Project no. 826278

SERUMS

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

Refined Software for Storage, Access, Blockchain and metadata Extraction for Smart Patient Health Records D2.4

Due date of deliverable: 31 October 2020

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	31/10/20	E Blackledge	SOPRA	Version 001.000

Executive Summary

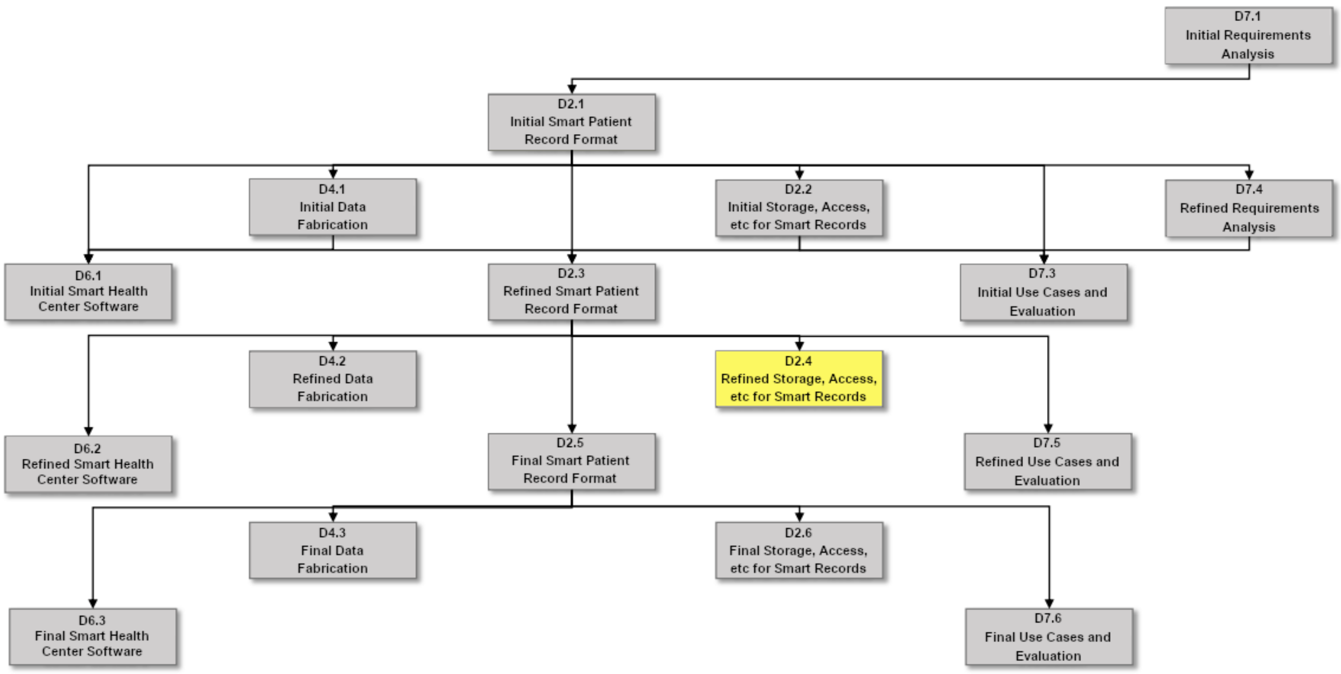

D7.1
Initial Requirements Analysis
D2.1
Initial Smart Patient Record Format
D4.1
Initial Data Fabrication
D2.2
Initial Storage, Access, etc for Smart Records
D7.4
Refined Requirements Analysis
D6.1
Initial Smart Health Center Software
D2.3
Refined Smart Patient Record Format
D7.3
Initial Use Cases and Evaluation
D4.2
Refined Data Fabrication
D2.4
Refined Storage, Access, etc for Smart Records
D6.2
Refined Smart Health Center Software
D2.5
Final Smart Patient Record Format
D7.5
Refined Use Cases and Evaluation
D4.3
Final Data Fabrication
D2.6
Final Storage, Access, etc for Smart Records
D6.3
Final Smart Health Center Software
D7.6
Final Use Cases and Evaluation

Figure 1: Dependencies between D2.4 and other deliverables

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 blockchain. Any interactions with the system is stored for audit on the blockchain.

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 refined versions of the software.

Chapter 2

Glossary of Terms

Throughout the document, various technical terms will be used that require at least some background knowledge to understand why and how they have been applied. This chapter will highlight some of the most important of these. Three of these in particular have similar names and are all related to some degree. These are database, data lake, and data vault. It is important to distinguish between these throughout the report.

2.1 Database

A database is a structure which holds organized information. A database contains tables which contain fields. The tables have usually been defined to serve one purpose and, as such, the fields are usually related in some way.

An example might be that within a hospital's database, they have a patient address table. Within the patient address table would be fields related to a patient's address. It would be very unlikely to find a field that held notes related to a patient's anesthesia during their last operation within this table. This would likely be found in its another table alongside further information about patient surgical procedures.

2.2 Data Lake

A data lake is a type of data storage that typically allows for the storage of multiple data types. Unlike a database, the data stored in a data lake is raw and unorganized. A data lake will typically allow for the storage of raw source files such as images, text documents, or spreadsheets.

Typically a data lake acts as an input to a system since the data requires processing before it is of use. Due to this fact, a data lake is not something that a non-researcher or data scientist would typically probe for data. A receptionist at the hospital who has been asked by a patient to check the date of their next appointment would query a bookings database rather than access the hospital's data lake.

2.3 Data Vault

Data vault is a *methodology for building databases*. A data vault differentiates itself from a standard database through its novel structure which allows for continuous change to its source data. As will be covered in detail in Appendix A, a more traditional way of modelling database would quickly become difficult to manage as we attempt to merge different sources.

To reiterate, however, since a data vault is just another way of modelling a database, it is a database.

2.4 Blockchain

Blockchain is a type of distributed database. Whereas our other data storage solutions (database, data lake, and data vault) represent a single copy of each piece of data that they contain, a blockchain contains multiple copies of the data. In fact, with the type of blockchain that we have chosen, each *node* on the blockchain holds a copy of the entire data set. The *nodes* for the **Serums** blockchain are hosted by the hospitals, with each hospital controlling one. The advantage that this type of data storage gives is that there is no single owner or controller of the data. Since a blockchain seeks consensus on the state of the data from all *nodes* on the network, it is more likely that the data is correct, and the risk of the data the blockchain contains being unavailable due to hardware or technical issues is reduced.

2.5 API

An Application Programming interface (API) is a method for allowing the sharing of an application's functionality through the use of inputs and outputs. These inputs and outputs are clearly defined and allow a user to interact with a piece of software without having to understand the process that is taking place behind the scenes or indeed interacting with the code that makes up the piece of software.

For instance, when checking your bank balance, you simply login and select that feature. In the background an API has run a piece of code which searches for your balance, returns the current value, and displays it.

Chapter 3

Smart Patient Health Format

To add context to this deliverable, this chapter will concern itself with an overview of T2.1 Smart Patient Record Format (SPHR). The reasoning for this is that all of the technologies described in this deliverable are in place to facilitate the creation and transmission of the SPHR. The most up to date deliverable for this task was D2.3 (M16).

3.1 Overview of the Smart Patient Health Record format

T2.1 Smart Patient Record Format asks that we *"define the precise format of a Smart Patient Record that will collate information from various sources and in various structured and unstructured formats"*. For this purpose we have chosen a concept for structuring databases known as data vault. The data vault concept is by Dan Linstedt and can be found detailed in *Building a Scalable Data Warehouse with Data Vault 2.0* [1].

A data vault allows for new sources of data to be added without the need for complex redesigns of the existing data structures [2]. This is of massive benefit for **Serums**, and specifically the SPHR, as it allows more healthcare systems to be added, as well as the development of new use cases. An example of this benefit can be found in Appendix A. Our **Smart Patient Health Record** format can be seen as the **application of access rules to the construction of a data vault**.

3.2 Data Vault

A data vault structure relies on three base type of tables: the Hubs, the Links, and the Satellites. The Hubs contain the business keys, the Links join the Hubs, and the Satellites contain the descriptive data. These three table types can be seen in Figure 3.1.

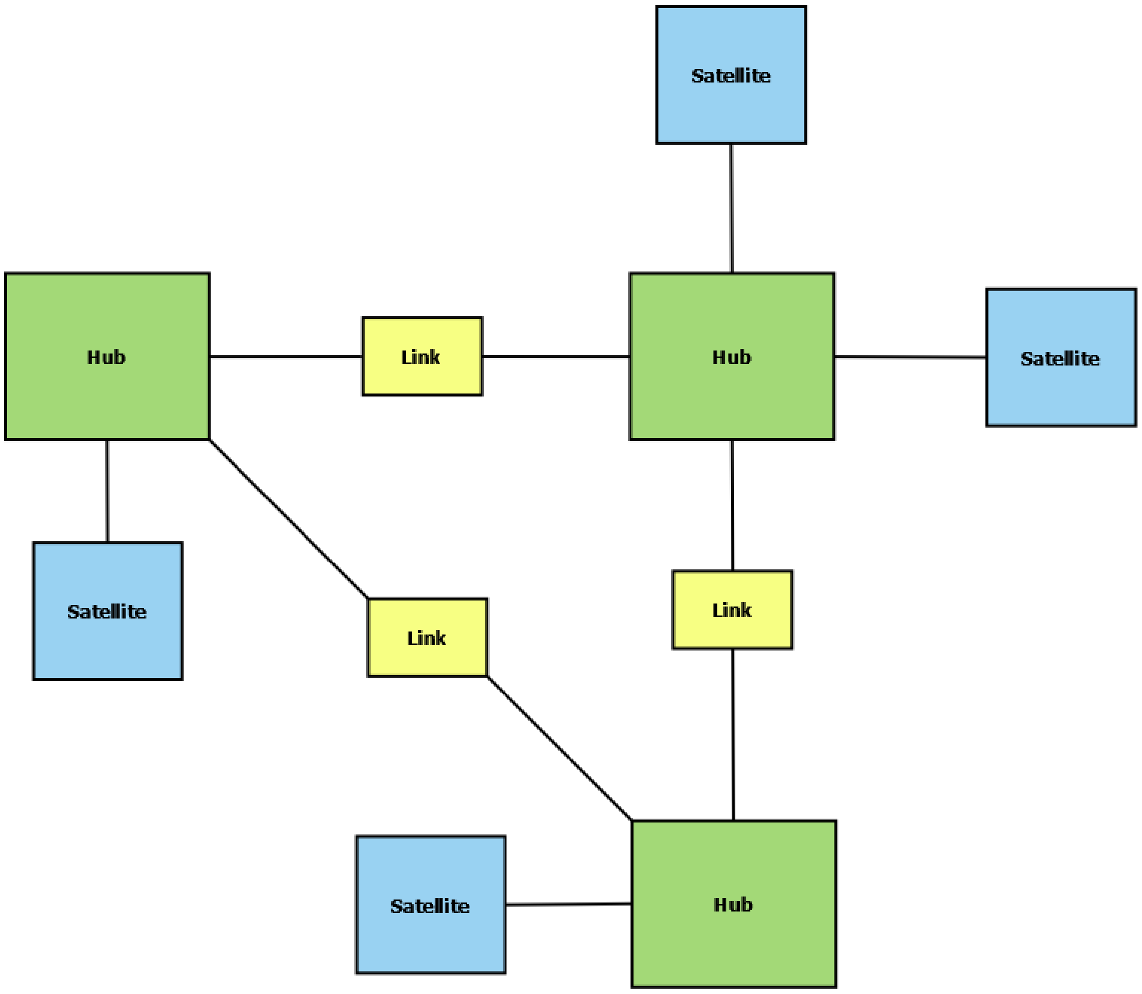
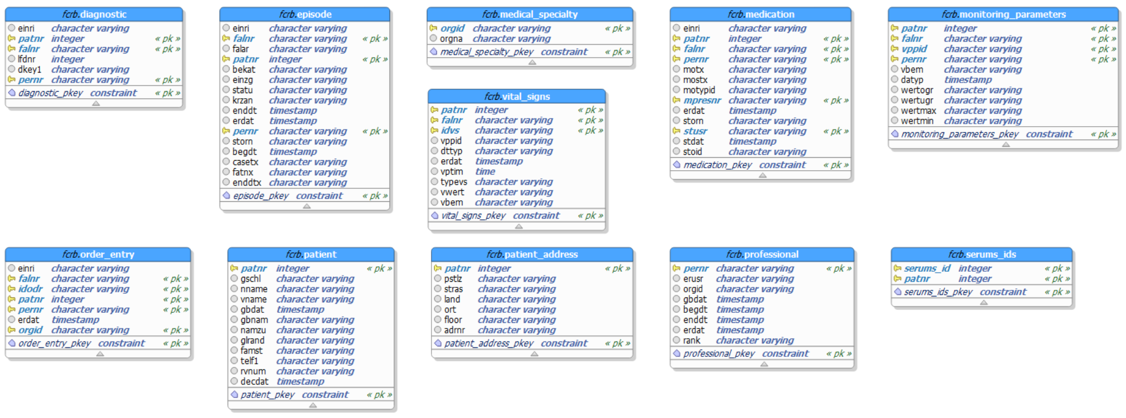


Figure 3.1: Depiction of the relationships between Hubs, Links, and Satellites

Our Hubs have been chosen as Time, Person, Object, Location, and Event (TPOLE). With these five categories, we are able to classify any incoming data (currently a manual process is required to determine which categories each piece of incoming data falls under (Appendix B), however, we are hopeful that some of this process can be automated by the delivery of D2.6 in month 34). Once the incoming data has been mapped to the data vault format we are able to automatically generate the SPHR on request, applying rules to the selection of data in the process (Section 5.2).

