4 based on the Rapid Information Factory (RIF) framework 5. The process to run this can be run from the terminal within the St. Andrews Virtual Machine 3.2.6. Within the folder *code/0100-SERUMS-RIF-DL-Hadoop-Setup* type:

```
$ python3 setup.py FCRB
```

This will result in the RIF structure being created within Hadoop with the prefix /FCRB. The code can be found in Appendix A .1

3.3.3 Converting Raw Files to Data Vault

The process is illustrated in the figure 3.1. As can be seen, there are four sets of code that transform the source data into the data vault format. These are as follows:

- Adding Files to Raw
- CSV to MySQL
- Raw to MySQL

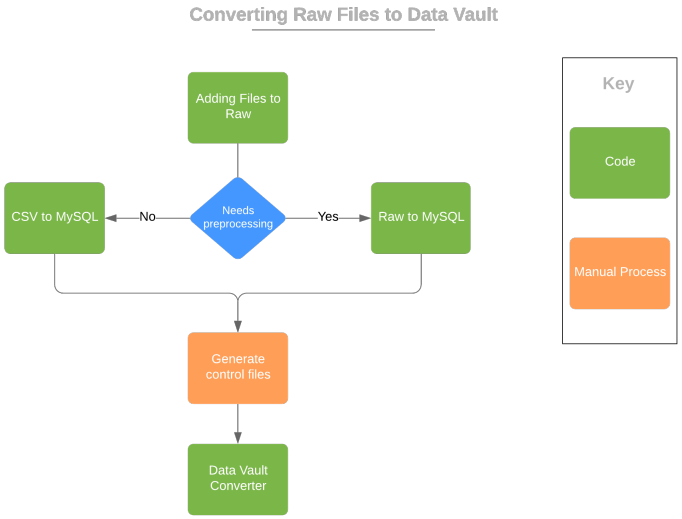

Figure 3.1: Processing Raw Files

- Data Vault Converter

Adding Files to Raw is concerned with moving files into their endpoint in Hadoop. This can handle any type of file, however at this stage of development we are mostly concerned with either csvs or text files.

Within the folder *code/adding_files_to_raw* type:

```
$ python3 adding_files_to_raw.py -l "data/
  (cont.)fcrb_source" -f medication.csv -d "/FCRB
  (cont.)/100-DL/100-Raw-Zone/200-Internal/100-
  (cont.)CSV/"
```

This would result in the file *medication.csv* being moved from *data/fcrb_source* to the end point */FCRB/100-DL/100-Raw-Zone/200-Internal/100-CSV/* inside of Hadoop. The code for this can be found within Appendix B .2

Raw to MySQL is concerned with taking the files that have been placed in the Raw Zone of Hadoop and converting them to tables within MySQL. As part of this the program does some prepossessing to determine some of the data types, making the final table in MySQL a more accurate representation of the source material. For instance if a csv has been imported and every field has been classed as a text field, this program will determine if any can be reclassified as integers or dates.

Within the folder *code/raw_to_mysql* type:

```
$ python3 raw_to_mysql.py -u root -p -l /FCRB/100-
  (cont.)DL/100-Raw-Zone/200-Internal/100-CSV/ -d
  (cont.) fcrb
```

This would result in the files found in */FCRB/100-DL/100-Raw-Zone/200-Internal/100-CSV/* to be converted into tables within the *fcrb* database within MySQL. The code for this can be found within Appendix C .3

CSV to MySQL is concerned with taking well structured csvs from the Raw Zone of Hadoop and converting them to tables within MySQL. While it works almost identically to *Raw to MySQL*, by limiting the amount of preprocessing required, it is a much faster program to run. In a future version of the project, these two programs will be wrapped into a single tool that is capable of deciding which would be the best choice to use. *CSV to MySQL* proved to work very well with the synthetic data provided by IBM.

Within the folder *code/csv_to_mysql* type:

```
$ python3 csv_to_mysql.py -u root -p -l /USTAN
   (cont.)/100-DL/100-Raw-Zone/200-Internal/100-
   (cont.)CSV/ -d ustan
```

This would result in the files found in */USTAN/100-DL/100-Raw-Zone/200-Internal/100-CSV/* to be converted into tables within the *ustan* database within MySQL. The code for this can be found within Appendix D .4

Data Vault Converter is concerned with transforming the tables found within MySQL into the data vault format. This is the final step in the processing of the raw data which allows us to begin building up the SPHR. If we take the source table for NDC SMR01 2.1 from the USTAN use case, we are aiming to abstract it out into the a data vault version of itself as seem the the NDC SMR01 Data Vault figure 2.2.

Unlike the previous sections, this requires a little bit of manual processing to ensure that the tables are correctly created. The user must create two csv control files and place them correctly within the file structure. In the example of the NDC SMR01 table as seen in figures 2.1 and 2.2 we must feed into the program a list of the columns, including their destination table name and the key of that table, as well as a list of the the hubs. These can be seen in tables 3.2 and 3.3 respectively.

At the moment these tables require a human-in-the-loop with some understanding of the underlying data. It is one of the design requirement for Serums is that we are able to automate this part of the process using Machine Learning to better improve the speed at which data sources can be introduced to the overall system.

name	destination	foreign_key
admission_date	sat_time_admissions_date	id_hub_time
discharge_date	sat_time_admissions_date	id_hub_time
length_of_stay	sat_time_admissions_date	id_hub_time
sex	sat_person_demographic_details	chi_hub_person
age_in_years	sat_person_demographic_details	chi_hub_person
ethnic_group	sat_person_demographic_details	chi_hub_person
marital_status	sat_person_demographic_details	chi_hub_person
postcode	sat_person_demographic_details	chi_hub_person
main_condition	sat_object_conditions_details	id_hub_object
other_condition_1	sat_object_conditions_details	id_hub_object
other_condition_2	sat_object_conditions_details	id_hub_object
other_condition_3	sat_object_conditions_details	id_hub_object
other_condition_4	sat_object_conditions_details	id_hub_object
main_operation_a	sat_object_operations_details	id_hub_object
main_operation_b	sat_object_operations_details	id_hub_object

Table 3.2: NDC SMR01 Column Control File

hub	primary_key
hub_person	chi
hub_time	id
hub_object	id

Table 3.3: NDC SMR01 Hub Control File

3.4 Access Control and Blockchain

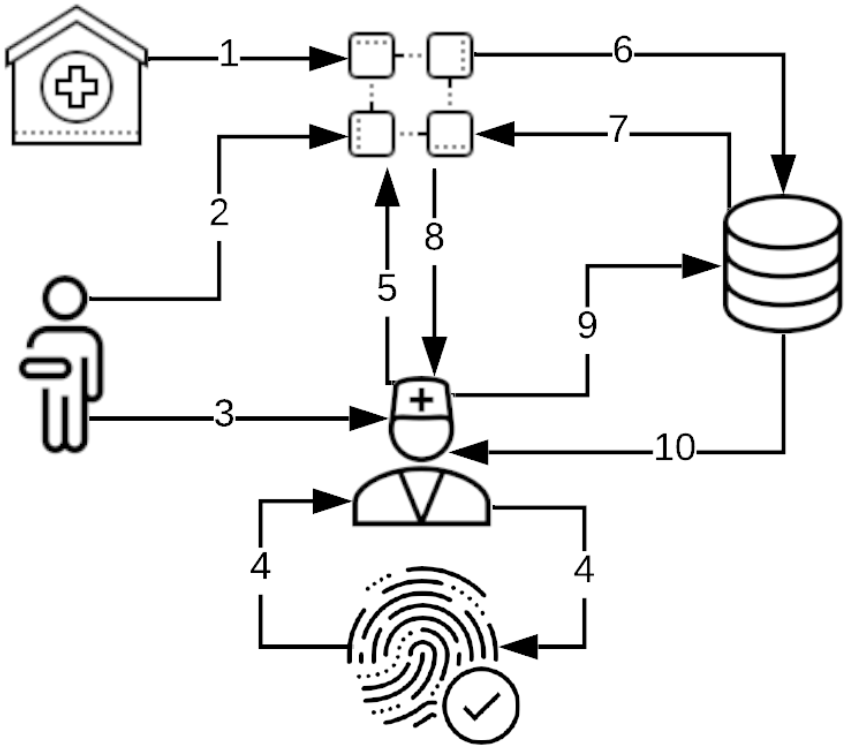

Figure 3.2: Block-chain and Access Control

3.4.1 Overview of Blockchain and Access Controls

As well as using blockchain to record access requests to the data vault, we are also using a feature of the Hyperledger Fabric framework known as smart contracts. Smart contracts work in similar way to regular contracts in that an agreement is set up between parties to which all parties concerned must adhere to. In other words, a smart contract contains a list of criteria which are checked when the blockchain is accessed and will only execute if all of the criteria have been met.

