decode

D3.7 Control and entitlement system for sensor data owners

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 732546

Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

D3.7 Control and entitlement system for sensor data owners

Version Number: V1.0

Lead beneficiary: Thingful

Due Date: January 31st, 2019

Author(s): Samuel Mulube (Thingful), Usman Haque (Thingful)

Editors and reviewers: Francesca Bria, Oleguer Sagarra (IMI), James Barrit (TW), Denis Rojo (Dyne.org)

Dissemination level:		
PU	Public	**X**
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)	

Approved by: Francesca Bria (Chief Technology and Digital Innovation Officer, Barcelona City Hall)

Date: [31/01/2019]

This report is currently awaiting approval from the EC and cannot be not considered to be a final version.

Contents

Tables

Figures

1 Introduction

DECODE is a project that aims to explore what societal benefits might be obtained should individuals be given the ability to exercise control over the generation, sharing and transmission of their personal data. This exploration has touched on a number of areas; however, this document is a report on the implementation of a possible approach to allowing users to exercise direct control over data generated by the low powered, and pervasive computing devices known commonly as the Internet of Things (IoT).

The Internet of Things describes a paradigm where low powered microprocessors can be embedded in an ever-increasing range of objects to imbue these everyday objects with behaviours and capabilities that would not be possible with non "smart" devices.

The development of these devices certainly has the potential to greatly enrich the lives of people; however, there is a significant potential downside to these devices in that these devices typically collect, store and transmit onwards data about end users, and users typically have very little visibility into what is collected, and where it goes once it has been transmitted back to the provider of the device.

There have been a number of significant examples of the issues with this sort of promiscuous data collection in recent years ranging from the sex toy manufacturer [1] whose devices were collecting and storing intimate details of user's sexual habits without their consent, to connected toys [2] or dolls [3] that collected data from millions of children (including images, audio and private chats) and were then unfortunately hacked and these databases sold online. There was also the case of the fitness tracking app [4] that inadvertently leaked the locations and layouts of secret US army bases, as well as a recent slightly troubling story of a dental insurer [5] whose business model appears to include sending free "smart" toothbrushes to end users and strongly encouraging them to install their app and which unobtrusively shares their brushing data. Even more recently a story has emerged concerning Amazon's Ring [6, 7] home security cameras which alleges that the video streams recorded within people's homes were stored in unencrypted form, and so were available for any staff within the company development team to view at will.

With these concerns in mind the IoT pilot of DECODE set out to try and think about mechanisms that could be developed that put control over the data collected by these devices back in the hands of users.

1.1 Project Participants

1.1.1 IMI (Institut Municipal d'Informàtica)

IMI [7], lead in this aspect by Dr Oleguer Sagarra, are the consortium partner who have taken the lead in managing the whole process of developing the IoT pilot, coordinating between the many partners and in designing the architecture of the system.

Their role within the project has been:

1. Project management, including facilitating a number of important meetings between all partners on site in Barcelona to design the system
2. Significant contributions to the system architecture and conceptual models that inform the whole project.

1.1.2 Thingful

Thingful are the primary developer of software components for the IoT pilot for DECODE. Thingful's role in the project is to:

1. Design the architecture of the IoT pilot
2. Implement the new software components that will fulfil this architecture
3. Support other consortium partners in integrating and deploying these components.

1.1.3 SmartCitizen

SmartCitizen are not part of the DECODE consortium however they have been enlisted to help support the IoT pilot in two ways: they have extensive experience in organising citizen science community engagement events (see the Making Sense project [9]), but they also have a powerful IoT platform and experience of building and deploying IoT sensor devices that could be used to demonstrate the benefits of the DECODE approach to giving users autonomy over the data generated by IoT devices.

SmartCitizen's role in the project then will be to:

1. organise a series of events where we will enlist citizens to take part in the project
2. distribute and deploy a number of sensor devices to community participants and use their data platform to collect data which we will then connect in to DECODE
3. be responsible for running the new components developed for DECODE on their infrastructure

1.1.4 Eurecat

Eurecat [10] are another consortium partner who are responsible for the development of the Barcelona Now (http://bcnnow.decodeproject.eu/) dashboard that is a crucial part of both of the Barcelona based pilots, as it is through the dashboard that end users will be able to see and interact with their data.

As such Eurecat's role in the IoT pilot is to:

1. Extend the functionality of the BCNNow dashboard to represent the IoT data collected by the devices distributed by SmartCitizen
2. To develop a new component that is able to read and decrypt the encoded data generated by the participating devices.
3. To develop an authentication layer to the dashboard that will allow access to be restricted to only users with valid credentials.

1.1.5 Thoughtworks

Thoughtworks role within the IoT pilot part of the DECODE project was to act as the crucial technical coordinating partner that helped all the other partners to work together in order to deliver the goals of the project. In addition, they had a team dedicated to DECODE who were responsible for the development of the mobile application that would be given to users in order to participate in DECODE.