3.3 The construction of a Smart Patient Health Record

The following Figures demonstrate how the incoming data from one of our use case partners undergoes transformation from the source structure into the data vault



fcrb.diagnostic

einri	character varying	
patnr	integer	« pk »
falnr	character varying	« pk »
lfdnr	integer	
dkey1	character varying	
pernr	character varying	« pk »
diagnostic_pkey	constraint	« pk »

fcrb.episode

einri	character varying	
falnr	character varying	« pk »
falar	character varying	
patnr	integer	« pk »
bekat	character varying	
einzg	character varying	
statu	character varying	
krzan	character varying	
enddt	timestamp	
erdat	timestamp	
pernr	character varying	« pk »
storn	character varying	
begdt	timestamp	
casetx	character varying	
fatnx	character varying	
enddtx	character varying	
episode_pkey	constraint	« pk »

fcrb.medical_specialty

orgid	character varying	« pk »
orgna	character varying	
medical_specialty_pkey	constraint	« pk »

fcrb.vital_signs

patnr	integer	« pk »
falnr	character varying	« pk »
idvs	character varying	« pk »
vppid	character varying	
dttyp	character varying	
erdat	timestamp	
vptim	time	
typevs	character varying	
vwert	character varying	
vbem	character varying	
vital_signs_pkey	constraint	« pk »

fcrb.medication

einri	character varying	
patnr	integer	« pk »
falnr	character varying	« pk »
pernr	character varying	« pk »
motx	character varying	
mostx	character varying	
motypid	character varying	
mpresnr	character varying	« pk »
erdat	timestamp	
storn	character varying	
stusr	character varying	« pk »
stdat	timestamp	
stoid	character varying	
medication_pkey	constraint	« pk »

fcrb.monitoring_parameters

patnr	integer	« pk »
falnr	character varying	« pk »
vppid	character varying	« pk »
pernr	character varying	« pk »
vbem	character varying	
datyp	timestamp	
wertogr	character varying	
wertugr	character varying	
wertmax	character varying	
wertmin	character varying	
monitoring_parameters_pkey	constraint	« pk »

fcrb.order_entry

einri	character varying	
falnr	character varying	« pk »
idodr	character varying	« pk »
patnr	integer	« pk »
pernr	character varying	« pk »
erdat	timestamp	
orgid	character varying	« pk »
order_entry_pkey	constraint	« pk »

fcrb.patient

patnr	integer	« pk »
gschl	character varying	
nname	character varying	
vname	character varying	
gbdat	timestamp	
gbnam	character varying	
namzu	character varying	
glrand	character varying	
famst	character varying	
telf1	character varying	
rvnum	character varying	
decdat	timestamp	
patient_pkey	constraint	« pk »

fcrb.patient_address

patnr	integer	« pk »
pstlz	character varying	
stras	character varying	
land	character varying	
ort	character varying	
floor	character varying	
adrnr	character varying	
patient_address_pkey	constraint	« pk »

fcrb.professional

pernr	character varying	« pk »
erusr	character varying	
orgid	character varying	
gbdat	timestamp	
begdt	timestamp	
enddt	timestamp	
erdat	timestamp	
rank	character varying	
professional_pkey	constraint	« pk »

fcrb.serums_ids

serums_id	integer	« pk »
patnr	integer	« pk »
serums_ids_pkey	constraint	« pk »

Figure 3.2: The source tables for the FCRB use case

form. It is worth mentioning here that we could select data from multiple sources which was never intended to be joined together. We would simply be storing more business keys in the Hubs and attaching additional Satellites. The rest of the process appears identical.

Starting with the source data for Fundació Clínic per a la Recerca Biomèdica (FCRB) (Figure 3.2) we can see how the data is structured when we make a request to access it. As will be covered in Section 5.1, the actual data that is selected from the source tables is controlled by a set of tags which are designed by the hospitals and selected by the patients. For the following examples we will be using all of the data from these source tables.

In Figure 3.3 we see the start of every one of our data vaults. Here we have generated all of the Hub and Link tables and implemented the relationships between them. Internally, this structure is called the data vault boilerplate.

Figure 3.4 shows the next step of data vault generation. Here the business keys have been added to each of the corresponding Hubs.

The final step of data vault creation is the addition of the Satellites. These are simply attached to their corresponding Hubs. Figure 3.5 shows the complete data vault structure for FCRB's use case. Should a patient wish to share all of the data from FCRB's use case, this is what their SPHR would look like at the time of delivery.

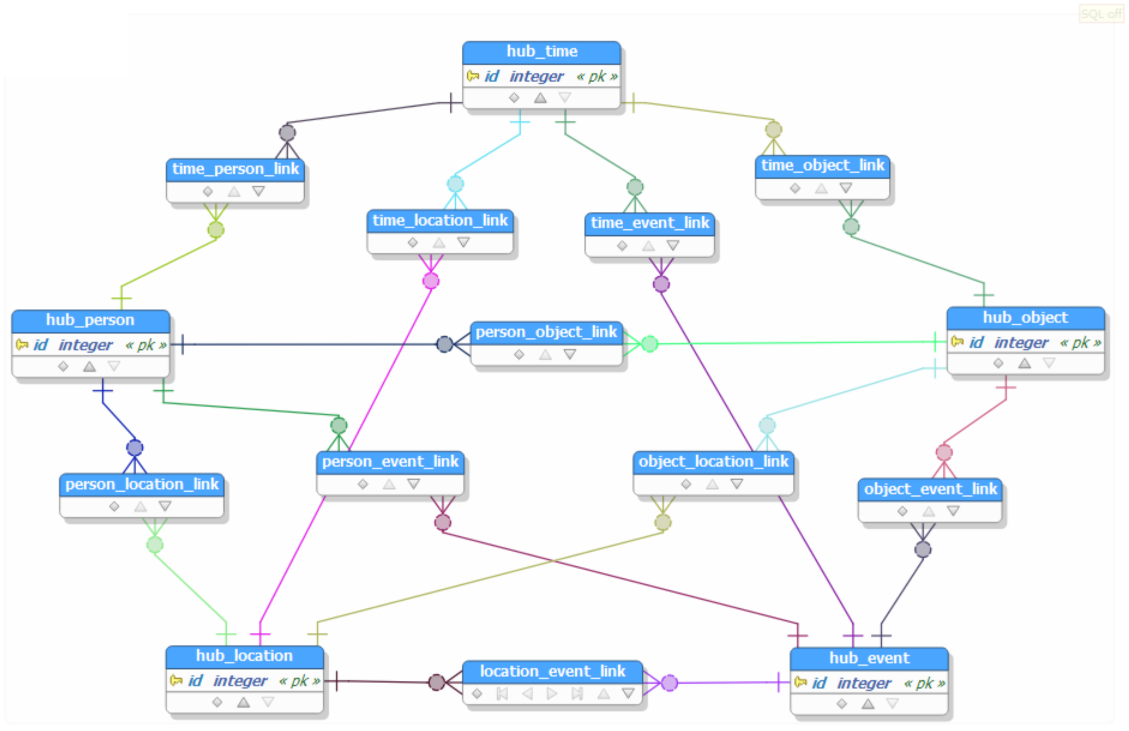


Figure 3.3: The boilerplate structure for our TPOLE data vault

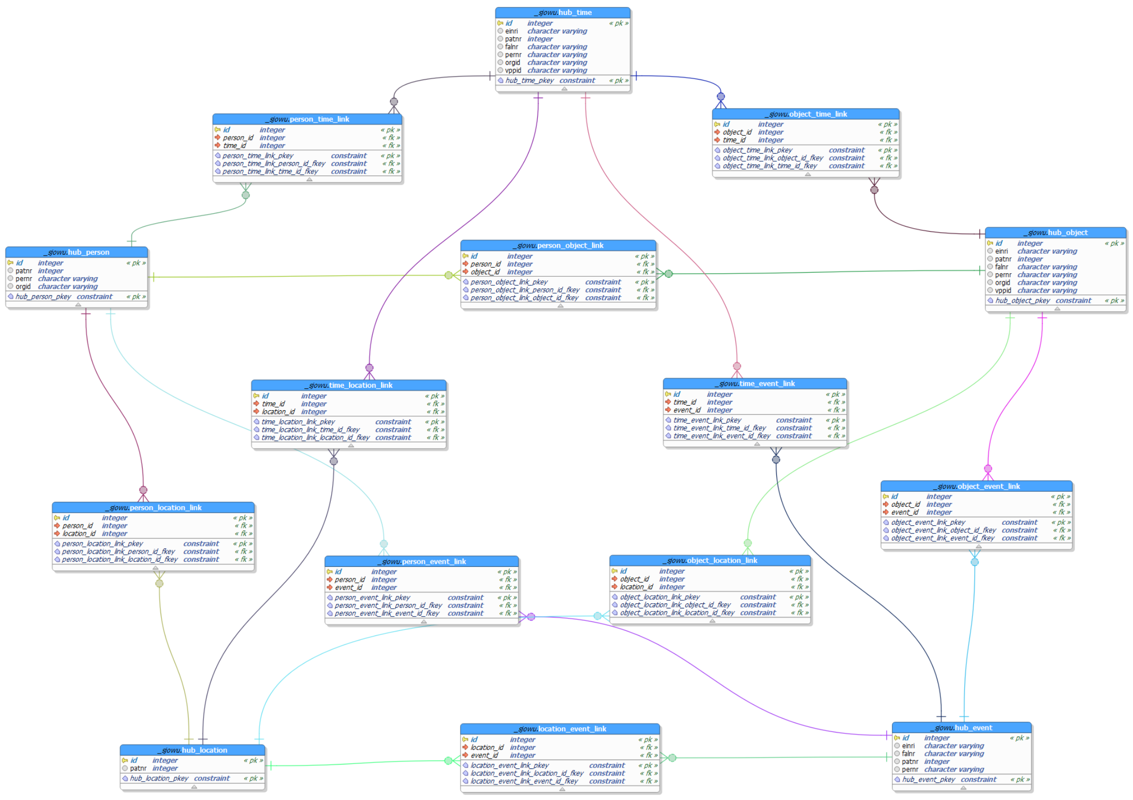


Figure 3.4: The Hubs filled with FCRB's business keys

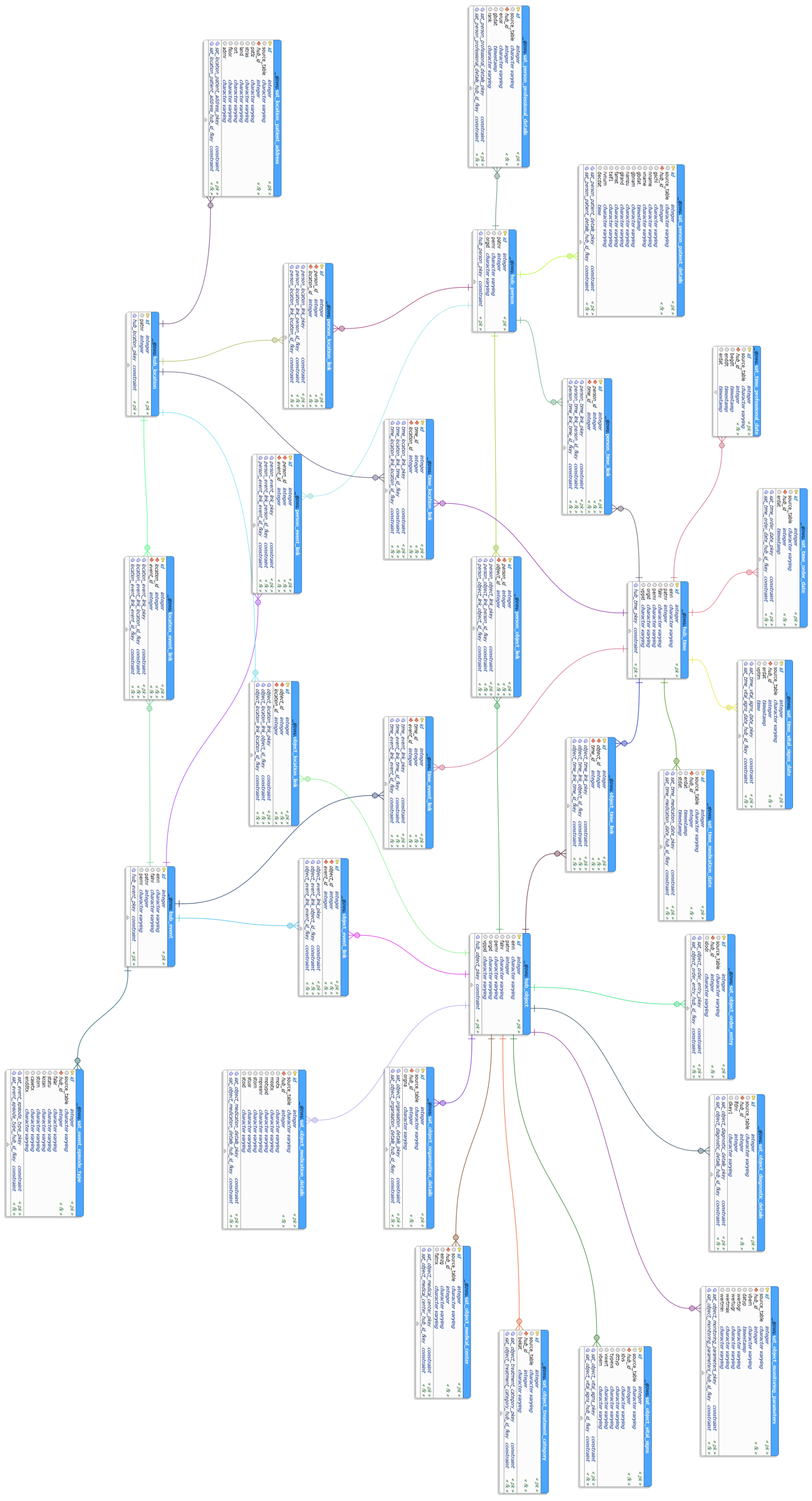


Figure 3.5: The complete data vault structure for FCRB’s data

Chapter 4

Data Lake

In order to store and process the data for **Serums** we have designed a data lake. More specifically, we have designed a framework for a data lake and provided each of the use case partners with their own instance of the data lake. This chapter will explore the structure of the data lake (Section 4.1), the technology used to create and power the data lake (Section 4.2), and how the data lake is implemented both in terms of how it operates at a local level in conjunction with a use case partner and at a global level where elements of it are accessible to the outside world (Section 4.3).