For Serums we have designed an API which allows both patients and hospitals the ability to set rules around the data that will limit who can see what and where the data may be sent by european law. For instance, in the ZMC use case, we have a patient interested in sharing their data with an external specialist regarding

a recent leg operation. As such a smart contract would be initialised by the patient that contained the id for the doctor, the id of their health centre, and the tags for the parts of the data that the patient wishes to share. In this case the tags would likely be "LEG", "OPERATION", "BASIC PERSONAL". The exact nomenclature for these tags has yet to be decided upon, however this should illustrate the principle behind the idea.

The specialist would then make a request to the Serums system. Having been authenticated, they would then be passed to the blockchain integration layer which would check their permissions against the smart contracts. Once the smart contract validated the rules for that specialist's access in the data vault, the blockchain would then pass this information over to the data vault.

At this stage the data vault would gather the relevant data for the patient, encrypt it, store it in a secure location and pass the key via the blockchain integration layer, and onto the specialist. The Serums system would then contact the data vault directly, asking for the encrypted data. At this point the data would then be delivered via some form of secure transfer such as SFTP, allowing the specialist to use the key they had already received to decrypt the data. This multi-phase encryption secures the data against unauthorised access without consent.

This process gives us the ability to record any requests to and from the complete system. Additionally it allows us to increase the security by passing the key and data by two distinct paths. The use of these alternative routes results in that even if the key were to be intercepted, an attacker would also have to intercept the data separately and match them in real-time.

This process is illustrated in in figure 3.2. The steps can be described as follows:

1. Hospital is added to the smart contract and initial rules are added following local legislation governing data sharing controls

2. Patient is added to the smart contract

3. Patient creates rule allowing healthcare professional access to a limited selection of their data

4. Healthcare professional authenticates themselves to the Serums system

5. The healthcare professional access rules are validated against the smart contract

6. The blockchain integration layer sends a request to the data vault for the relevant data

7. The data vault encrypts the data and sends key to the blockchain integration layer

8. The key is passed to the healthcare professional

9. The Serums system makes a request to the data vault for the encrypted data

10. The data vault passes the encrypted data to the healthcare professional where it can be decrypted by the previously sent key

3.4.2 Hyperledger Fabric

Blockchain is a programmable, distributed ledger with an immutable history of transactions. For every transaction consensus has to be reached among the participating organisations (or commonly denoted as nodes) before it can be written on the ledger.

Blockchain is programmable via the notion of a smart contract that is simply a piece of code, that is installed and executed within the blockchain network; the execution of a smart contract's function creates a transaction. Note that the transaction is written on the ledger of each node concurrently. Consequently, the ledgers are always synchronised. If a node has some downtime, when it restarts, it automatically synchronises its ledger to the ledgers of the rest of the nodes. In addition, a single ledger (of a single node) cannot be tampered unless the attacker can manage to concurrently infiltrate at least the majority (if not all) of the nodes, depending on the consensus protocol used.

In the proposed architecture a blockchain network is created where every relevant organisation (e.g a hospital) participates. The user's permissions that control access for the SPHR are programmed using smart contracts. This allows versatility as the rules used to form the permissions can be updated whenever required. However, due to the blockchain's nature, a single organisation cannot force an update of these rules as transactions will not be able to reach consensus and inevitably will not be written on the ledger.

For this version of the Serums blockchain, the smart contacts have been written in Hyperledger Fabric. Hyperledger Fabric has the advantage of being an API. As such we have known end points that we are able to interact with, handling all of our requests. This is in comparison to something like Ethereum smart contracts which requires the developer to know a coding language called solidity, and then to write all of the functions themselves.

3.5 Blockchain Setup

Full details of the blockchain setup process can be found in Appendices F .6, G .7, H .8, and I .9. This includes the instructions to deploy the blockchain itself, the frontend to interact with it, the backend that links them, and an ansible instance which helps to handle to deployment across multiple servers.

3.6 Machine Learning and Metadata

3.6.1 Overview of Machine Learning and Metadata Extraction

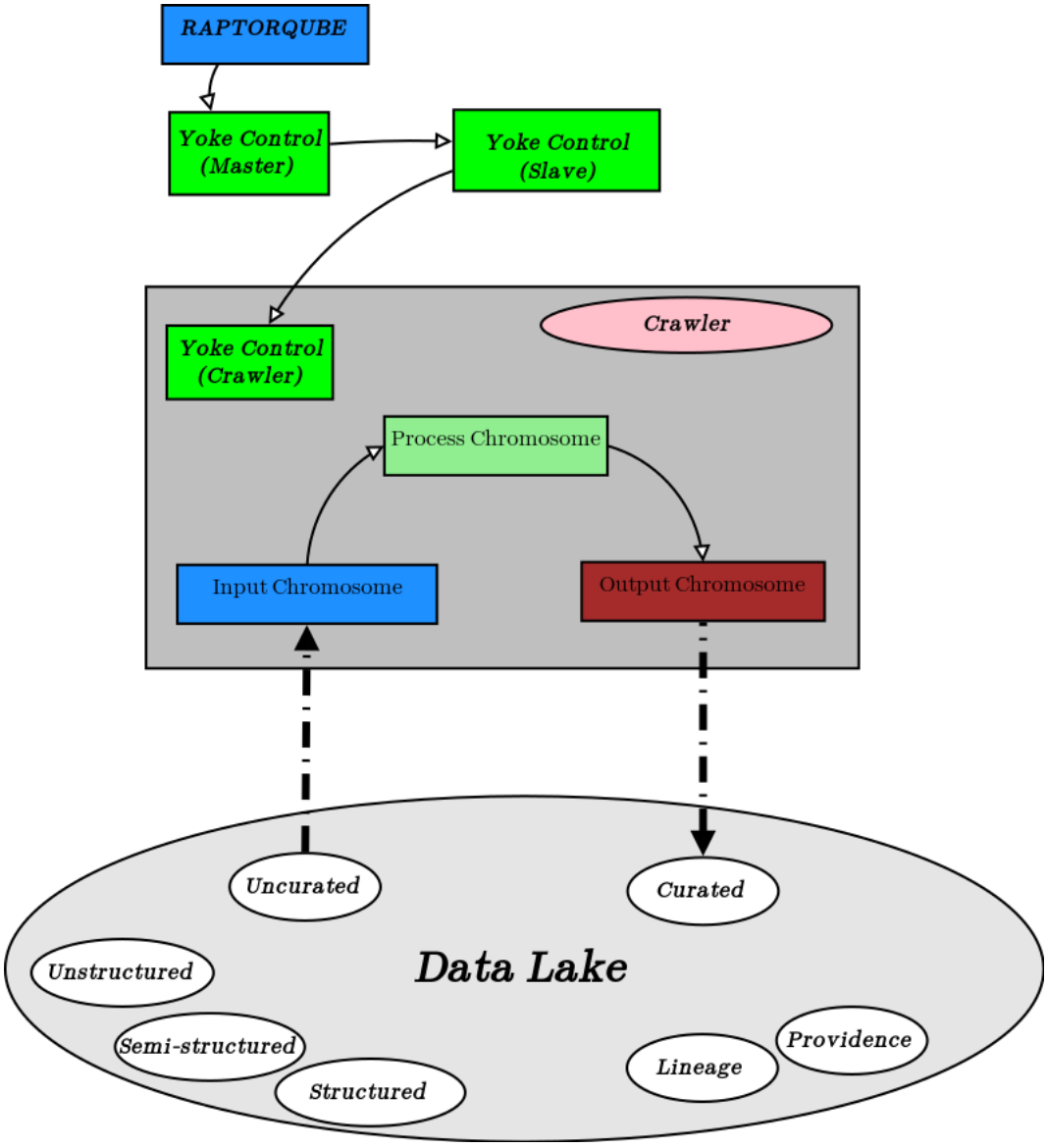

Figure 3.3: Machine Learning and Metadata Extraction Vision

Machine learning is the application of artificial intelligence to enable a system the ability to automatically learn and improve through repeated running.