1.1.6 Dyne.org

Dyne.org is a non-profit free software foundry with more than 15 years of expertise in developing tools and narratives for community empowerment. Their role within the consortium revolves around developing practical cryptographic tools that aim to facilitate the projects objectives of building tools that allow users to exercise autonomy over their data.

In relation to the IoT pilot Dyne have developed the Zenroom virtual machine (see 8.1 for more details). As secure encryption is at the heart of the DECODE philosophy we early on made the decision to use Zenroom for all cryptographic operations. This common substrate would allow us to share expertise effectively within the consortium so allowing partners like Thingful to use best practice encryption techniques that would be portable across applications.

2 Requirements and analysis

One of the promises of the Internet of Things (IoT) is that everything should talk to everything else. These talkative "things" include sensors, consumer appliances, home automation systems, and even connected vehicles. The frameworks through which such interconnectivity is arranged, controlled, and mediated – that is, how these things "entitle" each other to connect – is going to be a fundamental part of IoT. Managing this "entitlement," which defines who can access your device data, and under what conditions it can be found and used by others, will be one of the major challenges for consumers and businesses.

The data from all these things will be valuable not just to the companies that deploy them, but also to people or companies operating in other domains. For example, your thermostat might talk to your neighbor's weather station to determine an appropriate temperature setting, and then switch on the heating when your phone's GPS tells it that you're nearing home.

It's this many-to-many and cross-domain aspect of connectivity that distinguishes IoT from earlier remote monitoring/control systems and M2M (machine-to-machine) systems, where only one organization created, owned, and used the data. In the IoT, each connection won't be predetermined; these things should be able to structure their conversations on the fly, in an automated and ad-hoc manner. But this raises a number of questions and concerns around discovery, privacy, interoperability, and data-access privileges.

The problem space we decided to explore for this pilot doesn't touch on any questions of discovery or interoperability, but rather focuses on the aspects of privacy and data-access privileges, specifically trying to explore whether the use of strong encryption techniques might provide a mechanism for giving users autonomy over the use of their data.

2.1 Scope of the document

This document aims to cover the design of the components built by Thingful which are to be delivered to SmartCitizen to be operated within the IoT pilot, so we aim to cover the following areas:

- Overall system conceptual and architectural design

- Technical implementation details of the core components to be developed by Thingful, including any supporting libraries
- References to all completed components
- Defaults of encryption mechanisms and algorithms

Out of scope for this document are areas that are part of the overall system, but not the direct responsibility of Thingful. These areas include:

- Details of the implementation of the DECODE app (wallet)
- Specifications of the cryptographic authentication protocol to be used between SmartCitizen and the DECODE app, and then the DECODE app and the BarcelonaNow dashboard.

Also at the time of writing we have not yet heard from SmartCitizen the details of where the developed components will be deployed, presumably they will live on a SmartCitizen domain, but details on that are not yet known.

2.2 Conceptual Model

The field of cryptography sometimes moves at a heady pace, however modern, strong encryption techniques are thought to offer very effective encryption that we can have good confidence will protect the encrypted data from unauthorized parties (most of us don't have a nation state as a threat model, and even if we did it's certainly not guaranteed that even a nation state would be able to break a correctly implemented cryptographic scheme).

Therefore, the basic concept we decided to explore was whether we could design a system that provided the owner of a device (the data producer) with tools that would allow them to create strongly encrypted streams of data with different views of the source data from their devices, encrypted for specific identified recipients.