4.1 Structure

Since a data lake can accept data in any form, it is very easy for them to become dumping grounds for their users. A way to imagine this would be to picture a Downloads folder that is shared by an entire company and is never emptied. It is convenient that files downloaded from the internet have a default place to land, however, in time it becomes difficult to quickly locate specific files within this folder.

Given enough time, duplicate files can be downloaded to the folder simply because it was less effort to do this than it would have been to check whether a copy of the desired file already exists. If there are duplicates in place and they are live documents then there is a risk that some people update one copy whilst others update another copy with neither copy being a true representation of the sum of work done.

The solution that **Serums** proposes is to prescribe an precise structure to the data lake that is identical for all the use case partners. For example the address *100-DL/100-Raw-Zone/200-Internal/100-CSV/* exists in each of the data lakes and would be the location that any CSVs which were intended to be shared with the **Serums** platform would be placed. With all data lakes following this common format, it allows for a programmatic approach to how data is accessed across all users without the need to manually search for the location of data or to write custom code to comb through the data lakes. A complete directory tree for the data lake can be found in Appendix C.

The data lakes are broken up into six zones (Figure 4.1), each serving specific uses. For instance, when a new file is added to the data lake, it will be added into the raw zone. As different forms of processing are applied to the file (adding structure, metadata, classification, etc) copies, or copies of specific elements, of the file will be created in the structured zone, followed by the curated zone, before finally being placed in the consumer zone. It is this consumer zone from which the SPHR will be served.

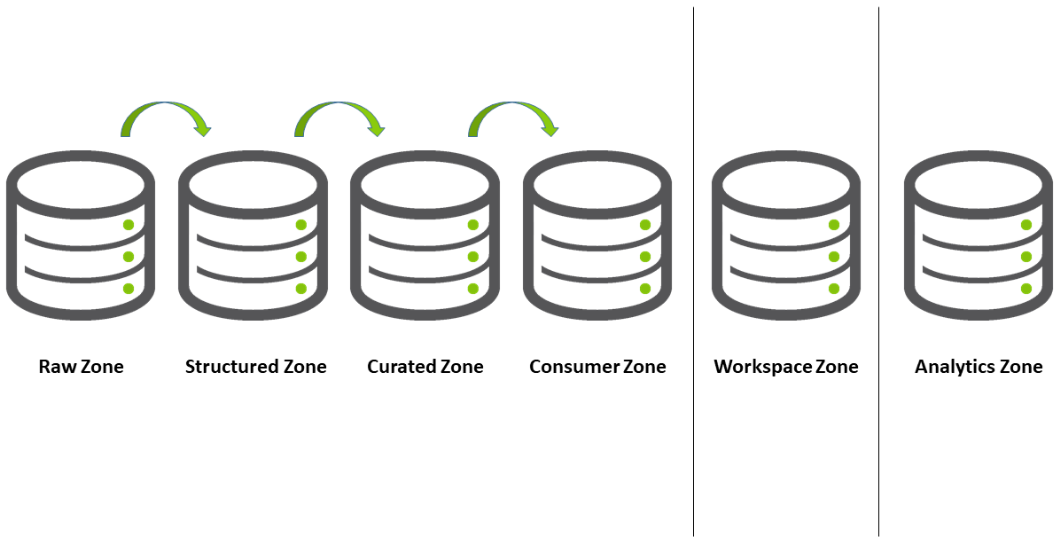


Figure 4.1: The six zones of the data lakes

There are an additional two zones, the analytics zone and the workspace zone. The workspace zone is simply a test bed for new code, which allows the **Serums** developers to work with copies of data, without running the risk of damaging the original versions. The analytics zone will feature copies of the curated zone's data, and will allow any authorised parties to run analytics from, again without the risk of source data being corrupted.

4.2 Technology

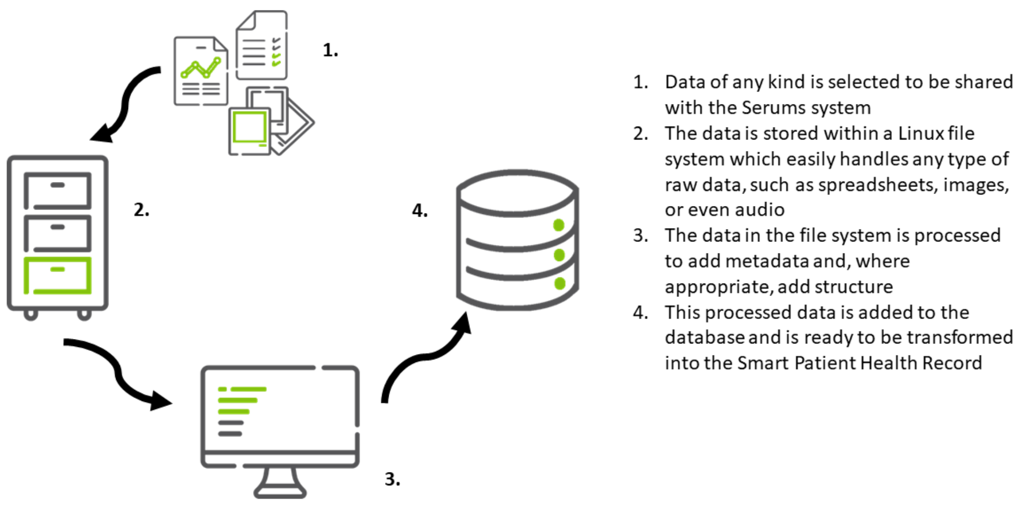

1.
2.
3.
4.
1. Data of any kind is selected to be shared with the Serums system
2. The data is stored within a Linux file system which easily handles any type of raw data, such as spreadsheets, images, or even audio
3. The data in the file system is processed to add metadata and, where appropriate, add structure
4. This processed data is added to the database and is ready to be transformed into the Smart Patient Health Record

Figure 4.2: The underlying technologies behind the Serums data lakes

To provide a platform that is capable of both storing a range of data and running a multitude of processes on the data, we have decided on a combination of a file system within a Linux drive and a PostgreSQL database[1]. The Linux file system allows us to store files of any conceivable format as well as deploy code easily. PostgreSQL offers a full featured relational database that allows us to add structure to our data, as well as store useful data types such as arrays and JSON for use in the rule generation and application (Section 5.2). Both of these technologies are open source which brings the benefits of reducing costs as well as being supported by large developer communities.

[1]For the purposes of **Serums** we have chosen to use a Linux drive for simplicity. In the real world, we would likely move towards a distributed file system such as Hadoop. The underlying principals would remain in regards to how it is structured, however, we would also gain a number of additional benefits. For instance, Hadoop has been specifically designed to work well with very large data sets through how the technology indexes the data it contains.Additionally, hardware failures are less catastrophic on a distributed file system. If one of our Linux file systems becomes corrupted then it loses everything, whereas, by sharing data across multiple servers, a distributed file system would only lose a portion of its data.

4.3 Implementation

As mentioned in Section 4.1, the data lakes are structured into zones (Figure 4.1). Ignoring the workplace and analytics zone, as data is processed it moves[2] between the core zones from raw, to structured, to curated, and finally into the consumer zone. As data is selected to move into the next zone, the data is processed and some form of transformation is applied to it before it lands into its new zone. What follows is the basic operations which take place at each transfer.

4.3.1 Raw Zone

The only entry point that users are allowed to deposit data into is the raw zone. In the opening months of **Serums** we worked with CSV files as the early output of **IBM's** data fabrication software (WP4). As such, these were stored at the appropriate address within the raw zone for each use case partner. Additionally, **ZMC** had an early set of data that was JSON and that was likewise stored at the appropriate address within their data lake.

To move from the raw zone to the structured zone requires processing. Python scripts have been written to handle both CSVs and JSON and are responsible for adding both structure and some basic metadata. The results of these processes are then stored within the PostgreSQL database for each use case partner. Here in the database the data is ready to be added to a SPHR [3].

4.3.2 Structured Zone

As its name suggests, the structured zone requires that all data within it must have some form of structure. Structured forms of data such as a CSV or the output from a relational database can both be stored directly in the PostgreSQL database layer of the data lake. Unstructured data such as an x-ray image might also be stored as a binary within the database, however, it is more likely that the metadata for the image would be stored in the database while the actual image file remains in the raw zone of the data lake or even remain on the hospital's host system. The metadata allows the image to be searchable by numerous aspects such as patient id, type of scan, location on body of the scan, address of where the file is located, etc. and as such is seen as being structured.

The structured zone is the direct source of data for the Smart Patient Health Record (SPHR). The process of moving from the structured zone to the curated

[2]Throughout this section, words such as *move* are used when talking about the flow of data from one zone to the next. In reality, a copy of the data is created in memory before any transformation takes place and it is the result of this which is actually moved into its new location.

[3]Since the successful deployment of the **IBM** data fabrication tool to the **Serums** hosting site, it is now possible to directly pull data from their output tables for use in the data lake. This has been implemented previously, however, since this deliverable will feed directly into proof of concept demonstrations taking place in month 24, we are using a static data source to fill the use case partners' source databases. This allows the same data to be shown to each participant of a use case's study.

zone is triggered when a request for a SPHR is accepted via the SPHR API (Chapter 6). The key element of the transformation taking place in this step is the selection of data to be moved. The selection filters the data based on the individual patient who is having their data accessed by the API as well as the application of the tags (Section 5.1) that are being applied to their data set. The resulting data set of these filters is then moved to the curated zone.

4.3.3 Curated Zone

Within a **Serums** data lake the curated zone is a standalone database within the PostgreSQL layer. Whenever data is moved into the curated zone, a unique new schema is created within the database. In PostgreSQL, schemas act as shards or folders within a database and allow us to store the results of an API request (the SPHR itself) at an isolated address within the database just before transmission via the consumer zone.

By storing the results with this method, we have built in flexibility to the system to allow for potential changes to the operation of the **Serums** system. For instance, the current plan is to delete these schemas on the successful response of the API. There is a case, however, to keep these schemas and allow for the data they contain to be received for a period via another end point on the API. This would speed up the processing of repeat requests. The final design for this will be in place for D2.6 (M34).

The last transformation that takes place to the data before it is moved to the consumer zone is for the data to be encrypted. More details of the encryption can be found in Section 6.3.

4.3.4 Consumer Zone

The consumer zone is the only place that data can be returned from. Currently the consumer zone exists only in memory as the encrypted SPHR is not currently stored. The encrypted data is formatted into a JSON object that is transmitted as a response to the API request. In the previous delivery for this set of tasks (D2.2) there was a provision to store the encrypted data. The address of this was returned as part of the response. This may be reinstated for the final version of the software but is awaiting decision as well.

4.3.5 Workspace Zone

The workspace zone is somewhere for developers to work within without risking the integrity of data within any of the other layers. As such it is required that data must be able to be pulled into this zone by a developer and that data is never publicly available from this zone. It should be possible to pull data from any of the other zones of the data lake to allow developers to work with data at different stages of its life cycle. Typically, the only people working in the workspace zone would be the in-house developers for a healthcare provider.

4.3.6 Analytics Zone

The analytics zone is in someways similar to the workspace zone in that the data within is never publicly available from this zone. The difference here, however, is that the data which is available for use in the analytics zone would typically be obtained from the curated or consumer zone. That is to say, data which has already been cleansed and is suitable to be used for analysis.

Chapter 5

Access Controls

Serums aims to give a patient the means to share their medical record with a high level of control over who has access to a particular subset of their data. We are designing the system to balance usability with strict adherence to international and national laws as well as the individual protocols each hospital has in place for data protection. The patient benefits from the trust that is created by the complete solution, as they will have visibility about who has access to which part of their data and can make their own decisions about the way they would like their data to be accessed. This chapter will look at the three core components that **WP2** implements to achieve this.

5.1 Tags

One of the earliest ideas for giving patients control of their data was to allow for the tagging of data. The original plan was to allow the patients to do the tagging themselves which would allow for complete control[1]. The main issue with this method was that it would likely be too complicated for the patient to do safely. Here the concept of safety relates to both the patient understanding the data they were selecting and ensuring that the tags they were generating were incapable of being in violation of laws such as GDPR.

A simpler and more secure method has been devised to solve this issue whilst still allowing the patient control. WP2 has designated a single set of 15 tags[2] that

[1]Tagging can be thought of as the grouping of subsets of data into a single set. Early thoughts were that this would allow a patient to have complete control over their data, down to an individual column of a single table. While this would be entirely possible to implement, it was decided that this would not be of benefit to either the patients or the healthcare providers.

[2]The 15 tags we have provided cover all of the data for the three use cases. If the scope of **Serums** were to expand to include data from more hospital departments it is entirely probable that this set would expand. To cover a use case that contained every department of every hospital in the EU would require significant work to ensure that a set of tags was designed that allowed all of a patient's data to be categorised whilst at the same time not becoming an unwieldy list that was of little use to the patient.

have been provided to all of the use case partners. It is then up to the hospitals themselves to decide how their data fits within these tags. Some of the source tables may fit neatly within a single tag. Other source tables may be split across a number of tags. Some data may even be applicable to more than one tag and some tags may apply to more than one table. It is entirely up to the hospital to decide how best to apply these. A complete list of the available tags and the typical data that is classified under each one can be found in Appendix D.

By handing the hospitals control for the association of their data to our tags, we have ensured that they can apply their own governance to their own data. It is therefor impossible for the patient to inadvertently tag data in such a way that might violate this. It also allows the same tag to be interpreted differently by different hospitals so that, should local laws prevent a specific type of data from being shared, a hospital would simply exclude that field from their version of the tag. An example of the difference between how a single tag can be assigned differently can be seen in the Listings for **FCRB's** diagnostic tag (Listing 1) and **ZMC's** diagnostic tag (Listing 2).