Metadata is additional descriptive attributes for a piece of data. For instance, if

we took the value "seven" we could attach some attributes to it that might contain things such as that it is a string, it is five characters in length, and that it also contains a number. Using these attributes we can classify data in many additional ways. Furthermore, using these classifications, we can begin to look for underlying patterns and similarities that might not have been initially obvious.

By feeding metadata into machine learning algorithms, we can automate this classification and pattern finding.

3.6.2 Machine Learning and Metadata in Serums

One of the aims of this work package is to automate the processing of data, enabling the automatic creation and curation of the Smart Patient Health Record (SPHR). To achieve this we will be utilising a combination of machine learning tools and techniques. These will be used to extract metadata from the synthesised data provided by WP5, classify the results under one of our targets of Time, Person, Object, Location, or Event (T-P-O-L-E), and look for similarities in order to form the Satellites of the SPHR as described in chapter 2.

Figure 3.3 illustrates our ultimate goal for the machine learning model. This will link the data lake with the rapid information factory as described in chapters 4 and 5 respectively.

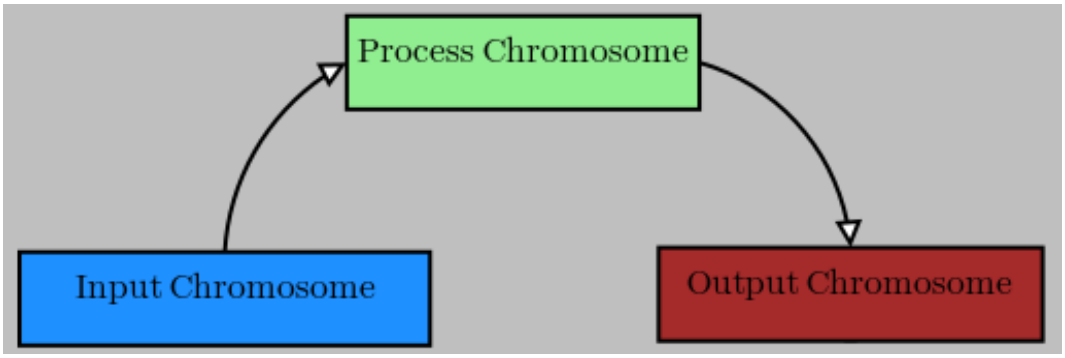

Figure 3.4: Metadata Extraction

For the initial version of the software we have focused on the metadata extraction, completing a section of the target as can be seen in figure 3.4. This has been achieved primarily as a test bed for the deployment within the virtual machines, as we experiment with the limits of what is possible within this environment. Our ultimate goal will rely heavily on parallel processing so we may look towards a cloud based solution, however we now have a clearer understanding of what may be achievable in the current set up.

3.6.3 Machine Learning and Metadata Setup

The initial version of the software relies on a python package named *pandas-profiling*. This package generate a profile report on the data that gives the following results:

- **Essentials:** type, unique values, missing values
- **Quantile statistics** like minimum value, Q1, median, Q3, maximum, range, interquartile range
- **Descriptive statistics** like mean, mode, standard deviation, sum, median absolute deviation, coefficient of variation, kurtosis, skewness
- **Most frequent values**
- **Histogram**
- **Correlations** highlighting of highly correlated variables, Spearman, Pearson and Kendall matrices
- **Missing values matrix**, count, heatmap and dendrogram of missing values

As can be seen in the code found in Appendix J .10, the data is brought in from our MySQL store. This is data that has already been given some structure by previous processes. It is then ran through the pandas-profiling package and the results are then collated. Currently this is outputted as an HTML file.

Development in the future will be required in order to change the way in which the output is generated, with the goal being to add these as attributes to the source data and storing the results as a data frame in the structured layer of the Hadoop data lake.

Chapter 4

Data Lake

4.1 General Data Lake Description

A data lake is a centralised repository that allow the storage of any structured and unstructured data in a single structure. A data lake also scales to a larger scale model with ease. The use of the data lake can range from simple storage, to a base from which to run analytics or big data processing, and machine learning at-scale.

The Serums data lake will combine all of these potential uses, with the structure designed to clearly separate each task.

4.2 Data Lake Zones

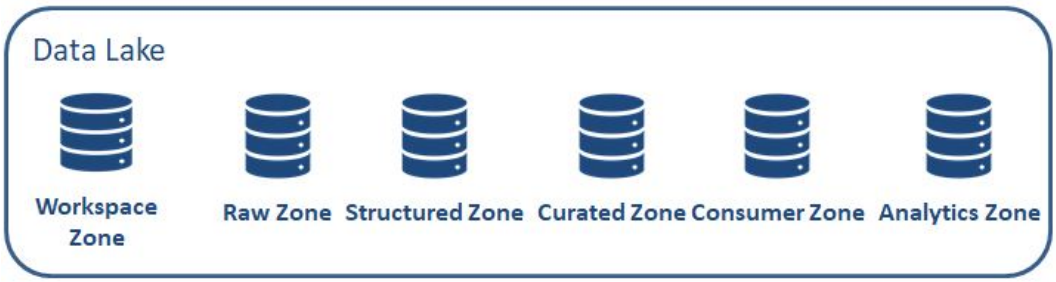

Figure 4.1: Data Lake Zones

- Workspace Zone
  - The Workspace Zone is used for any internal processing of the data
  - It can be used for prototyping and development of new methods and functions
  - It is temporary

  - It is never exposed to the outside world

- Raw Zone
  - The Raw Zone is the entry point for data into the system
  - No processing is ever done here
  - It is only an input

- Structured Zone
  - The Structured Zone is where the raw data is processed and given structure
  - The data is profiled and metadata generated
  - The metadata is added to the data and will stay with it
  - It is never exposed to the outside world

- Curated Zone
  - The Curated Zone is where the data vault is generated
  - It uses the metadata to drive the structure
  - It is never exposed to the outside world

- Consumer Zone
  - The Consumer Zone contains a stable version of the data vault
  - It is exposed to the API gateway
  - It is from where the data is sent on request

- Analytics Zone
  - The Analytics Zone is where any mass external machine learning takes place
  - Under discussion is that it may be readily available for research
    * This would be based on consent from the patient as well as the application of data masking, such as that being developed by IBM

Chapter 5

Rapid Information Factory

5.1 General Rapid Information Factory Description

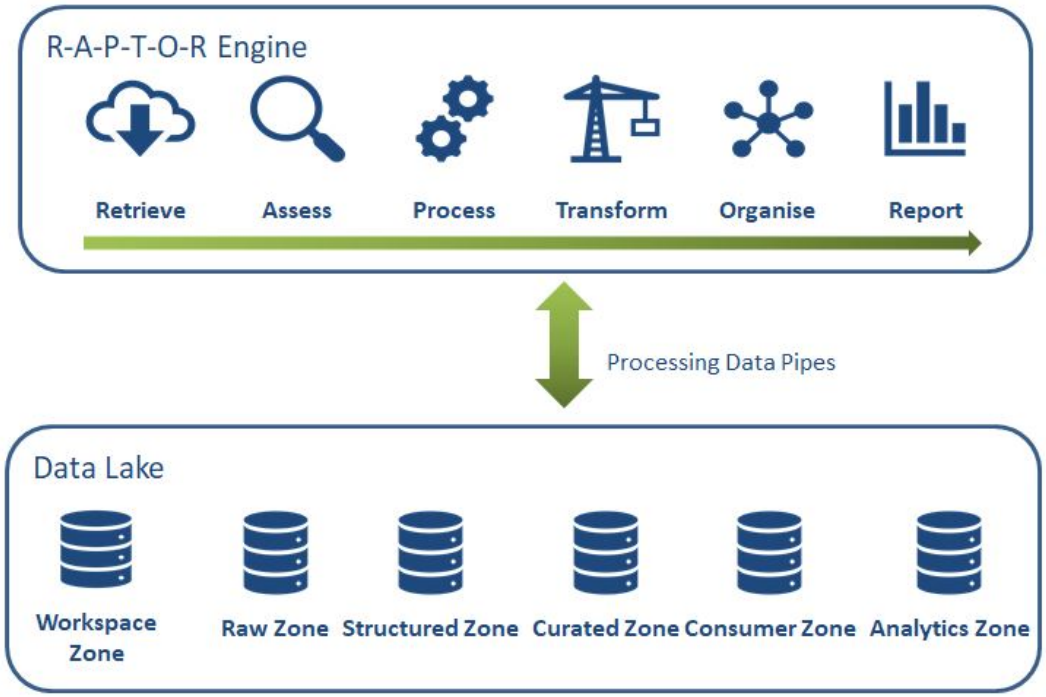

Figure 5.1: Raptor and the Data Lake

The Rapid Information Factory (RIF) is a framework for processing data at-scale. Specifically for Serums, the RIF will drive the management of the data lake 4. The processing of the data within the data lake is handled by the RAPTOR engine.

5.2 What is R-A-P-T-O-R?

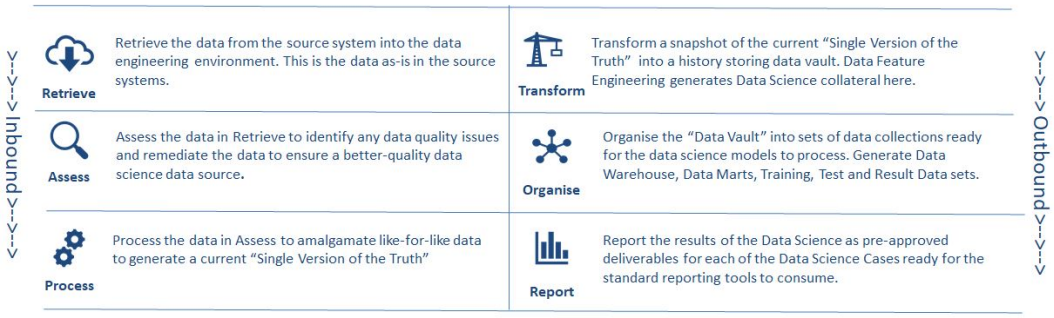

Figure 5.2: Raptor Description

The Retrieve-Assess-Process-Transform-Organise-Report (R-A-P-T-O-R) is a standardised methodology to process data though a pipeline spread over six steps. Figure 5.2 shows a generic description of the system.

The process enhances the processing to make the data sharing more effective and efficient. The basic solution is a hub-an-spoke design to reduce the complexity of interaction between data sources. This R-A-P-T-O-R engine creates reusable data flows using the following data pipelines:

- Retrieve
  - Collects the data from the raw zone in the data lake to transfer it into the base format for the processing
- Assess
  - Assess the data formats and data quality for the data lake to approve the quality to comply to the General Data Protection Regulation (GDPR) - Right to rectification the data lake
- Process
  - Consolidate the data into a universal healthcare data vault using the Time-Person-Object-Location-Event (T-P-O-L-E) hubs
- Transform
  - Extract data from data vault to data warehouse for the patient data container as approved via smart contract and management via a blockchain system to approve what healthcare tags has consent approval
- Organise

  - Stores the extracted encrypted healthcare container in encrypted format within the data lake with using data attributes via healthcare tags

- Report
  - The report step enables the system to transfer the encrypted healthcare container to the approved data receiver

6. Conclusion

The data sources between the different healthcare providers are a diverse set of formats. The current process assists the conversion of the raw data as extracted from the healthcare provider into an universal format. The blockchain enables an immutable data sharing contract between the citizen and their healthcare providers. The multi-phase encryption and access technologies ensures the data security of the overall system.

The next stage of the data lake and data vault will ensure effective stability and enhancements of the healthcare process currently in use between healthcare providers.

Appendices

.1 Appendix A - 0100-SERUMS-RIF-DL-Hadoop-Setup

```
# 0100-SERUMS-RIF-DL-Hadoop-Setup
# Getting the arguments from the command line

import sys
hospital_name = sys.argv[1]

# Setting up RIF routes

rif_results = []
base = '/{}/000-RIF'.format(hospital_name)


#100-Functional-Layer

group = '100-Functional-Layer'
end = ['100-Retrieve', '200-Assess', '300-Process', '
   (cont.)400-Transform', '500-Organize', '600-Report'
   (cont.)]

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)end[x] + '/'
    rif_results.append(concatenated_route)


# 200-Operational-Management-Layer

group = '200-Operational-Management-Layer'
sub = '100-Crawler-Definitions'
end = ['100-Crawler-Master-Service-Definitions',
       '200-Crawler-Workcell-Service-Definitions',
       '300-Crawler-Input-Definitions',
       '400-Crawler-Output-Definitions']

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub + '/' + end[x] + '/'
    rif_results.append(concatenated_route)

sub = '200-Crawler-Management'
end = ['100-Retrieve-Population',
```

```
        '200-Assess-Population',
        '300-Process-Population',
        '400-Transform-Population',
        '500-Organise-Population',
        '600-Report-Population']

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub + '/' + end[x] + '/'
    rif_results.append(concatenated_route)

sub = ['300-Parameters',
        '400-Scheduling',
        '500-Monitoring',
        '600-Communication',
        '700-Alerting',
        '800-Codes-Management']

for x in range(0, len(sub)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub[x] + '/'
    rif_results.append(concatenated_route)


# 300-Audit-Balance-Control-Layer

group = '300-Audit-Balance-Control-Layer'
sub = ['100-Audit',
        '200-Balance',
        '300-Control']

for x in range(0, len(sub)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub[x] + '/'
    rif_results.append(concatenated_route)


# 400-Utility-Layer

group = '400-Utility-Layer'
sub = ['100-Maintenance-Utilities',
        '200-Data-Utilities',
        '300-Processing␣Utilities']
```

```
for x in range(0, len(sub)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub[x] + '/'
    rif_results.append(concatenated_route)


# 500-Business-Layer

group = '500-Business-Layer'
sub = ['100-Functional-Requirements',
       '200-Non-functional-Requirements',
       '300-Data-Profiles',
       '400-Sun-Models']

# Setting up Data Lake routes

dl_results = []
base = '/{}/100-DL'.format(hospital_name)


# 000-Workspace-Zone

group = '000-Workspace-Zone'

concatenated_route = base + '/' + group + '/'
dl_results.append(concatenated_route)


# 100-Raw-Zone

group = '100-Raw-Zone'
sub = '100-External'
end = ['100-University-of-St-Andrews',
       '200-Zuyderland-Medisch-Centrum',
       '300-Fundacio␣Clinic',
       '900-Other']

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub + '/' + end[x] + '/'
    dl_results.append(concatenated_route)

sub = '200-Internal'
end = ['100-CSV',
```

```
        '200-TEXT',
        '300-JSON',
        '400-XML',
        '900-Human-in-the-Loop']

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub + '/' + end[x] + '/'
    dl_results.append(concatenated_route)

sub = '300-Archive'
end = ['100-CSV',
        '200-TEXT',
        '300-JSON',
        '400-XML',
        '900-Human-in-the-Loop']

for x in range(0, len(end)):
    concatenated_route = base + '/' + group + '/' +
       (cont.)sub + '/' + end[x] + '/'
    dl_results.append(concatenated_route)

# 200-Structured-Zone

group = '200-Structured-Zone'

concatenated_route = base + '/' + group + '/'
dl_results.append(concatenated_route)


# 300-Curated-Zone

group = '300-Curated-Zone'
dv = ['Hub','Satellite','Link']
tpole = ['Time','Person','Object','Location','Event']

for dvpath in dv:
    if dvpath == 'Link':
        for hubpath1 in tpole:
            for hubpath2 in tpole:
                if hubpath1 != hubpath2:
                    hubpath = hubpath1 + '-' +
                       (cont.)hubpath2
                    concatenated_route = base + '/' +
```

```
                        (cont.)group + '/' + dvpath + '
                        (cont.)/' + hubpath + '/'
                    dl_results.append(
                        (cont.)concatenated_route)
    else:
        for hubpath in tpole:
            concatenated_route = base + '/' + group +
               (cont.)'/' + dvpath + '/' + hubpath + '
               (cont.)/'
            dl_results.append(concatenated_route)


# 400-Consumer-Zone

group = '400-Consumer-Zone'

concatenated_route = base + '/' + group + '/'
dl_results.append(concatenated_route)


# 500-Analytics-Zone

group = '500-Analytics-Zone'

concatenated_route = base + '/' + group + '/'
dl_results.append(concatenated_route)


# Listing the results

print(rif_results)
print(dl_results)


# Generating CSV for the end points

# import pandas as pd
# df = pd.DataFrame(rif_results, columns=['End Points
   (cont.)'])
# df.to_csv('RIF End Points.csv')

# df = pd.DataFrame(dl_results, columns=['End Points
   (cont.)'])
# df.to_csv('DL End Points.csv')
```

```
# Making the DL

import os

for route in rif_results:
    os.system("hadoop␣fs␣-mkdir␣-p␣{}".format(route))

for route in dl_results:
    os.system("hadoop␣fs␣-mkdir␣-p␣{}".format(route))

os.system("hadoop␣fs␣-ls␣/")
```