In both examples we can see that the tags have been applied to subsets of data from two tables, however, we can see that the source tables have very different naming specifications for their fields. This is due to the difference in how the systems the hospitals use operate and how they store their data. Again, by passing control of the setting of these tags to the hospitals, we ensure that the people in charge of this data who understand its purpose best can appropriately assign it to the tags.

The tags themselves are simple to construct. They contain four elements, three of which are mandatory, with an extra field for edge cases that will be discussed below. The three mandatory fields are as follows:

- Tag: The name of the tag which the object relates to
- Table: The full name of the table in the data lake (Section 4.3)
- Fields: The individual fields of the source table that fall within the tag's description

The remaining element, *key_lookup*, is used only when a source table does not itself contain a patient's id field[3]. We have only one example of this in any of our use cases. The table itself is related to the definition of doctors' specialities. In this case, we must locate another table which contains both the fields for the doctors' ids and the patients' ids. As such we provide a way for the API (Chapter 6) to link these fields appropriately. An example of this can be found in Listing 3.

[3]Since WP2 is concerned with individual patients being able to share their data, we must know the patient's id when requesting data.

Listing 1: An example of the *diagnostic* tag for **FRCB's** data

```
[
    {
        "tag": "diagnostic",
        "table": "fcrb.diagnostic",
        "fields": [
            "einri", "patnr", "falnr", "pernr",
            "lfdnr", "dkey1"
        ],
        "key_lookup": {}
    },
    {
        "tag": "diagnostic",
        "table": "fcrb.episode",
        "fields": [
            "einri", "falnr", "patnr", "pernr", "bekat",
            "falar", "statu", "krzan", "storn", "casetx",
            "enddtx", "einzg", "fatnx"
        ],
        "key_lookup": {}
    }
]
```

Listing 2: An example of the *diagnostic* tag for **ZMC's** data

```
[
    {
        "tag": "diagnostic",
        "table": "zmc.patient_diagnostic",
        "fields": [
            "patnr", "type", "name", "anatomical_location",
            "laterality", "begin_date", "end_date"
        ],
        "key_lookup": {}
    },
    {
       "tag": "diagnostic",
       "table": "zmc.patient_documents",
       "fields": [
            "patnr", "report_title", "department",
            "date", "content"
        ],
        "key_lookup": {}
    }
]
```

Listing 3: An of a table that requires the extra *key_lookup* field to be completed

```
[
    {
        "tag": "healthcare_providers",
        "table": "fcrb.professional",
        "fields": [
            "pernr", "orgid", "erusr", "gbdat",
            "rank", "begdt", "enddt", "erdat"
        ],
        "key_lookup": {
            "table": "fcrb.order_entry",
            "keys": [
                "patnr"
            ],
            "field": "orgid"
        }
    }
]
```

5.2 Rules

The use of tags defines *which* data can be shared, however, rules must also be designed to govern *how* the data will be shared. Whilst the ability to construct tags was removed from the patients, the ability for patients to construct rules is essential for the project [4].

A rule itself is constructed of:

- Whose data is covered by the rule
- Who the rule is applied to i.e. a specific doctor or a group of specialists
- Which hospital(s) is the source of the data the rule applies to
- Whether the rule grants or denies permission for the person/people to access the data
- How long the rule is applicable i.e. a specific end date or simply on-going
- Which tag(s) the rule governs (Section 5.1)

The creation process for a rule has been kept deliberately simple for patients. The user interface is shown in Figure 5.1. Here we can see that the patient is able to

[4]While patients will indeed be able to construct their own rules, the hospital administration will also be capable of setting up rules for their patients. It is likely that when a patient signs up for **Serums**, a base set of rules will be applied to their data in line with the local governance for medical data. This would benefit the patient by ensuring their data is covered by the data protection guidelines set out by their local health board.

select whether to grant or deny access, the institution the rule is applicable to, the set of doctors within the institution the rule is applicable to, the tags that the rule is applicable to, and whether or not there is a time frame for the rule to be applied for. Combined with the patient's own id which is known by the system at login, this screen is able to capture all of the required pieces of information required to create a rule.

The responsibility for the storage of the rule is split between the blockchain (for a complete discussion on the workings on the blockchain, please refer to Section 5.3) and the data lake which contains the patient's data. On submission of the rule creation process, the rule is first created in the blockchain. This stores the type of rule (access or deny), whose data the rule governs, who the rule applies to, and any expiry date for the rule[5]. Once the rule is saved in the blockchain, the rule id for the newly created rule is then passed to the corresponding data lake alongside the tags that the rule applies to. These are saved in a special location in the PostgreSQL layer of the data lake.

A full explanation for how the rules are applied to the data can be found in Chapter 6.

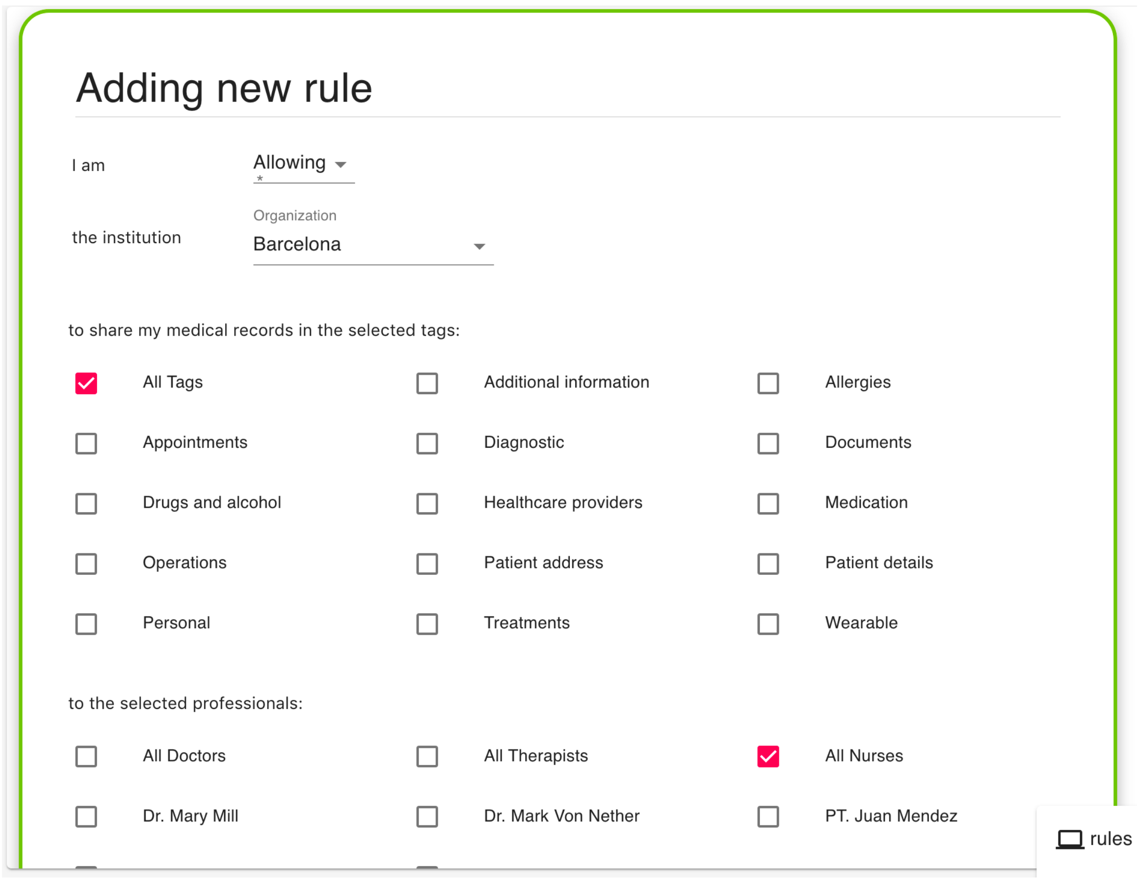
5.3 Blockchain

The main reason why a blockchain is used for the **Serums** application, is because the responsibility for the managing the access rules is shared between all participants of the network. The distributed nature of a blockchain is an inherent mechanism to build trust, as there is no one single entity responsible for the information/rules. If we were to simply use a security matrix on a database, there would still be one party that would manage the database and thus the data itself. Additionally, a blockchain gives a complete history of any rules that have been created out-of-the-box, whereas a database would have to have a custom implemented solution for this.

In theory, there are multiple blockchain types/platforms that could be used. Based on the scope of the work, we have selected Hyperledger Fabric (HLF) which is mature enough for business applications and provides all the necessary functionality that is required by the **Serums** project. This includes its rule based engine, smart contracts, which allows for the embedding of business logic on the blockchain. The logic of performing CRUD operations/validating access rules is written into code and that code is deployed on the blockchain. This is close to a 'traditional' way of programming and thus more accessible. HLF allows for smart contracts to be written in Golang, Node.js (JavaScript) and Java.

The granularity of the blockchain and its implementation of the access rules allows the patient to manage their data. Each member of the consortium (currently

[5]Currently the blockchain also stores the tags the patient selects. It is currently under discussion as to whether the tags should remain stored on the blockchain or instead stored in the data lake. Both are currently implemented.


Adding new rule

I am Allowing

the institution Organization Barcelona

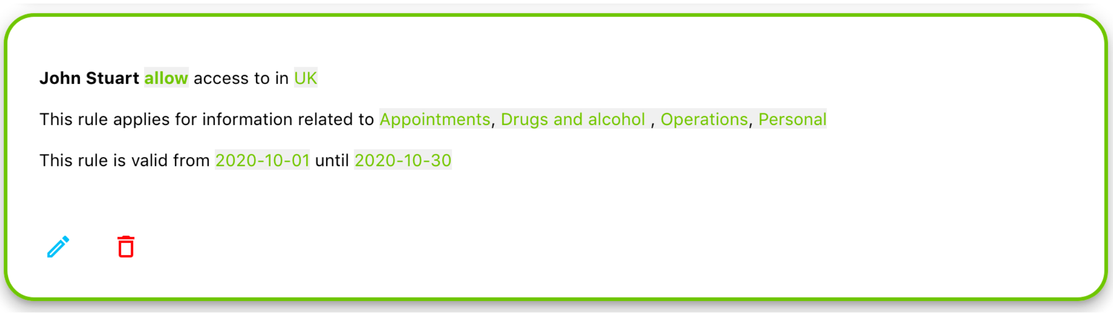
to share my medical records in the selected tags:

✓ All Tags	☐ Additional information	☐ Allergies
☐ Appointments	☐ Diagnostic	☐ Documents
☐ Drugs and alcohol	☐ Healthcare providers	☐ Medication
☐ Operations	☐ Patient address	☐ Patient details
☐ Personal	☐ Treatments	☐ Wearable

to the selected professionals:

☐ All Doctors	☐ All Therapists	✓ All Nurses
☐ Dr. Mary Mill	☐ Dr. Mark Von Nether	☐ PT. Juan Mendez

Figure 5.1: Details from the rule creation page


John Stuart **allow** access to in UK

This rule applies for information related to Appointments, Drugs and alcohol , Operations, Personal

This rule is valid from 2020-10-01 until 2020-10-30

Figure 5.2: The result of rule creation that the patient can view

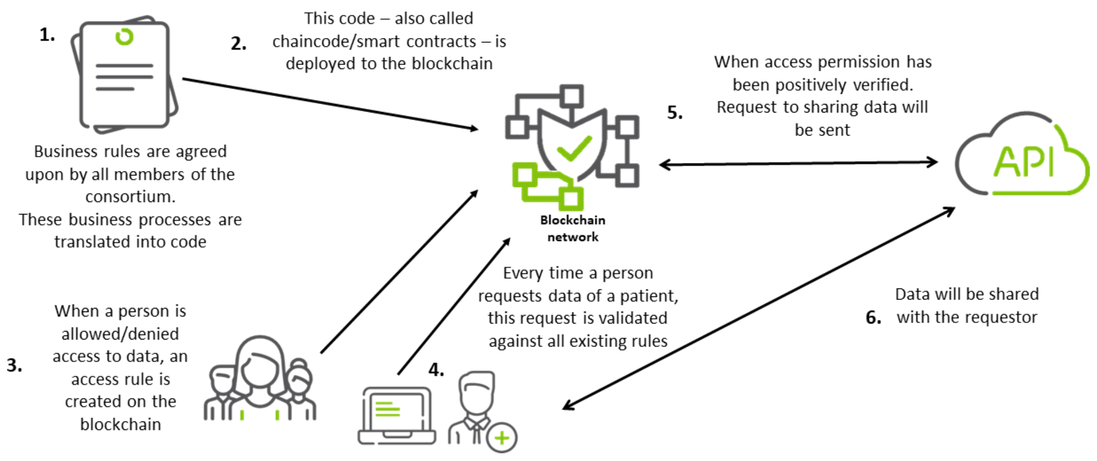

1.
Business rules are agreed upon by all members of the consortium.
These business processes are translated into code
2.
This code – also called chaincode/smart contracts – is deployed to the blockchain
Blockchain network
5.
When access permission has been positively verified. Request to sharing data will be sent
API
3.
When a person is allowed/denied access to data, an access rule is created on the blockchain
4.
Every time a person requests data of a patient, this request is validated against all existing rules
6.
Data will be shared with the requestor

Figure 5.3: The blockchain process flow

USTAN, ZMC and FCRB) hosts a node of the blockchain network – allowing them to have shared management of access to the patient data. Each node holds a complete history of the ledger, thus allows full transparency and audit trail of the events history equally for all partners on the network.

As covered in Section 5.2, aspects of the rules are stored on the blockchain. An example of this can be found in Listing 4. Here we can see that *patient1* has granted *doctor1* permission to see the data found within the tags *wearable* and *personal_details*. This remains true until *16/6/21*. It is worth mentioning that there is a decision pending on whether the storage of tags is perhaps too much personal information so they may be removed for the final version. The other key piece of information is the *id* of the rule.