.2 Appendix B - Adding Files to Raw

```
import argparse
import os

parser = argparse.ArgumentParser()
parser.add_argument("-l", "--location", help="Select␣
   (cont.)folder␣name")
parser.add_argument("-f", "--file", help="Select␣file␣
   (cont.)name")
parser.add_argument("-d", "--destination", help="
   (cont.)Select␣destination")

args = parser.parse_args()

if(args.file and args.destination):
    if(args.location):
        what_to_put = "{}/{}".format(args.location,
           (cont.)args.file)
    else:
        what_to_put = "{}".format(args.file)
    where_to_put_it = args.destination
    os.system("hadoop␣fs␣-put␣{}␣{}".format(
       (cont.)what_to_put, where_to_put_it))
    os.system("hadoop␣fs␣-ls␣{}".format(
       (cont.)where_to_put_it))
```

.3 Appendix C - Raw to MySQL

```
import pandas as pd
import numpy as np
import mysql.connector as mysql
import getpass
import argparse
import sys
import subprocess
import re
import os
from sqlalchemy import create_engine

parser = argparse.ArgumentParser()
parser.add_argument("-u", "--user", help="Set␣user")
parser.add_argument("-p", "--password", help="Set␣
   (cont.)password", action="store_true")
parser.add_argument("-l", "--location", help="Select␣
   (cont.)hadoop␣folder␣name")
parser.add_argument("-d", "--database", help="Select␣
   (cont.)MySQL␣database␣to␣pass␣files")

args = parser.parse_args()

if(args.password):
    try:
        pwd = getpass.getpass()
    except Exception as error:
        print('ERROR', error)
    else:
        print('Password␣entered')

engine = create_engine("mysql+pymysql://{}:{}
   (cont.)@localhost/{}".format(args.user, pwd, args.
   (cont.)database))
connection = engine.connect()

cmd = "hadoop␣fs␣-ls␣{}".format(args.location)
files = str(subprocess.check_output(cmd, shell=True)).
   (cont.)strip().split('\n')
pattern = r'(?<=\/).+?(?=\\)'
```

```
tables_to_make = []

file_names = re.findall(pattern, files[0])
for file in file_names:
    cmd = "hadoop fs -cat /{}".format(file)
    response = str(subprocess.check_output(cmd, shell=
       (cont.)True))
    tables_to_make.append(response)

index = 0
for table in tables_to_make:
    table = str(table)
    column_pattern = r'(?<=\').+?(?=\)'
    columns = re.findall(column_pattern, table)
    columns = columns[0].split(',')
    columns = [column.strip() for column in columns]
    df = pd.DataFrame(columns=columns)
    row_pattern = r'(?=[A-Z]).+?(?=\n)'

    table = table.replace('"', '')
    table = '"' + table + '"'

    rows = re.findall(row_pattern, table)
    rows.pop(0)
    for row in rows:
        values = row.split(',')
        df_to_append = pd.DataFrame([values], columns=
           (cont.)columns)
        df = df.append(df_to_append, ignore_index=True
           (cont.))

    name = os.path.splitext(os.path.basename(
       (cont.)file_names[index]))[0]
    index+=1
    df.to_sql(name, con=connection)
    print("CREATED {}".format(name.upper()))
```

.4 Appendix D - CSV to MySQL

```
import pandas as pd
import numpy as np
import getpass
```

```
import argparse
import sys
import subprocess
import re
import os
from sqlalchemy import create_engine
from dateutil.parser import parse

parser = argparse.ArgumentParser()
parser.add_argument("-u", "--user", help="Set␣user")
parser.add_argument("-p", "--password", help="Set␣
   (cont.)password", action="store_true")
parser.add_argument("-l", "--location", help="Select␣
   (cont.)hadoop␣folder␣name")
parser.add_argument("-d", "--database", help="Select␣
   (cont.)MySQL␣database␣to␣pass␣files")

args = parser.parse_args()

if(args.password):
    try:
        pwd = getpass.getpass()
    except Exception as error:
        print('ERROR', error)
    else:
        print('Password␣entered')

engine = create_engine("mysql+pymysql://{}:{}
   (cont.)@localhost/{}".format(args.user, pwd, args.
   (cont.)database))
connection = engine.connect()

cmd = "hadoop␣fs␣-ls␣{}".format(args.location)
files = str(subprocess.check_output(cmd, shell=True)).
   (cont.)strip().split('\n')
pattern = r'(?<=\/).+?(?=\\)'

tables_to_make = []

file_names = re.findall(pattern, files[0])
for file in file_names:
    cmd = "hadoop␣fs␣-cat␣/{}".format(file)
    response = str(subprocess.check_output(cmd, shell=
       (cont.)True))
```

```
    tables_to_make.append(response)

def validate_fields(string, fuzzy=False):
    type = ""
    try:
        int(string)
        type = "int"
        return type
    except:
        pass
    try:
        float(string)
        type = "float"
        return type
    except:
        pass
    try:
        parse(string, fuzzy=fuzzy)
        type = "date"
        return type
    except:
        pass
    return "string"

index = 0
for table in tables_to_make:
    table = str(table)
    table = table.replace('b', '', 1)
    table = table.replace("'", "")
    column_pattern = r'(?=[a-zA-Z]).+?(?=\\n)'
    columns = re.findall(column_pattern, table)
    columns = columns[0].split(',')
    columns = [column.strip() for column in columns]
    df = pd.DataFrame(columns=columns)

    row_pattern = r'(?<=\\n).+?(?=\\n)'

    table = table.replace('"', '')
    table = '"' + table + '"'

    rows = re.findall(row_pattern, table)
    for row in rows:
        values = row.split(',')
        df_to_append = pd.DataFrame([values], columns=
```

```
            (cont.)columns)
        df = df.append(df_to_append, ignore_index=True
            (cont.))

    name = os.path.splitext(os.path.basename(
        (cont.)file_names[index]))[0]
    index+=1

    for keys, values in df.iteritems():
        type = ""
        value_array = []
        for value in values:
            type = validate_fields(value)
            value_array.append(type)
            same_value = value_array.count(value_array
                (cont.)[0]) == len(value_array)
        if(same_value):
            if(type == 'date'):
                df[keys] = pd.to_datetime(values)
            if(type == 'int'):
                df[keys] = pd.to_numeric(values)
            if(type == 'float'):
                df[keys] = df[keys].astype(float)
    print('\n')
    df.to_sql(name, con=connection)
    print("CREATED␣{}".format(name.upper()))
```

.5 Appendix E - Converting to Data Vault

```
import pandas as pd
from sqlalchemy import create_engine
import getpass
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--user", "-u", help="Input␣MySQL␣
    (cont.)username")
parser.add_argument("--password", "-p", help="Input␣
    (cont.)MySQL␣password", action="store_true")
parser.add_argument("--primary_key", "-k", help="Input
    (cont.)␣primary␣key␣of␣source␣table")
parser.add_argument("--source_table", "-s", help="
```

```
   (cont.)Input␣source␣table␣name")

args = parser.parse_args()

if(args.password):
    try:
        pwd = getpass.getpass()
    except Exception as error:
        print('ERROR', error)
    else:
        print('Password␣entered')

user = args.user
source_table = args.source_table
source_key = args.primary_key
engine = create_engine("mysql+pymysql://{}:{}
   (cont.)@localhost/{}".format(user, pwd,
   (cont.)source_table))

df = pd.read_sql("select␣*␣from␣{}".format(
   (cont.)source_table), con=engine)

column_controller = pd.read_csv('./columns/{}_to_dv.
   (cont.)csv'.format(source_table))

destination_and_key = column_controller[['destination'
   (cont.), 'foreign_key']].drop_duplicates()

ids = pd.read_sql("select␣{}␣from␣{}".format(
   (cont.)source_key, source_table), con=engine)
ids = ids.chi_general_data.unique()
ids = pd.DataFrame(columns=[source_key], data=ids)

hubs = pd.read_csv('./hubs/{}_hubs.csv'.format(
   (cont.)source_table))

primary_hub = hubs[hubs.index == 0]

secondary_hubs = hubs[hubs.index != 0]

def setting_up_the_hubs_and_links(primary_hub,
   (cont.)secondary_hubs, ids):
    hub = primary_hub['hub'][0]
    hub_key = primary_hub['primary_key'][0]
```

```
    for id_keys, id_values in ids.iterrows():
        main_id = id_values[0]
        main_query = "insert␣into␣{}␣({})␣values␣({});
           (cont.)".format(hub, hub_key, main_id)
        engine.execute(main_query)
        for hub_keys, hub_values in secondary_hubs.
           (cont.)iterrows():
            secondary_hub = hub_values[0]
            secondary_key = hub_values[1]
            query = "insert␣into␣{}␣({})␣values␣({});"
               (cont.).format(secondary_hub,
               (cont.)secondary_key, main_id)
            engine.execute(query)
        for hub_keys, hub_values in secondary_hubs.
           (cont.)iterrows():
            secondary_hub = hub_values[0]
            secondary_key = hub_values[1]
            many_to_many_query = "insert␣into␣many_{}
               (cont.)_has_many_{}␣({}_{},␣{}_{},␣{})␣
               (cont.)values␣({},␣{},␣0);".format(hub,
               (cont.) secondary_hub, hub_key, hub,
               (cont.)secondary_key, secondary_hub,
               (cont.)secondary_key, main_id, main_id)
            engine.execute(many_to_many_query)

def control_search(df, destination_name):
    columns = df.loc[df['destination'] ==
       (cont.)destination_name]
    return columns['name']

def df_search(df, id_column, id_to_check, columns,
   (cont.)column_array, value_array):
    result = df.loc[df[id_column] == id_to_check]
    values = result[columns].values
    value_array.append(values)
    value_array = value_array[0].tolist()
    for entry in value_array:
        column_array.append(columns)

def adding_keys_to_arrays(column_array, value_array,
   (cont.)primary_key_name, foreign_key_name,
   (cont.)foreign_key_value):
    column_array.insert(0, primary_key_name)
```

```
    column_array.append(foreign_key_name)

    value_array.insert(0, 0)
    value_array.append(foreign_key_value)

def building_query(column_array, value_array,
   (cont.)destination_table):
    column_array = tuple(column_array)
    value_array = tuple(value_array)

    column_query = "insert␣into␣{}␣{}␣values".format(
       (cont.)destination_table, column_array)
    column_query = column_query.replace("'", "")

    full_query  = column_query + '{}'.format(
       (cont.)value_array)
    return full_query

setting_up_the_hubs_and_links(primary_hub,
   (cont.)secondary_hubs, ids)

for keys, values in destination_and_key.iterrows():
    columns = control_search(column_controller, values
       (cont.)[0])
    for index, id in ids.iterrows():
        column_array = []
        value_array = []
        df_search(df, source_key, id[0], columns,
           (cont.)column_array, value_array)

        for i in range(0, len(column_array)):
            query_column_array = list(column_array[i])
            query_value_array = list(value_array[0][i
               (cont.)])
            adding_keys_to_arrays(query_column_array,
               (cont.)query_value_array, 'id', '{}'.
               (cont.)format(values[1]), id[0])
            query = building_query(query_column_array,
               (cont.) query_value_array, '{}'.format(
               (cont.)values[0]))
            engine.execute(query)
```

.6 Appendix F

.6.1 serums-blockchain-master

Hyperledger Fabric meets Kubernetes Fabric Meets K8S

What is this? This repository contains a couple of Helm charts to:

- Configure and launch the whole HL Fabric network, either:
  - A simple one, one peer per organisation and Solo orderer
  - Or scaled up one, multiple peers per organization and Kafka or Raft orderer
- Populate the network:
  - Create the channels, join peers to channels, update channels for Anchor peers
  - Install/Instantiate all chaincodes, or some of them, or upgrade them to newer version
- Backup and restore the state of whole network

Who made this?

This work is a result of collaborative effort between APG and Accenture NL.

We had implemented these Helm charts for our project's needs, and as the results looks very promising, decided to share the source code with the HL Fabric community. Hopefully it will fill a large gap! Special thanks to APG for allowing opening the source code.

We strongly encourage the HL Fabric community to take ownership of this repository, extend it for further use cases, use it as a test bed and adapt it to the Fabric provided samples to get rid of endless Docker Compose files and Bash scripts.

License

This work is licensed under the same license with HL Fabric;

Requirements

- A running Kubernetes cluster, Minikube should also work, but not tested
- HL Fabric binaries
- Helm, developed with 2.11, newer 2.xx versions should also work
- jq 1.5+ and yq 2.6+
- Argo, both CLI and Controller
- Minio, only required for backup/restore flows
- Run all the commands in *fabric-kube* folder

Network Architecture

- **Simple Network Architecture:** Simple Network
- **Scaled Up Kafka Network Architecture:** Scaled Up Network
- **Scaled Up Raft Network Architecture:** Scaled Up Raft Network

Note: Due to TLS, transparent load balancing is not possible with Raft orderer as of Fabric 1.4.1.

Launching The Network First install chart dependencies, you need to do this only once:

```
$ helm repo add kafka http://storage.googleapis.
   (cont.)com/kubernetes-charts-incubator
$ helm dependency update ./hlf-kube/
```

Then create necessary stuff:

```
$ ./init.sh ./samples/simple/ ./samples/chaincode/
```

This script:

- Creates the *Genesis block* using *genesisProfile* defined innetwork.yaml file in the project folder
- Creates crypto material using *cryptogen* based on crypto-config.yaml file in the project folder
- Creates channel artifacts by iterating over *channels* in network.yaml using *configtxgen*
- Compresses chaincodes into tar archives by iterating over *chaincodes* in network.yaml *Copies everything created into main chart folder:* hlf-kube

Now, we are ready to launch the network:

```
$ helm install ./hlf-kube --name hlf-kube -f
   (cont.)samples/simple/network.yaml -f samples/
   (cont.)simple/crypto-config.yaml
```

This chart creates all the above mentioned secrets, pods, services, etc. cross configures them and launches the network in unpopulated state.

Wait for all pods are up and running:

```
$ kubectl get pod --watch
```

In a few seconds, pods will come up: Screenshot_pods Congrulations you have a running HL Fabric network in Kubernetes!

Creating channels

Next lets create channels, join peers to channels and update channels for Anchor peers:

```
$ helm template channel-flow/ -f samples/simple/
  (cont.)network.yaml -f samples/simple/crypto-
  (cont.)config.yaml | argo submit - --watch
```

Wait for the flow to complete, finally you will see something like this: Screenshot_channel_flow **Installing chaincodes**

Next lets install/instantiate/invoke chaincodes

```
$ helm template chaincode-flow/ -f samples/simple/
  (cont.)network.yaml -f samples/simple/crypto-
  (cont.)config.yaml | argo submit - --watch
```

Wait for the flow to complete, finally you will see something like this: Screenshot_chaincode_flow

Install steps may fail even many times, never mind about it, it's a known Fabric bug, the flow will retry it and eventually succeed.

Lets assume you had updated chaincodes and want to upgrade them in the Fabric network. Firt update chaincode 'tar' archives:

```
$ ./prepare_chaincodes.sh ./samples/simple/ ./
  (cont.)samples/chaincode/
```

Then make sure chaincode ConfigMaps are updated with new chaincode tar archives:

```
$ helm upgrade hlf-kube ./hlf-kube -f samples/
  (cont.)simple/network.yaml -f samples/simple/
  (cont.)crypto-config.yaml
```

Or alternatively you can update chaincode ConfigMaps directly:

```
$ helm template -f samples/simple/network.yaml -x
  (cont.)templates/chaincode-configmap.yaml ./hlf
  (cont.)-kube/ | kubectl apply -f -
```

Next invoke chaincode flow again with a bit different settings:

```
$ helm template chaincode-flow/ -f samples/simple/
  (cont.)network.yaml -f samples/simple/crypto-
  (cont.)config.yaml -f chaincode-flow/values.
  (cont.)upgrade.yaml --set chaincode.version=2.0
  (cont.) | argo submit - --watch
```

All chaincodes are upgraded to version 2.0! Screenshot_chaincode_upgade_all
Lets upgrade only the chaincode named *very-simple* to version 3.0:

```
$ helm template chaincode-flow/ -f samples/simple/
  (cont.)network.yaml -f samples/simple/crypto-
  (cont.)config.yaml -f chaincode-flow/values.
  (cont.)upgrade.yaml --set chaincode.version=3.0
  (cont.) --set flow.chaincode.include={very-
  (cont.)simple} | argo submit - --watch
```

Chaincode *very-simple* is upgarded to version 3.0! Screenshot_chaincode_upgade_single

Scaled-up Kafka network Now, lets launch a scaled up network backed by a Kafka cluster.