It is this *id* that, alongside whether permission is granted or not (as is shown by the *action* field), that is returned when an attempt is made to access a patient's data. This is then passed to the data lake which interprets the rule and applies the corresponding tags to the data. Full details are covered in Chapter 6.

When making a request to the blockchain for permission to access data, a user must pass the id of the patient whose data they are asking for, the tags (again, which may be removed from the blockchain's responsibility), and a JSON Web Token (JWT). The patient's id and the tags form the body of the request (Listing 5), while the JWT is included in the header for the request. The JWT itself is another packet of data that has been encrypted. Once decrypted it is very similar to the body of the request in that it is a JSON object containing further information. In this case it contains the id of the person making the request.

Listing 4: A rule as stored on the blockchain

```
{
    "access": [
            {
            "name": "wearable"
        },
        {
            "name": "personal_details"
        }
    ],
    "action": "ALLOW",
    "expires": "2021-06-16T13:20:03.970Z",
    "grantee": {
            "id": "doctor1",
            "orgId": "ZMC",
            "type": "INDIVIDUAL"
    },
        "grantor": {
                "id": "patient1",
                "orgId": "ZMC",
                "type": "INDIVIDUAL"
        },
        "id": "RULE_6d71c87c-726e-43c9-8887-7fee4be2fcd9"
}
```

Combined, the request body and JWT can be processed by the blockchain to check the validity of the request and determine whether the user has permission to see the requested data.

In addition to checking the validity of a request, the blockchain also records the access request to its log. This happens whether permission is granted or not and provides an immutable record of access attempts to see a patient's data. This type of logging allows the system administrators from each healthcare provider on the network to be able to monitor access requests and to keep an eye on any suspicious activity. An example of the logging can be seen in Listing 6.

Listing 5: The body of a request asking for access to a patient's data

```
{
    "patientId": "patient1",
    "patientOrgId": "ZMC",
    "access": [
            {
            "name": "wearable"
        },
        {
            "name": "personal_details"
        }
    ]
}
```

Listing 6: An example of the access log for a successful request

```
{
        "accessGranted": true,
        "id": "ACCESS_REQUEST_26cc863d-01cf-4f27-8d2d-32ff4b6ccee6",
        "requestor": {
                "groupIds": [
                        "DOCTOR"
                ],
                "individualId": "doctor1",
                "orgId": "ZMC"
        },
        "target": {
                "access": [
                        {
                                "name": "OPERATION"
                        }
                ],
                "patientId": "patient1",
                "patientOrgId": "ZMC"
        },
        "timestamp": "2020-06-22T10:12:54.000Z"
}
```

Chapter 6

API

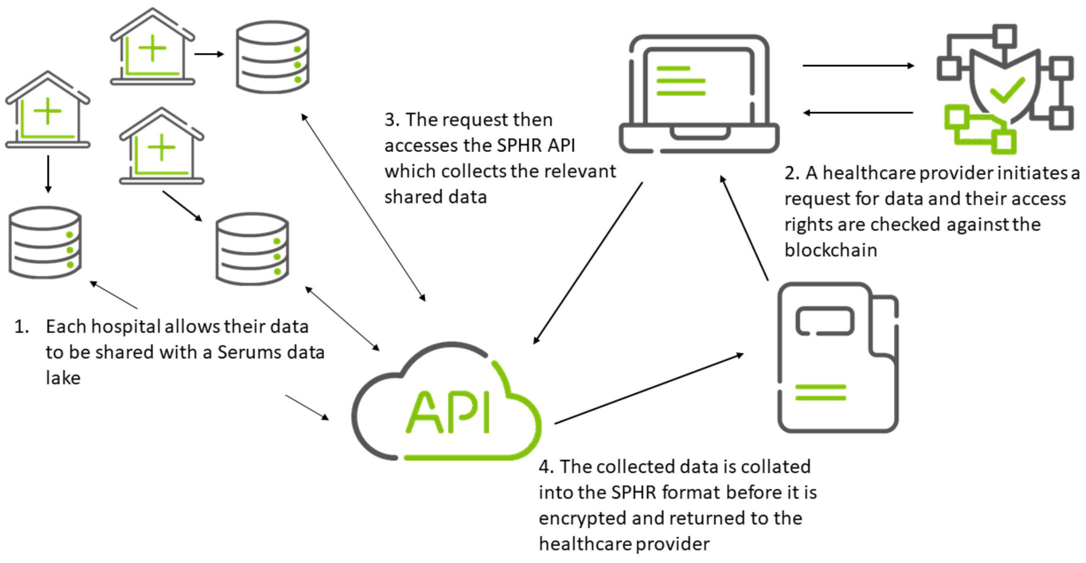

3. The request then accesses the SPHR API which collects the relevant shared data
2. A healthcare provider initiates a request for data and their access rights are checked against the blockchain
1. Each hospital allows their data to be shared with a Serums data lake
API
4. The collected data is collated into the SPHR format before it is encrypted and returned to the healthcare provider

Figure 6.1: An overview of Work Package 2's technology

Figure 6.1 shows a very high level overview of the entire work package. As each of the core components that make up the API have been covered in previous chapters, this chapter will concern itself with how the API integrates them all and then how the data is handled before it is returned to the user.

The **Serums** Smart Patient Health Record (SPHR) API allows the interaction of the data lake (Chapter 4) with the outside world. Before a patient's data becomes available to be used with the **Serums** system, they must first opt into the service. This process involves them creating an account that will in turn generate a *Serums id*. This id is unique to them and will be used by the SPHR API to associate their medical records across any number of healthcare providers that are **Serums** partners[1].

Once a *Serums id* has been generated and associated with their patient id (Section 6.2), they are able to create rules to control their data (Section 5.2). Once at least one rule that allows sharing of data is in place, their data is able to be accessed by the person(s) they have authorised.

It is worth noting that their healthcare data does not automatically become shared with the **Serums** data lake. This must be a manual process that the healthcare provider undertakes by updating their data lake with the newly registered patient's data. This ensures that only **Serums** users have their data available and minimises the risk that a non-service user might find their data being shared in a way they have not authorised.

6.1 Process Flow

Below is a complete list of steps which covers the process that allows for a SPHR to be shared.

1. A healthcare provider signs up to the **Serums** system as an organisation
2. The healthcare provider is added to the blockchain network
3. The healthcare provider registers its staff to the system, this includes both healthcare professionals and system administrators
4. A patient of the healthcare provider signs up to the **Serums** system as a user
5. A *Serums id* is generated for the patient and is stored by the authentication module (WP5)
6. The *Serums id* is associated to the patient's id within the healthcare provider
7. A base set of rules is applied to the patient's records which brings their data inline with the healthcare provider's base data protection requirements
8. The patient is free to create their own rule sets - *optional*
9. A medical professional is able to access the patient's data if a rule exists that grants them permission
10. Upon request, a SPHR is generated in real time that brings together all of the data which the rule applies to. This may combine sets of data from multiple healthcare providers if that is how the patient has designed that rule

[1]For the purpose of this project, the list of partners are **USTAN**, **FCRB**, and **ZMC**.

Steps 1, 2, and 3 must all be in place before the healthcare provider can offer **Serums** as a service to their users. Step 2 helps the network grow which builds in redundancy. Step 3 is required as otherwise the patient would not be able to grant them permission to view their data.

Step 4 is down to individual patients to decide whether or not they want to be part of the **Serums** system. It is important that they are free to take this decision themselves and revoke it at any point. As will be covered in Section 6.2, the ability to leave is very simple.

Steps 5, 6, and 7 should all form part of an automated process which occurs when a patient registers with **Serums**. Currently, work is underway to integrate work packages 2 and 5 (the authentication module). Once this work is complete, it will be possible to automatically register a new user's *Serums id* with their home hospital's patient id. We anticipate that by D2.6 we will have a set of rules in place that are automatically applied according to that hospital's guidelines.

Step 8 has been covered in some detail in Chapter 5 but to reiterate, this is the optional ability that **Serums** offers to patient's that allows them to design access rules in real-time to grant or deny access to their data. Depending on the regulations for the hospital they design a rule for, this can apply to either a group of healthcare workers, or even a single individual.

Steps 9 and 10 are only possible if all of the previous steps have been followed, and a rule exists that applies to the healthcare professional making the request. It is in these steps that the work covered in Chapter 5 is applied to the patient's data at time of request and the SPHR they have the right to access is created and delivered in an encrypted form (Section 6.3).

6.2 End Points

This section will cover all of the available end points of the Smart Patient Health Record (SPHR) API as of this deliverable.

6.2.1 Add User

The purpose of this end point is to associate a patient's *Serums id* with the patient id that a hospital uses for them. Since their *Serums id* is generated by the authentication module and remains a constant, this is a convenient way to link multiple patient ids to a single user. This allows **Serums** to combine health records from multiple sources into a single SPHR.

Listing 7 show's the body of the request to the */add_user* end point. Here we see the *serums_id* 10523 being linked to the *patient_id* 4641202 within **FCRB's** data lake.

Listing 7: The **/add_user** request body

```
{
  "serums_id": 10523,
  "patient_id": 1075835,
  "hospital_id": "ZMC"
}
```

Listing 8: The **/remove_user** request body

```
{
  "serums_id": 10523,
  "hospitals": [
    "FCRB",
    "USTAN",
    "ZMC"
  ]
}
```

6.2.2 Remove User

The purpose of this end point is to remove the association of a patient's *Serums id* from one or more of their patient ids within a hospital(s) data lake. This allows a quick method to immediately prevent the API from sharing any of their data, even before the data itself has been removed from the corresponding data lake(s). With no association between a *Serums id* and their patient id within a hospital, the API will simply return an empty SPHR (assuming that an access rule exists on the blockchain).

The end point is capable of accepting multiple hospitals, which allows a patient to be removed in a single action. This can be seen in Figure 8. It is then up to the hospitals to remove their data from their entry point of their data lake. The exact method to achieve this has not yet been decided, however, it will be in place by the final iteration of the software for WP2 (D2.6 - M34).

6.2.3 Add Rule

The purpose of this end point is to save the relevant information for a rule to the data lake. Full details of the rules can be found in Chapter 5. As can be seen in Listing 9, we are passed the *hospital_id* whose data lake the rule will be saved to, the *rule_id* that has been generated by the blockchain, and the *tags* for the data.

Additionally, we have provided an additional field *filters*. This allows an even greater level of control over the data to be selected. For instance, **FCRB** has specified that they need to filter their data by episode number. This is a unique identifier related to each incident a patient receives treatment for and is very useful for grouping their data by.

Listing 9: The **/add_rule** request body

```
{
  "rule_id": "RULE_6d71c87c-726e-43c9-8887-7fee4be2fcd9",
  "hospital_id": "ZMC",
  "tags": [
    "wearable",
    "patient_details"
  ],
  "filters": ""
}
```

Listing 10: The **/get_sphr** request body

```
{
  "serums_id": 10523,
  "rule_id": "RULE_6d71c87c-726e-43c9-8887-7fee4be2fcd9",
  "hospital_id": "ZMC",
  "public_key": "-----BEGIN PUBLIC KEY-----
    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA24IJ1BITwdCmERIJgk2v
    plEwjsXNupYWE23cy8bSvmVhHuZZKRZU5YPGQwaFZERFT0/fJvtRFinYhe9KOmXw
    HYhrrDCIOgXkYvzJMZ+IWTDMHmrmuXjC/9UfarpzE4mudCIWCVffcGaItP2aMlhu
    E4dxxH9S7xbr4F6mRDmZBbup7H4hnkJAnsy4LMM8vf8eREk4VqW1MQhougdmDCi9
    Xt2ZvOxD4tVEW3wTTWUPm+1ZZk3eyV+twnbp2O3T9EIYQ1Nm7vNkjVgucSDXMKT6
    iCThYCG2a6ogf33Z5mPsSlAT+Q1IfL+8FNkbUH0K/ZJqV8SlI68FPZ/v/rC3TXLs
    4wIDAQAB
    -----END PUBLIC KEY-----"
}
```

6.2.4 Get SPHR

The purpose of this end point is to retrieve a SPHR. As can be seen from Figure 10, the request requires the *serums_id* of the patient, the *rule_id* as was generated by the blockchain, the *hospital_id* of where the patient's data resides[2], and a *public_key*. The *public_key* is used as part of the encryption process to allow for the secure transmission of the SPHR. More details of this process is covered in Section 6.3.

The process of creating a SPHR is covered in previous chapters and an example of the returned data for this query can be found in Appendix E.

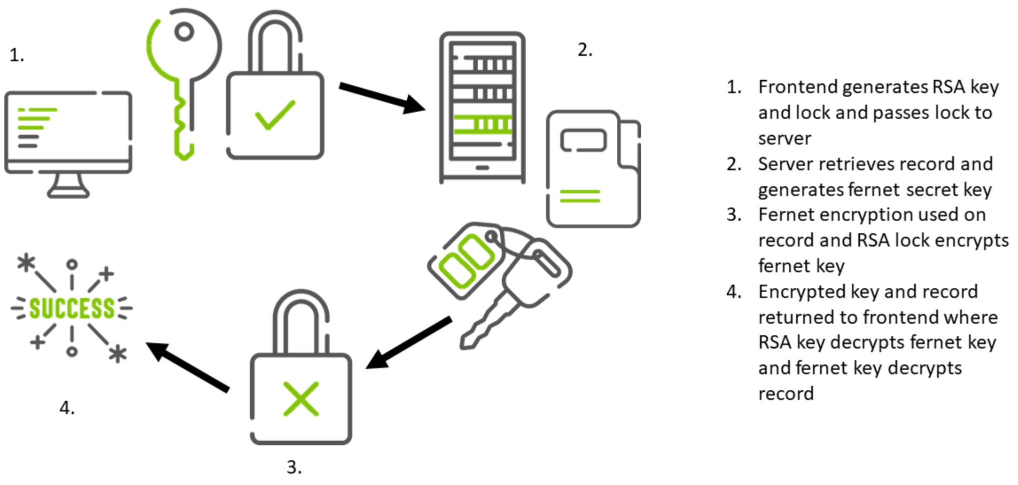

1.
2.
3.
4.
SUCCESS
1. Frontend generates RSA key and lock and passes lock to server
2. Server retrieves record and generates fernet secret key
3. Fernet encryption used on record and RSA lock encrypts fernet key
4. Encrypted key and record returned to frontend where RSA key decrypts fernet key and fernet key decrypts record

Figure 6.2: The encryption process for the Smart Patient Health Record

6.3 Encryption

Due to the fact that **Serums** aims to facilitate the sharing of medical records with healthcare providers outside of a hospital's direct network, it is of vital importance that the data be shared securely. As such, encryption is a key component in allowing this to take place. The process that **Serums** follows involves dual encryption with the frontend client handling one part of the duo and the backend server handling the other. Figure 6.2 is a useful reference for following the process.