First tear down everything:

```
$ argo delete --all
$ helm delete hlf-kube --purge
```

Wait a bit until all pods are terminated:

```
$ kubectl get pod --watch
```

Then create necessary stuff:

```
$ ./init.sh ./samples/scaled-kafka/ ./samples/
  (cont.)chaincode/
```

Lets launch our scaled up Fabric network:

```
$ helm install ./hlf-kube --name hlf-kube -f
  (cont.)samples/scaled-kafka/network.yaml -f
  (cont.)samples/scaled-kafka/crypto-config.yaml
  (cont.)-f samples/scaled-kafka/values.yaml
```

Again lets wait for all pods are up and running:

```
$ kubectl get pod --watch
```

This time, in particular wait for 4 Kafka pods and 3 ZooKeeper pods are running and ‘ready‘ count is 1/1. Kafka pods may crash and restart a couple of times,

this is normal as ZooKeeper pods are not ready yet, but eventually they will all come up. Screenshot_pods_kafka

Congratulations you have a running scaled up HL Fabric network in Kubernetes, with 3 Orderer nodes backed by a Kafka cluster and 2 peers per organisation. Your application can use them without even noticing there are 3 Orderer nodes and 2 peers per organisation.

Lets create the channels:

```
$ helm template channel-flow/ -f samples/scaled-
  (cont.)kafka/network.yaml -f samples/scaled-
  (cont.)kafka/crypto-config.yaml | argo submit -
  (cont.) --watch
```

And install chaincodes:

```
$ helm template chaincode-flow/ -f samples/scaled-
  (cont.)kafka/network.yaml -f samples/scaled-
  (cont.)kafka/crypto-config.yaml | argo submit -
  (cont.) --watch
```

Scaled-up Raft network

Now, lets launch a scaled up network based on three Raft orderer nodes spanning two Orderer organisations. This sample also demonstrates how to enable TLS and use actual domain names for peers and orderers instead of internal Kubernetes service names. Enabling TLS globally is mandatory as of Fabric 1.4.1. Hopefully will be resolved soon.

For TLS, we need hostAliases support in Argo workflows and also in Argo CLI, which is implemented but not released yet. You can install Argo controller from Argo repo with the below command. We have built Argo CLI binary from Argo repo for Linux which can be downloaded from here. **Use at your own risk!**

```
$ kubectl apply -n argo -f https://raw.
  (cont.)githubusercontent.com/argoproj/argo/
  (cont.)master/manifests/install.yaml
```

Compare scaled-raft-tls/configtx.yaml with other samples, in particular it uses actual domain names like _peer0.atlantis.com_ instead of internal Kubernetes service names like_hlf-peer–atlantis–peer0_. This is necessary for enabling TLS since otherwise TLS certificates won't match service names.

Also in network.yaml file, there are two additional settings. As we pass this file to all Helm charts, it's convenient to put these settings into this file.

```
$ tlsEnabled: true
$ useActualDomains: true
```

First tear down everything:

```
$ argo delete --all
$ helm delete hlf-kube --purge
```

Wait a bit until all pods are terminated:

```
$ kubectl get pod --watch
```

Then create necessary stuff:

```
$ ./init.sh ./samples/scaled-raft-tls/ ./samples/
  (cont.)chaincode/
```

Lets launch our Raft based Fabric network in _broken_ state:

```
$ helm install ./hlf-kube --name hlf-kube -f
  (cont.)samples/scaled-raft-tls/network.yaml -f
  (cont.)samples/scaled-raft-tls/crypto-config.
  (cont.)yaml
```

The pods will start but they cannot communicate to each other since domain names are unknown.

Run this command to collect the host aliases:

```
$ kubectl get svc -l addToHostAliases=true -o
  (cont.)jsonpath='{"hostAliases:\n"}{range..
  (cont.)items[*]}-␣ip:␣{.spec.clusterIP}{"\n"}␣␣
  (cont.)hostnames:␣[{.metadata.labels.fqdn}]{"\n
  (cont.)"}{end}' > samples/scaled-raft-tls/
  (cont.)hostAliases.yaml
```

Or this one, which is much convenient:

```
$ ./collect_host_aliases.sh ./samples/scaled-raft-
  (cont.)tls/
```

Let's check the created hostAliases.yaml file:

```
$ cat samples/scaled-raft-tls/hostAliases.yaml
```

The output will be something like:

```
hostAliases:
 - ip: 10.0.110.93
   hostnames: [orderer0.groeifabriek.nl]
 - ip: 10.0.32.65
```

```
  hostnames: [orderer1.groeifabriek.nl]
- ip: 10.0.13.191
  hostnames: [orderer0.pivt.nl]
- ip: 10.0.88.5
  hostnames: [peer0.atlantis.com]
- ip: 10.0.88.151
  hostnames: [peer1.atlantis.com]
- ip: 10.0.217.95
  hostnames: [peer10.aptalkarga.tr]
- ip: 10.0.252.19
  hostnames: [peer9.aptalkarga.tr]
- ip: 10.0.64.145
  hostnames: [peer0.nevergreen.nl]
- ip: 10.0.15.9
  hostnames: [peer1.nevergreen.nl]
```

The IPs are internal ClusterIPs of related services. Important point here is, as opposed to pod ClusterIPs, service ClusterIPs are stable, they won't change if service is not deleted and re-created.

Next, let's update the network with this host aliases information. These entries goes into pods' */etc/hosts* file via Pod hostAliases spec.

```
$ helm upgrade hlf-kube ./hlf-kube -f samples/
  (cont.)scaled-raft-tls/network.yaml -f samples/
  (cont.)scaled-raft-tls/crypto-config.yaml -f
  (cont.)samples/scaled-raft-tls/hostAliases.yaml
```

Again lets wait for all pods are up and running:

```
$ kubectl get pod --watch
```

Congrulations you have a running scaled up HL Fabric network in Kubernetes, with 3 Raft orderer nodes spanning 2 Orderer organizations and 2 peers per organization. But unfortunately, due to TLS, your application cannot use them with transparent load balancing, you need to connect to relevant peer and orderer services separately.

Lets create the channels:

```
$ helm template channel-flow/ -f samples/scaled-
  (cont.)raft-tls/network.yaml -f samples/scaled-
  (cont.)raft-tls/crypto-config.yaml -f samples/
  (cont.)scaled-raft-tls/hostAliases.yaml | argo
  (cont.)submit - --watch
```

And install chaincodes:

```
$ helm template chaincode-flow/ -f samples/scaled-
   (cont.)raft-tls/network.yaml -f samples/scaled-
   (cont.)raft-tls/crypto-config.yaml -f samples/
   (cont.)scaled-raft-tls/hostAliases.yaml | argo
   (cont.)submit - --watch
```

Configuration
There are basically 2 configuration files: crypto-config.yaml and network.yaml.

- crypto-config.yaml
  - This is Fabric's native configuration for *cryptogen* tool. We use it to define the network architecture. We honour *OrdererOrgs*, *PeerOrgs*, *Template.Count* in PeerOrgs (peer count) and even *Template.Start*

```
OrdererOrgs:
  - Name: Groeifabriek
    Domain: groeifabriek.nl
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: Karga
    Domain: aptalkarga.tr
    EnableNodeOUs: true
    Template:
      Start: 9  # we also honour Start
      Count: 1
    Users:
      Count: 1
```

- network.yaml
  - This file defines how network is populated regarding channels and chaincodes

```
network:
  # only used by init script to create genesis
     (cont.)block
  genesisProfile: OrdererGenesis
```

```
# defines which organizations will join to which
   (cont.) channels
channels:
  - name: common
    # all peers in these organizations will join
       (cont.) the channel
    orgs: [Karga, Nevergreen, Atlantis]
  - name: private-karga-atlantis
    # all peers in these organizations will join
       (cont.) the channel
    orgs: [Karga, Atlantis]
  # defines which chaincodes will be installed
     (cont.)to which organizations
    chaincodes:
      - name: very-simple
        # chaincode will be installed to all
           (cont.)peers in these organizations
        orgs: [Karga, Nevergreen, Atlantis]
        # at which channels are we instantiating
           (cont.)/upgrading chaincode?
        channels:
        - name: common
          # chaincode will be instantiated/
             (cont.)upgraded using the first
             (cont.)peer in the first
             (cont.)organization
          # chaincode will be invoked on all
             (cont.)peers in these organizations
          orgs: [Karga, Nevergreen, Atlantis]
          policy: OR('KargaMSP.member','
             (cont.)NevergreenMSP.member','
             (cont.)AtlantisMSP.member')

      - name: even-simpler
        orgs: [Karga, Atlantis]
        channels:
        - name: private-karga-atlantis
          orgs: [Karga, Atlantis]
          policy: OR('KargaMSP.member','
             (cont.)AtlantisMSP.member')
```

For chart specific configuration, please refer to the comments in the relevant values.yaml files.