The two sides, the client and the server, rely on different forms of encryption. The client uses a form of asymmetric cryptography known as public key encryption. The server uses a form of symmetric cryptography also known as secret key cryptography.

Every time a request is made for a SPHR, the frontend client generates a new RSA public/private key pair[3]. The public key can be shared, even across untrusted networks, and is used to encrypt data which can only be decrypted by the matched private key. As its name suggests, the private key must be kept private and is never shared. In **Serums**, the private key remains in memory for the SPHR request and is

[2]Currently the *get_sphr* end point only takes a single hospital, however, the final version of the API will be able to take a list which will allow medical records from multiple sources to be joined into a single SPHR.

[3]While RSA has been used here, it is possible for any type of public key encryption to be used. For the final version of the software, we may experiment with other options such as Diffie–Hellman which may prove to be more secure in the long run.

Listing 11: An example of a fernet secret key

```
-s6eI5hyNh8liH7Gq0urPC-vzPgNnxauKvRO4g03oYI=
```

never stored. The public key is shared as part of the request data. To see the body of the request in full, refer to Section 6.2.

While this method should allow for the secure encryption and transmission of data, we have decided to reinforce the security by adding in a second type of encryption to actually lock the data. For this, the server uses fernet encryption. This works by generating a secret key. This secret key is used to prime an encryption algorithm which is then used to encrypt the SPHR.

In order to decrypt the SPHR, the client must have an exactly matched secret key (an example of which can be found in Listing 11) with which to prime its own fernet decryption algorithm. As such, the public key from the client is used to encrypt the secret key on the server and now both the encrypted data and the encrypted secret key can be passed back to the client. Here the client decrypts the secret key with its private key before using this newly decrypted secret key to decrypt the SPHR.

As stated above, this process takes place anew every time a request is made with new keys being generated by both the client and the server. This means that if SPHRs were to be intercepted by a bad actor, they would have to spend the computing time to brute force each request separately.

Chapter 7

Planned Changes

This chapter concerns itself with the upcoming changes which are planned before the final software deliverable for WP2 in month 34.

7.1 Final Code Changes

As mentioned in a couple of chapters, there are still decisions pending for a couple of features. One of these is where the responsibility lies for storing the tags (Section 5.1). There are cases for and against storing them on the blockchain vs storing them in the data lakes. This relates to privacy concerns, so feedback from medical ethics specialists will be sought to answer this.

Another decision relates to whether or not to store the encrypted SPHR beyond their transmission. This may come down to an individual decision by the hospitals as to how they plan to use this data and interact with the API. This would see the creation of a new end point for the API that can retrieve an already existing record.

7.2 Default Rules

As was stated in Section 5.2, we expect that the hospitals will design a default set of rules to apply to each patient that signs up to **Serums**. These would be added to the blockchain and data lake at time of registration.

8. 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.

Bibliography

[1] Dan Linstedt and Michael Olschimke. *Building a Scalable Data Warehouse with Data Vault 2.0*. 2015.

[2] Daniel Linstedt and Michael Olschimke. Introduction to Data Warehousing. In *Data Vault 2.0*. 2016.

Appendices

Appendix A

Problems Data Vaults Solve

One of the key issues we are aiming to solve with the Smart Patient Health Record (SPHR) is the combination of sets of data into a single source that can be easily and securely transmitted. For healthcare providers who share the same systems, such as those who use SAP, this would be a relatively painless exercise. However, even within our small set of use case partners, there are key differences between the systems which we are looking to model. The following set of figures is an illustrative example of the benefits of using data vault as the backbone for our SPHR.

For our example, we start with three tables as depicted in Figure A.1. These tables are from a fictitious hospital and are for the patients, the doctors, and the treatments of the patients.

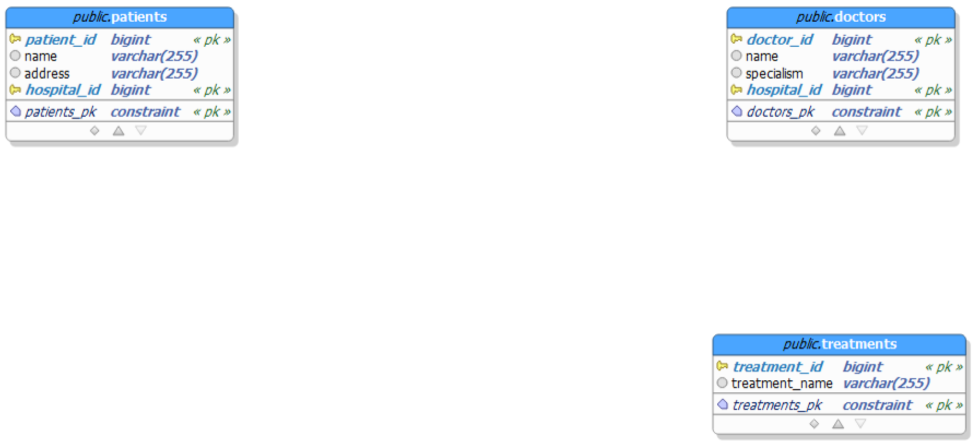
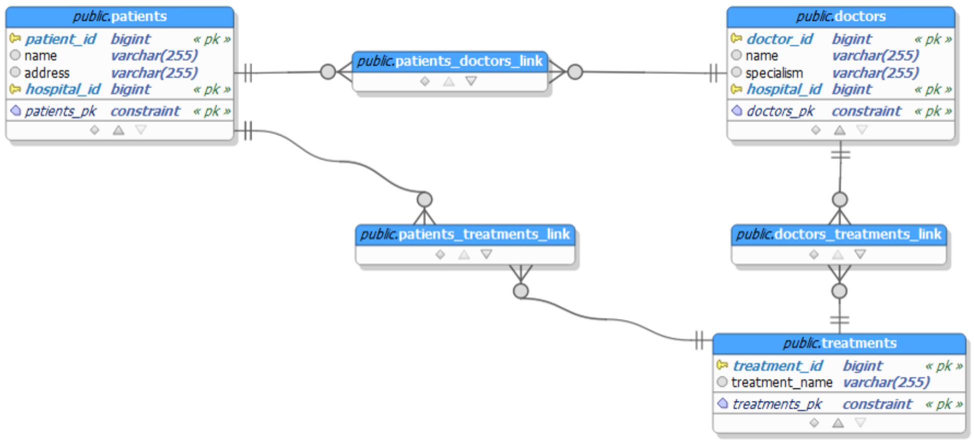

public.patients
patient_id bigint « pk »
name varchar(255)
address varchar(255)
hospital_id bigint « pk »
patients_pk constraint « pk »
public.doctors
doctor_id bigint « pk »
name varchar(255)
specialism varchar(255)
hospital_id bigint « pk »
doctors_pk constraint « pk »
public.treatments
treatment_id bigint « pk »
treatment_name varchar(255)
treatments_pk constraint « pk »

Figure A.1: Three tables which we intend to have relationships between

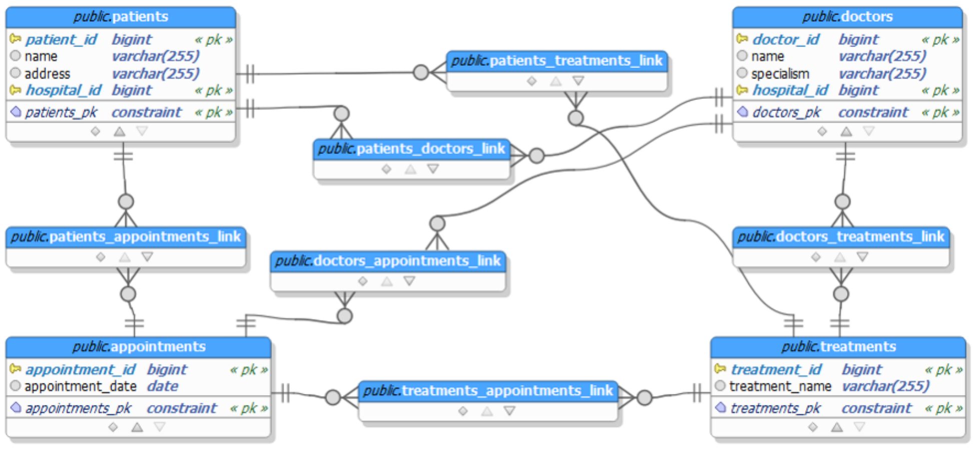
Our example hospital links these tables the using many-to-many relationship model. Specifically a patient can see many different doctors and the doctors can see many different patients. Additionally a doctor many prescribe many different treatments to each of their many different patients and a patient may receive many different treatments as prescribed by their many different doctors. Figure A.2 demonstrates the join tables that can be used to facilitate this type of relationship.



public.patients
patient_id bigint « pk »
name varchar(255)
address varchar(255)
hospital_id bigint « pk »
patients_pk constraint « pk »
public.patients_doctors_link
public.doctors
doctor_id bigint « pk »
name varchar(255)
specialism varchar(255)
hospital_id bigint « pk »
doctors_pk constraint « pk »
public.patients_treatments_link
public.doctors_treatments_link
public.treatments
treatment_id bigint « pk »
treatment_name varchar(255)
treatments_pk constraint « pk »

Figure A.2: The addition of join tables between our three start tables

As our example hospital continues to expand its IT capabilities it adds in the appointments booking system. There is a desire for the appointments system to maintain many-to-many relationships with all the existing tables to ensure flexibility in how the system can be used. This single new data source results in the need for three new join tables to be created and maintained as can be seen in Figure A.3 and is a demonstration of how rapidly scaling issues can be created.



public.patients
patient_id bigint « pk »
name varchar(255)
address varchar(255)
hospital_id bigint « pk »
patients_pk constraint « pk »
public.patients_treatments_link
public.doctors
doctor_id bigint « pk »
name varchar(255)
specialism varchar(255)
hospital_id bigint « pk »
doctors_pk constraint « pk »
public.patients_doctors_link
public.patients_appointments_link
public.doctors_appointments_link
public.doctors_treatments_link
public.appointments
appointment_id bigint « pk »
appointment_date date
appointments_pk constraint « pk »
public.treatments_appointments_link
public.treatments
treatment_id bigint « pk »
treatment_name varchar(255)
treatments_pk constraint « pk »

Figure A.3: A forth table and its relationships added to the existing schema

The example here is extreme however it provides insight into one of the core issues that we would encounter if we were to use standard database modelling techniques. What follows is the same example data set but but this time it is modelled using our data vault technique.

An important step that is not covered here is the mapping of incoming data to its new structure. For each field of each source table we currently must know where it is going to end up. In other words, which Satellite it is going to form part of. This is covered in more depth in Appendix B. Simply put, we categorise each field of each incoming table under one of the Hub categories. Fields which are similar enough in scope are grouped together in Satellites. This does allow for single source tables to be split across multiple Hubs and in turn multiple Satellites.

As covered in Section 3.3 we always start with a boilerplate structure for our data vault, with five Hubs (Time, Person, Object, Location, and Event) with many-to-many relationships formed between each of them in the form of Links. Figure A.4 shows a simplified version of this, reducing the numbers of Links between the Hubs for readability. A complete version of this boilerplate structure can be found in Figure 3.3.

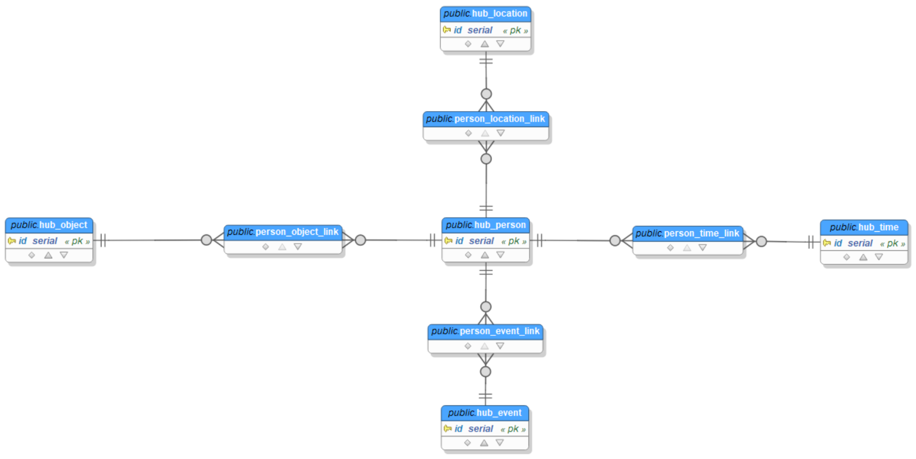


Figure A.4: A simplified data vault boilerplate

With the boilerplate ready, we can extract the business keys from the incoming data and insert them into the appropriate Hub(s) based on the destination Satellite(s) for each source table. Figure A.5 shows the results of this process as applied to our original three tables: patients, doctors, and treatments.

Note here that both the patient and doctor Satellites are attached to the Person Hub and that the treatments Satellite is attached to the Object Hub. The many to many relationships are still maintained in all of these circumstances.