TLS TLS

Using TLS is a two step process. We first launch the network in broken state, then collect ClusterIPs of services and attach them to pods as DNS entries using pod hostAliases spec.

Important point here is, as opposed to pod ClusterIPs, service ClusterIPs are stable, they won't change if service is not deleted and re-created.

Backup Restore Requirements

- Persistence should be enabled in relevant components (Orderer, Peer, CouchDB)
- Configure Argo for some artifact repository. Easiest way is to install Minio
- An Azure Blob Storage account with a container named *hlf-backup* (configurable)

At the moment, backups can only be stored at Azure Blob Storage but it's quite easy to extend backup/restore flows for other mediums, like AWS S3. See bottom of backup-workflow.yaml.

IMPORTANT: Backup flow does not backup contents of Kafka cluster, if you are using Kafka orderer you need to manually back it up. In particular, Kafka Orderer with some state cannot handle a fresh Kafka installation, see this (Jira ticket), hopefully Fabric guys will fix this soon.

Backup Restore Flow

First lets create a persistent network:

```
$ ./init.sh ./samples/simple-persistent/ ./samples
  (cont.)/chaincode/
$ helm install --name hlf-kube -f samples/simple-
  (cont.)persistent/network.yaml -f samples/
  (cont.)simple-persistent/crypto-config.yaml -f
  (cont.)samples/simple-persistent/values.yaml ./
  (cont.)hlf-kube
```

Again lets wait for all pods are up and running, this may take a bit longer due to provisioning of disks.

```
$ kubectl get pod --watch
```

Then populate the network, you know how to do it :)

Backup

Start backup procedure and wait for pods to be terminated and re-launched with *Rsync* containers.

```
$ helm upgrade hlf-kube --set backup.enabled=true
  (cont.)-f samples/simple-persistent/network.
  (cont.)yaml -f samples/simple-persistent/crypto
  (cont.)-config.yaml -f samples/simple-
  (cont.)persistent/values.yaml ./hlf-kube
```

```
$ kubectl get pod --watch
```

Then take backup:

```
$ helm template -f samples/simple-persistent/
  (cont.)crypto-config.yaml --set backup.target.
  (cont.)azureBlobStorage.accountName=<your
  (cont.)account name> --set backup.target.
  (cont.)azureBlobStorage.accessKey=<your access
  (cont.)key> backup-flow/ | argo submit -  --
  (cont.)watch
```

Screenshot_backup_flow

This will create a folder with default 'backup.key' (html formatted date *yyyy-mm-dd*), in Azure Blob Storage and hierarchically store backed up contents there.

Finally go back to normal operation:

```
$ helm upgrade hlf-kube -f samples/simple-
  (cont.)persistent/network.yaml -f samples/
  (cont.)simple-persistent/crypto-config.yaml -f
  (cont.)samples/simple-persistent/values.yaml ./
  (cont.)hlf-kube
$ kubectl get pod --watch
```

Restore

Start restore procedure and wait for pods to be terminated and re-launched with *Rsync* containers.

```
$ helm upgrade hlf-kube --set restore.enabled=true
  (cont.) -f samples/simple-persistent/network.
  (cont.)yaml -f samples/simple-persistent/crypto
  (cont.)-config.yaml -f samples/simple-
  (cont.)persistent/values.yaml ./hlf-kube
$ kubectl get pod --watch
```

Then restore from backup:

```
$ helm template --set backup.key='<backup key>' -f
  (cont.) samples/simple-persistent/crypto-config
  (cont.).yaml --set backup.target.
  (cont.)azureBlobStorage.accountName=<your
  (cont.)account name> --set backup.target.
  (cont.)azureBlobStorage.accessKey=<your access
  (cont.)key> restore-flow/ | argo submit  -  --
  (cont.)watch
```

Screenshot_restore_flow
Finally go back to normal operation:

```
$ helm upgrade hlf-kube -f samples/simple-
   (cont.)persistent/network.yaml -f samples/
   (cont.)simple-persistent/crypto-config.yaml -f
   (cont.)samples/simple-persistent/values.yaml ./
   (cont.)hlf-kube
$ kubectl get pod --watch
```

Limitations

- TLS
  - Transparent load balancing is not possible with TLS as of Fabric 1.4.1. So, instead of *Peer-Org*, *Orderer-Org* or *Orderer-LB* services, you need to connect to individual *Peer* and *Orderer* services
- Multiple Fabric networks in the same Kubernetes cluster
  - This is possible but they should be run in different namespaces. We do not use Helm release name in names of components, so if multiple instances of Fabric network is running in the same namespace, names will conflict.

FAQ and more Please see [FAQ](FAQ.md) page for further details. Also this post at Accenture's open source blog provides some additional information like motivation, how it works, benefits regarding NFR's, etc.

.7 Appendix G

.7.1 serums-frontend-master

Build the docker image, then expose port 3000. You need to set the backend url in /api/getRules.js. For example: http://your-domain/v1/api

.8 Appendix H

.8.1 serums-backend-master

Backend to connect to the blockchain of serums.

To run, build the docker image, and run it. Needs /tmp/crypto to contain all the cryptographic artifacts, and the connection profile set to /usr/src/app/src/config/network.json

This runs the backend on port 3001.

.9 Appendix I

.9.1 serums-ansible-master

1. First you need a kubernetes cluster (the ansible scripts in this folder can server as a baseline, although they may need some updating)

2. You need docker installed

3. If not yet done, generate the blockchain artifacts using **serums_manager.sh init** in serums-blockchain. If you regenerate them compared to defaults, you need to update the **connection-profile.json** in serums-blockchain/hlf-explorer, because the paths (especially for the private keys) will have changed

4. You need a docker registry setup. On bare metal, you can use this, other services such as GCP or AWS will provide one for you

5. Edit the address of the backend public endpoint serums-frontend/api/getRules.js, and change the URLs in ingress-template.yaml to correspond to your domains

6. Run **create_app.sh**. This script takes as first argument the location in which serums-blockchain, serums-backend and serums-frontend folders are. Second argument is the address of your docker registry, which defaults to Trow

.10 Appendix J

```
import matplotlib
matplotlib.use('Agg')
import pandas as pd
import numpy as np
import getpass
import argparse
import sys
import subprocess
import re
import os
from sqlalchemy import create_engine
from dateutil.parser import parse
import pandas_profiling

parser = argparse.ArgumentParser()
parser.add_argument("-u", "--user", help="Set␣user")
parser.add_argument("-p", "--password", help="Set␣
   (cont.)password", action="store_true")
```

```
parser.add_argument("-d", "--database", help="Select␣
   (cont.)MySQL␣database␣to␣connect␣to")
parser.add_argument("-t", "--table", help="Select␣
   (cont.)table␣to␣process")
args = parser.parse_args()

if(args.password):
    try:
        pwd = getpass.getpass()
    except Exception as error:
        print('ERROR', error)
    else:
        print('Password␣entered')

engine = create_engine("mysql+pymysql://{}:{}
   (cont.)@localhost/{}".format(args.user, pwd, args.
   (cont.)database))
connection = engine.connect()

sql = "select␣*␣from␣{}".format(args.table)

df = pd.read_sql(sql, con=connection)

pandas_profiling.ProfileReport(df)

profile = pandas_profiling.ProfileReport(df)
rejected_variables = profile.get_rejected_variables(
   (cont.)threshold=0.9)

profile = pandas_profiling.ProfileReport(df)
profile.to_file(outputfile="/api_output/{}.html".
   (cont.)format(args.table))
```