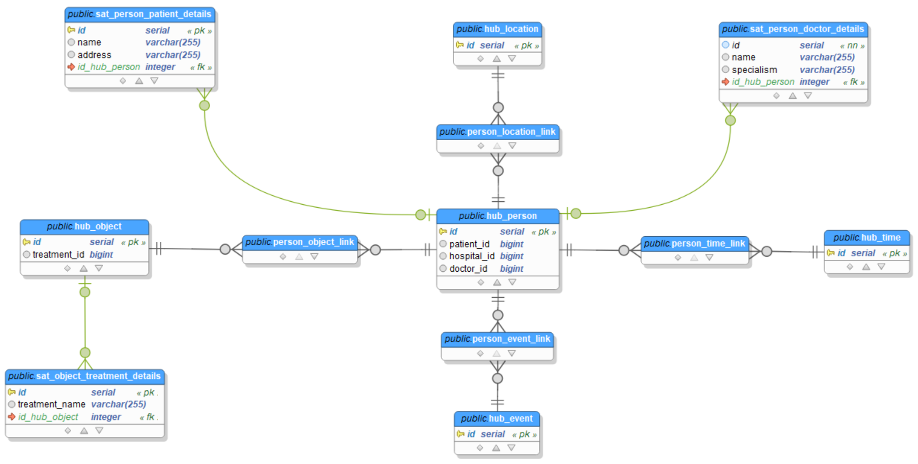


Figure A.5: Our original three tables attached to the data vault skeleton

For the final step of this example we will add in the appointments table. This time it is worth noting that rather than requiring multiple changes to the overall schema of the database, we simply insert the business key (appointment id) into the Time Hub and attach the rest of the appointments table's data as a Satellite to the Time Hub. This can be seen in Figure A.6.

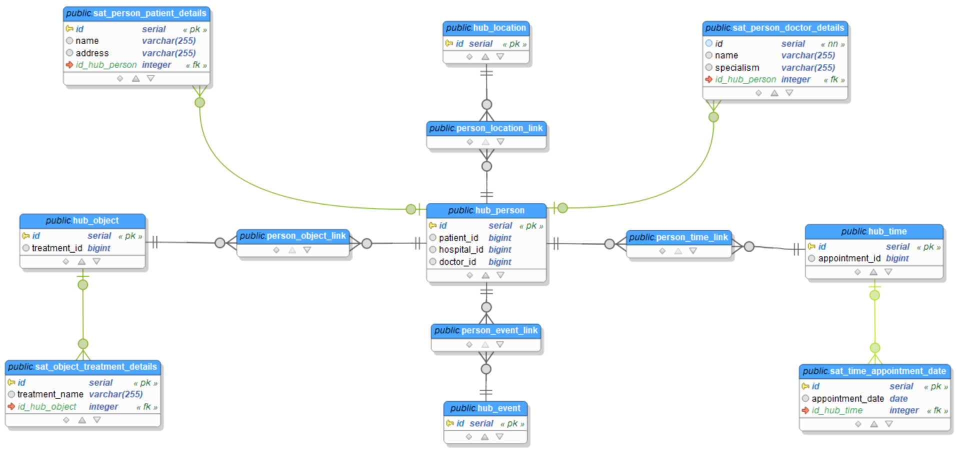


Figure A.6: The forth table now in place on the data vault

By maintaining a consistent core structure to the data vault, we are able to rapidly implement changes to the the incoming data as well as easily join together disparate data sets that were never intended to be joined in the first place.

Appendix B

Data Vault Design - Vault Bot

WP2's Smart Patient Health Records are generated in real time as a result of rules (Section 5.2) being applied to the source data (Section 4.3) which is then transformed in a data vault (Section 3.3). For this to run smoothly, we currently have to manually set the mapping of every source field to their destination Hub or Satellite (Appendix A).

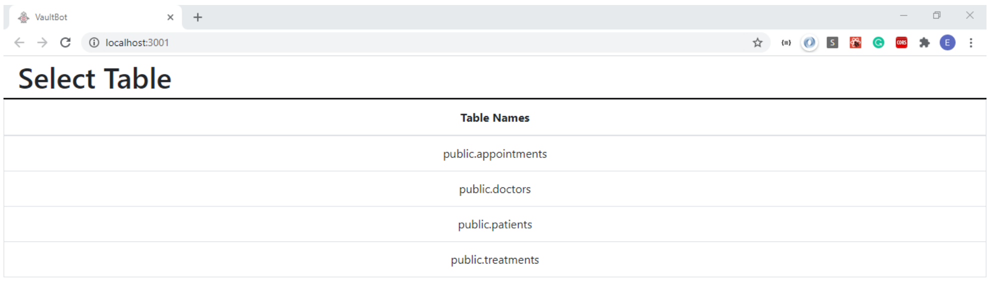
While the use cases for the **Serums** have largely stayed the same since the initial proposal, the data which the use case partners have wanted to include has changed repeatedly. With each data change come the need to manually edit or create both the new code to reflect these changes in the data vault structure and the control files which are used to map the fields from the source to their destination. This was a time consuming effort that was rife with room for error when working with some of the larger data sets.

To improve this situation, **SOPRA** has developed tooling known as **Vault Bot** to increase both the speed and accuracy of any changes to the source data that are requested. This chapter will cover some of the functionality that it offers. It is still a very early build, however, it has already improved the speed of development and further updates to the tool will continue to be added.

The first major improvement is that there is a graphical interface which links directly to the source database. This can be seen in Figures B.1, B.2, and B.3 (for the purposes of this chapter we are continuing to use the example hospital as used in Appendix A). The advantage this gives is that the source table, the field names, and the field types are captured automatically so there is no room for human error.

A further example of improvement can be found on the table design screen as shown in Figures B.2 and B.3. Whilst not required for our example here, it is entirely possible to design multiple Satellites at the same time from the same table, such as when subsets of the table are categorised under different Hub types or broken down to be more atomic. This ensures that all the correct business keys are correct across the Hubs as well as guaranteeing that the Link tables are correctly utilized during the transfer of data from source to the data vault.

The final improvement is the auto-generation of the data vault schema and the



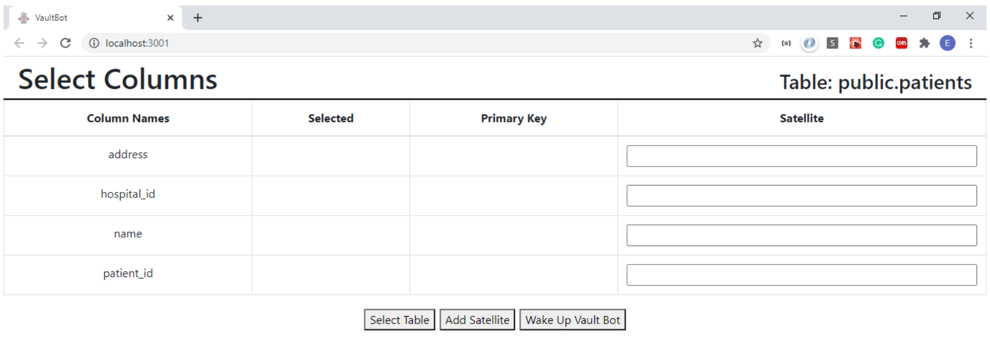
Select Table

Table Names
public.appointments
public.doctors
public.patients
public.treatments

Figure B.1: The table select screen for our fictitious hospital from Appendix A

control files which maps the source data onto its destination Hub or Satellite. The new schema is written as a series of SQLAlchemy classes and delivered as a file which can be ran in Python3 to both create the data vault and, if necessary, be used as a declarative class file for forming an Object-Relational Mapping (ORM). This allows interaction between the database and the outside world without the need for SQL, removing an otherwise potential avenue for misuse. Examples for the code for the Hubs, Links, and Satellites can be found in the Listings 12, 13, and 14 respectively.

The relevant control files are read by the API (Section 4.3) at the point at which the data is being copied from the source tables into the newly generated data vault. An example of the control file that is generated for the patients table can be found in the Listing 15.

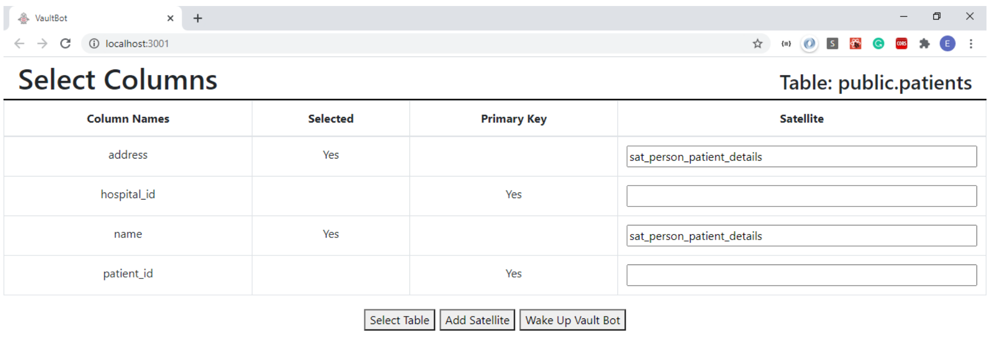


Select Columns

Table: public.patients

Column Names	Selected	Primary Key	Satellite
address			
hospital_id			
name			
patient_id			

Figure B.2: The data vault design screen for the patients table



Select Columns

Table: public.patients

Column Names	Selected	Primary Key	Satellite
address	Yes		sat_person_patient_details
hospital_id		Yes	
name	Yes		sat_person_patient_details
patient_id		Yes	

Figure B.3: The completed data vault design screen for the patients table

Listing 12: An example of the auto-generated code for a Hub

```
hub_person={'__tablename__': 'hub_person',
'__table_args__':{'schema':'public'},
'id': Column(column_types['integer'], primary_key=True)}
primary_keys = []
for keys, values in {
    'doctor_id': {'data_type': 'bigint'},
    'hospital_id': {'data_type': 'bigint'},
    'patient_id': {'data_type': 'bigint'}
}.items():
    primary_keys.append(
        {keys: Column(column_types[values['data_type']])}
    )
for key in primary_keys:
    hub_person.update(key)

person = type('PUBLIC_Hub_Person',(Base,),hub_person)
```

Listing 13: An example of the auto-generated code for a Link

```
person_object_link={'__tablename__': 'person_object_link',
'__table_args__':{'schema': 'public'},
'id': Column(column_types['integer'], primary_key=True),
'person_id': Column(column_types['integer'], ForeignKey(hub_person.id)),
'object_id': Column(column_types['integer'], ForeignKey(hub_object.id))}

person_object = type('PUBLIC_Person_Object_Link',(Base,),person_object_link)
```

Listing 14: Two examples of the auto-generated code for the Satellites

```
person_doctor_details = type(
    'PUBLIC_Sat_Person_Doctor_Details',(Base,), new_satellite
)

new_satellite={'__tablename__':'sat_person_patient_details',
'__table_args__':{'schema':  'public'},
'id': Column(column_types['integer'], primary_key=True),
'source_table': Column(column_types['string']),
'hub_id': Column(column_types['integer'], ForeignKey(hub_person.id))}

columns = []
for keys, values in {
    'name': {'data_type': 'varchar(255)'},
    'address': {'data_type': 'varchar(255)'}
}.items():
    columns.append(
        {keys: Column(column_types[values['data_type']])}
    )
for column in columns:
    new_satellite.update(column)


person_patient_details = type(
    'PUBLIC_Sat_Person_Patient_Details',(Base,), new_satellite
)

new_satellite={'__tablename__':'sat_object_treatments_details',
'__table_args__':{'schema':  'public'},
'id': Column(column_types['integer'], primary_key=True),
'source_table': Column(column_types['string']),
'hub_id': Column(column_types['integer'], ForeignKey(hub_object.id))}

columns = []
for keys, values in {
    'treatment_name': {'data_type': 'varchar(255)'}
}.items():
    columns.append({keys: Column(column_types[values['data_type']])})
for column in columns:
    new_satellite.update(column)

object_treatments_details = type(
    'PUBLIC_Sat_Object_Treatments_Details',(Base,), new_satellite
)
```

Listing 15: An example of an auto-generated control file

```
control_files = {}

public_patients_hubs = {
    'table': 'public.patients',
    'hubs': [
        {
            'hub': 'hub_person',
            'keys': [
                'patient_id',
                'hospital_id'
            ],
            'data_types': {
                'patient_id': {'data_type': 'bigint'},
                'name': {'data_type': 'varchar(255)'},
                'address': {'data_type': 'varchar(255)'},
                'hospital_id': {'data_type': 'bigint'}
            }
        }
    ]
}
public_patients_satellites = {
    'satellites': [
        {
            'satellite': 'sat_person_patient_details',
            'columns': [
                'name', 'address'
                ],
                'hub': 'hub_person',
                'hub_id': 0,
                'data_types': {
                    'name': {'data_type': 'varchar(255)'},
                    'address': {'data_type': 'varchar(255)'
                }
            },
            'source_table': 'public.patients'
        }
    ]
}
public_patients_links = {
    'links': []
}

control_files.update({'public.patients': {
    'hubs': public_patients_hubs,
    'satellites': public_patients_satellites,
    'links': public_patients_links
}})
```

Appendix C

The data lake directory tree

```
100-DL
├── 000-Workspace-Zone
├── 100-Raw-Zone
│   ├── 100-External
│   │   ├── 100-University-of-St-Andrews
│   │   ├── 200-Zuyderland-Medisch-Centrum
│   │   ├── 300-Fundacio-Clinic
│   │   └── 900-Other
│   ├── 200-Internal
│   │   ├── 100-CSV
│   │   ├── 200-TEXT
│   │   ├── 300-JSON
│   │   ├── 400-XML
│   │   └── 900-Human-in-the-Loop
│   └── 300-Archive
│       ├── 100-CSV
│       ├── 200-TEXT
│       ├── 300-JSON
│       ├── 400-XML
│       └── 900-Human-in-the-Loop
├── 200-Structured-Zone
├── 300-Curated-Zone
│   ├── Hub
│   │   ├── Event
│   │   ├── Location
│   │   ├── Object
│   │   ├── Person
│   │   └── Time
│   ├── Link
│   │   ├── Event-Location
│   │   │
│   │   ├── Event-Object
│   │   ├── Event-Person
│   │   ├── Event-Time
│   │   ├── Location-Event
│   │   ├── Location-Object
│   │   ├── Location-Person
│   │   ├── Location-Time
│   │   ├── Object-Event
│   │   ├── Object-Location
│   │   ├── Object-Person
│   │   ├── Object-Time
│   │   ├── Person-Event
│   │   ├── Person-Location
│   │   ├── Person-Object
│   │   ├── Person-Time
│   │   ├── Time-Event
│   │   ├── Time-Location
│   │   ├── Time-Object
│   │   └── Time-Person
│   └── Satellite
│       ├── Event
│       ├── Location
│       ├── Object
│       ├── Person
│       └── Time
├── 400-Consumer-Zone
└── 500-Analytics-Zone
```

Appendix D

The complete list of tags

The following is the current list of tags which have been prescribed to the use case partners of **Serums**. These were designed to cover all of the data for all of the use cases, with every field of every source table falling into at least one of them. With this list being a relatively new aspect of the project, there is a chance that it will be altered in some way before the final submission (D2.6).

- Personal
  - A very limited subset of data that usually only contains the patient's name, sex, and key measurements
- Patient Address
  - Limited to only the patient's address
- Patient Details
  - An expanded set of data to that of the *personal* tag that also contains the details found in *patient address*. Additional fields might also contain things such as telephone number, marital status, nationality, etc.
- Patient Appointments
  - Any data related to appointments such as the date, the reason for the appointment, and the name or id of the healthcare professional
- Wearable
  - Any data which is generated by a wearable medical device. Both ZMC and FCRB have examples of this technology in their use cases
- Diagnostic

  - Any data related to a patient's diagnosis. This covers a range of sources for all three use case partners that includes medical reports, specific data related to the location of the condition, the dates related to the condition's start and end, etc.

- Medication
  - Any data related to medication the patient has been prescribed as part of their treatment
- Operations
  - Any data related to an operation the patient has undertaken such as the date, the reason, and the outcome
- Documents
  - Any documents connected to a patient such as doctors' notes or referrals
- Treatments
  - Any data related to the treatments the patient is under going. This can include elements found in the *medication*, *operations*, and *documents* tags
- Healthcare Providers
  - Details of the healthcare providers. This includes elements such as name or id, department, and speciality
- Drugs and Alcohol
  - Any data related to substance abuse
- Allergies
  - Any data related to allergies
- Additional Information
  - Currently used as a catch all to cover edge cases which do not yet fit into another tag. Currently the data which is covered under this tag includes additional details about living conditions and notes about hearing difficulty
- All
  - Mostly used as a debug tool as it returns all of the data for a patient. It is unlikely this tag would be made available in the final version

Appendix E

An example Smart Patient Health Record

When a patient's Smart Patient Health Record (SPHR) is returned, it is encrypted. Full details of this can be found in Section 6.3. The following results are what would be returned from the request as depicted in Listing 16. Listing 17 shows how the data is returned to the user, whereas Listing 18 shows the data after decryption.

Listing 16: The body of a request for a SPHR

```
{
  "serums_id": 10523,
  "rule_id": "RULE_6d71c87c-726e-43c9-8887-7fee4be2fcd9",
  "hospital_id": "ZMC",
  "public_key": "-----BEGIN PUBLIC KEY-----
    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA24IJ1BITwdCmERIJgk2v
    plEwjsXNupYWE23cy8bSvmVhHuZZKRZU5YPGQwaFZERFT0/fJvtRFinYhe9KOmXw
    HYhrrDCIOgXkYvzJMZ+IWTDMHmrmuXjC/9UfarpzE4mudCIWCVffcGaItP2aMlhu
    E4dxxH9S7xbr4F6mRDmZBbup7H4hnkJAnsy4LMM8vf8eREk4VqW1MQhougdmDCi9
    Xt2ZvOxD4tVEW3wTTWUPm+1ZZk3eyV+twnbp2O3T9EIYQ1Nm7vNkjVgucSDXMKT6
    iCThYCG2a6ogf33Z5mPsSlAT+Q1IfL+8FNkbUH0K/ZJqV8SlI68FPZ/v/rC3TXLs
    4wIDAQAB
    -----END PUBLIC KEY-----"
}
```

Listing 17: An Encrypted SPHR and Key

```
{
  "key": "SEkV1o9WYGwq6ZUsMPPA4dPoW+HEcbYorcxZPGAsdTqenOvoyi
    w47QwcxsaemhAy6yntsIQcqO/wOuRKgnjoJLJb5p4vCWHsph94gujAlw
    DYvxAuTIuIb/AUUs7tDf8K4Iw6pVx3Nl9aKQuyKfShN27+p72g8eR6Vu
    D8JR4xK4nTOu18pKkHke8rrk39m7hCylhx0HKXYJERkNzXDglsy6TiD5
    U3RgBtoYEWvLWs8xnj7GQJOlSsoAqd+J6uzvqjtfzTuqSgsEG854WJf+
    DuWjllF7wNMi+WazvpX99jvCYorrCMSCFb/5LcERiqxccJaX9WEnhAF+
    88uzdN/TJg9Q==",
  "data": "gAAAAABfkX7nIH6hT5muPd9KR7ku-Hbinqa_JcC_ioxTXxteE
    Z4puV9BR3h6rJWO0JT2HSujp0KmhJ9lW2UObw3vBvXeZM0woSIyo3zhF
    u9SOq1BLTL4n7OpNoYuMWaMKaLAEXXu-suc-9nNhs3ak32S0IhJMRKl1
    01Ff0uDn2CpPJwfWcoZGVp-2mnZ28Zr1iD65XeENKoTRXq426Agfnc3S
    F0sVafIZjTOPDVM49Yb97oocf63BfvpWsliE4kar8zCKc3U3lZGSZbqm
    oXjo98C6x52BzwhxWApvn_X53OtHnvCUs3AvAsxvLhhE4hy4jqTXS1Ki
    l3jutTIu6CvpyVseVyM0Y1uQ5HtcMo6VV-MMoOUk38vJvy81o6cIvhnw
    BucV6W3yWGIMqXvlpmBTu1u_ul-h3MJNBzswb9mIdPAZADp7BvUbGqXL
    da19iCupUcvEOhmLZF2zcQ9XHh0V6yjeo6FJzs7IM68KwNfyxx24FgQD
    g6TotW1vHXUnOI9O8beYwCRoldjzdjOOFbxOAwPMBjSMR4x34EH7S8AB
    ZL-tf5CMZgyPWnflpSwn4vxt5NJtKmROKICT8l932XswN_5HrqNpFUjM
    kLNqu-FDIsbTPO-TBh4yvZ5aisUKpvop1sC1CbjmkCEyKDLiQ7tqrVVG
    613cT7qePfq0Q159YYNgkx8BXVRMEcWESdiaaktrkTKHZYKE8PK2yCiU
    Ci-mr6LZTLHuBX8K8vQ4cSs4IdHwc6p1N6knpSrWbvdfv4MEAimpErl9
    Z8RoWSso3smoHI3yOq6zdF3k1_uj_jsLOmUhvbJDcdRCtE-oBdxe2zG0
    A9ZwvTxNzpP2TqiDnIGmCjYM9lxmsU78_yYziT4FpTA2aOGhLiHgbtT2
    ...."
}
```

Listing 18: A Decrypted SPHR

```
{
  "serums_id": 1,
  "personalInfo": {
    "firstName": "Calimerass",
    "lastName": "CALIMERASS",
    "dateOfBirth": 323395200000,
    "gender": "2",
    "nationality": "NL",
    "weight": "NA",
    "height": "NA"
  },
  "events": [
    {
      "eventId": "NA",
      "associatedId": 1075835,
      "when": "NA",
      "where": "NA",
      "who": "NA",
      "what": [
        "wearable",
        "patient_details"
      ],
      "data_vault_table": "hub_event",
      "details": {
        "0": {
          "id": 1,
          "patnr": 1075835
        },
        "1": {
          "id": 2,
          "patnr": 1075835
        },
        "2": {
          "id": 3,
          "patnr": 1075835
        },
        "3": {
          "id": 4,
          "patnr": 1075835
        }
      }
    },
    {
      "eventId": "NA",
      "associatedId": 1075835,
      "when": "NA",
      "where": "NA",
      "who": "NA",
      "what": [
        "wearable",
      "patient_details"
    ],
    "data_vault_table": "hub_person",
    "details": {
      "0": {
        "id": 1,
        "patnr": 1075835
      },
      "1": {
        "id": 2,
        "patnr": 1075835
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "hub_location",
    "details": {
      "0": {
        "id": 1,
        "patnr": 1075835
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "hub_time",
    "details": {
      "0": {
        "id": 1,
        "patnr": 1075835
      },
      "1": {
        "id": 2,
        "patnr": 1075835
      },
      "2": {
        "id": 3,
        "patnr": 1075835
      },
      "3": {
        "id": 4,
        "patnr": 1075835
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "person_location_link",
    "details": {
      "0": {
        "id": 1,
        "person_id": 1,
        "location_id": 1
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "time_event_link",
    "details": {
      "0": {
        "id": 1,
        "time_id": 1,
        "event_id": 1
      },
      "1": {
        "id": 2,
        "time_id": 2,
        "event_id": 2
      },
      "2": {
        "id": 3,
        "time_id": 3,
        "event_id": 3
      },
    "3": {
      "id": 4,
      "time_id": 4,
      "event_id": 4
    }
  }
},
{
  "eventId": "NA",
  "associatedId": 1075835,
  "when": "NA",
  "where": "NA",
  "who": "NA",
  "what": [
    "wearable",
    "patient_details"
  ],
  "data_vault_table": "sat_event_exercise_measurements",
  "details": {
    "0": {
      "id": 1,
      "source_table": "wearable",
      "hub_id": 1,
      "nr_sst": 45,
      "steps_total": 6047,
      "cadence": 77,
      "cyc_rot": 245,
      "cyc_rpm": 39
    },
    "1": {
      "id": 2,
      "source_table": "wearable",
      "hub_id": 2,
      "nr_sst": 39,
      "steps_total": 5885,
      "cadence": 80,
      "cyc_rot": 0,
      "cyc_rpm": 0
    },
    "2": {
      "id": 3,
      "source_table": "wearable",
      "hub_id": 3,
      "nr_sst": 33,
      "steps_total": 5892,
      "cadence": 76,
      "cyc_rot": 0,
      "cyc_rpm": 0
    },
    "3": {
      "id": 4,
      "source_table": "wearable",
      "hub_id": 4,
      "nr_sst": 46,
        "steps_total": 6000,
        "cadence": 81,
        "cyc_rot": 0,
        "cyc_rpm": 0
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "sat_time_exercise_measurements",
    "details": {
      "0": {
        "id": 1,
        "source_table": "wearable",
        "hub_id": 1,
        "day_nr": 1,
        "time_total": 41300,
        "time_passive": 35849,
        "time_active": 5449,
        "time_sit": 31045,
        "time_stand": 4804,
        "time_walk": 5071,
        "time_cycle": 378,
        "time_hi": 0
      },
      "1": {
        "id": 2,
        "source_table": "wearable",
        "hub_id": 2,
        "day_nr": 2,
        "time_total": 42674,
        "time_passive": 37939,
        "time_active": 4733,
        "time_sit": 31227,
        "time_stand": 6712,
        "time_walk": 4733,
        "time_cycle": 0,
        "time_hi": 0
      },
      "2": {
        "id": 3,
        "source_table": "wearable",
        "hub_id": 3,
        "day_nr": 3,
        "time_total": 41071,
        "time_passive": 36401,
        "time_active": 4668,
        "time_sit": 31652,
        "time_stand": 4749,
        "time_walk": 4668,
        "time_cycle": 0,
        "time_hi": 0
      },
      "3": {
        "id": 4,
        "source_table": "wearable",
        "hub_id": 4,
        "day_nr": 4,
        "time_total": 45420,
        "time_passive": 40762,
        "time_active": 4655,
        "time_sit": 34417,
        "time_stand": 6345,
        "time_walk": 4655,
        "time_cycle": 0,
        "time_hi": 0
      }
    }
  },
  {
    "eventId": "NA",
    "associatedId": 1075835,
    "when": "NA",
    "where": "NA",
    "who": "NA",
    "what": [
      "wearable",
      "patient_details"
    ],
    "data_vault_table": "sat_person_patient_details",
    "details": {
      "0": {
        "id": 1,
        "source_table": "patient_details",
        "hub_id": 1,
        "gschl": "2",
        "nname": "Calimerass",
        "nnams": "CALIMERASS",
        "vname": "TSTO",
        "titel": null,
        "namzu": null,
        "gbdat": 323395200000,
        "gbnam": "Test PTE - IMS Team.",
        "gbnas": "TESTPTE-IMSTEAM.",
        "gland": "NL",
        "natio": "NL",
        "land": "NL",
        "telf1": "06-21185097"
      }
    }
```

```
      },
      {
        "eventId": "NA",
        "associatedId": 1075835,
        "when": "NA",
        "where": "NA",
        "who": "NA",
        "what": [
          "wearable",
          "patient_details"
        ],
        "data_vault_table": "sat_location_patient_address",
        "details": {
          "0": {
            "id": 1,
            "source_table": "patient_details",
            "hub_id": 1,
            "pstlz": "2986 VA",
            "ort": "Ridderkerk",
            "stras": "Koolmees 13 A bus 5"
          }
        }
      },
      {
        "eventId": "NA",
        "associatedId": 1075835,
        "when": "NA",
        "where": "NA",
        "who": "NA",
        "what": [
          "wearable",
          "patient_details"
        ],
        "data_vault_table": "sat_person_patient_measurements",
        "details": {
          "0": {
            "id": 1,
            "source_table": "patient_measurements",
            "hub_id": 2,
            "height": 195,
            "weight": 90,
            "date": 1537401600000
          }
        }
      }
    ]
  }
```