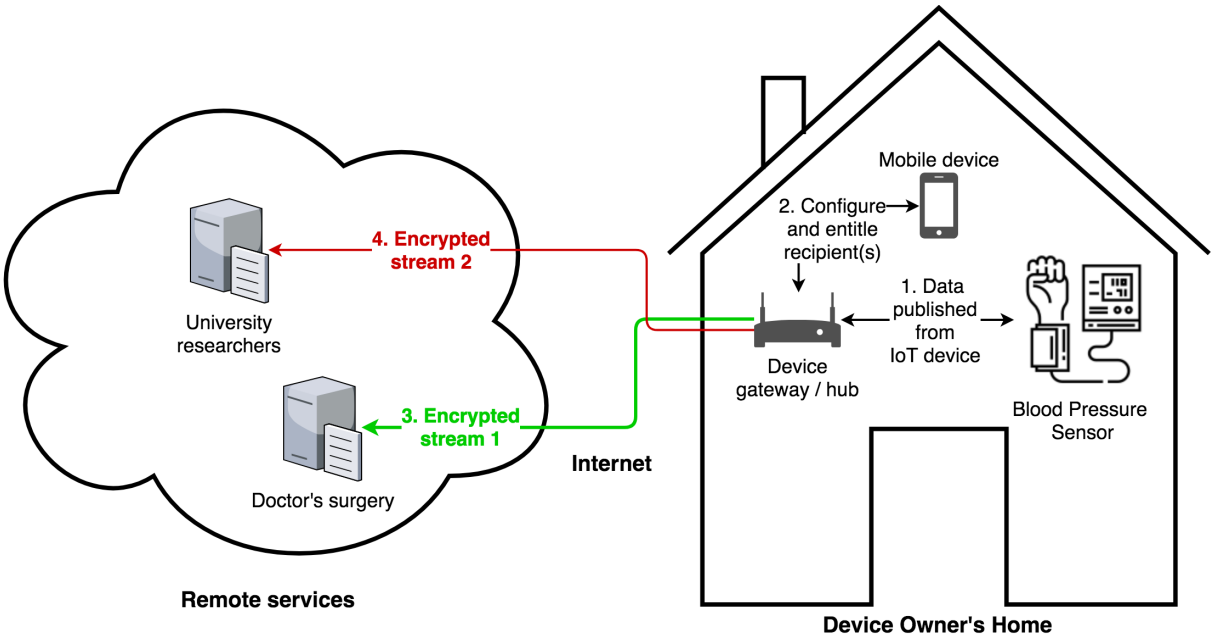

Figure 1 - Conceptual view of encryption scheme

The schematic shown in Figure 1 shows an early conceptual model of how the entitlement and encryption mechanism might work. In the diagram shown we have a hypothetical "smart" blood pressure sensor device which has been configured to connect and share data for local use to a local device gateway or hub. This device gateway could provide a local view of the data which would allow the user to track their health themselves, however to explore the ideas of entitlement and encryption, the gateway would also include some service that would allow the user to create differential encrypted streams that could be published to remote services.

In the example shown in the diagram, the user has received the public key of their doctor's surgery, then using their mobile device they have then created a new encrypted stream using that public key (step 2) which shares the precise readings from the sensor with the doctor's surgery.

Thereafter the device gateway / hub will start sending all data received from the blood pressure sensor across the internet to the system being operated by their doctor. Because the data has been encrypted with the doctor's surgery public key (and a private key kept within the gateway device), the individual can be confident that provided both parties keep their private keys secure, there is very low risk of someone intercepting that data.

In addition, the user has also received the public key of a set of university researchers and has decided to create a different encrypted stream of data for them. This stream will not be the precise values read from the sensor, but rather an aggregated view of

the data that provides enough information to the researchers to be useful for their models but doesn't breach the privacy of the individual.

The above model skims over the implementation details in terms how the user might obtain those keys, and trust their provenance, but the idea shown formed the basis of the IoT pilot. In section 2.3 we look in a little more detail at how we decided to implement the creation of different views of the data (entitlement policies), and in section 2.4 we look in a little more detail at the encryption scheme we chose for the pilot.

2.3 Entitlement Policy Definition and Design

Before giving details of the system design process, we should first define what we mean by an Entitlement Policy in the context of the IoT pilot as the term Policy may have many different meanings, particularly in the context of encryption schemes.

When we use the term Entitlement Policy we are talking about a set of operations that may be defined and applied to the raw sensor data being emitted by an IoT device. To give a concrete example, let's say we have an IoT device that has the following sensors on board, and is capable of emitting a data stream with all of these values:

- Temperature (℃)
- Light levels (lux)
- Sound level (dBA)
- Carbon Monoxide levels (ppm)
- Battery level (%)

A user may wish to entitle some recipient to receive a view of that data, but they may not wish to share all of those values as to do so may compromise some aspects of their privacy.

For example, the combination of the temperature, light and sound level sensors of the above device could reveal very precisely whether or not the owner of the device was currently at home, and if this data was made public (either intentionally or by accident), then an unscrupulous actor could use the information to target the user's property. In contrast the carbon monoxide levels, or the batter level sensors could be safely shared as they are not subject to the same potential information leakage.

To give another example, given a medical IoT device that measures blood pressure, the owner of such a device may be perfectly happy to share the precise values measured with their doctor, but only to share aggregated data with a research project looking at blood pressure across a cohort of participants.

To address these specific requirements, we define an Entitlement Policy to be:

A set of operations defined for each individual sensor value emitted by a device, where an operation can be one of three functions:

- SHARE – this function is the simplest and just means that any data received for this particular sensor should be transmitted onwards exactly as received, i.e. a temperature reading of 23.2 ℃, should be emitted exactly as received from the sensor.
- BIN – this function means rather than emitting the raw value, instead we apply a binning function and emit this value instead. To use the same example temperature as the previous example, say we had a set of bins defined as being from 0-15 ℃, 15-30 ℃, and finally anything over 30 ℃, given an input of 23.2 ℃ we should output a value where we just record a single instance within the 15-30 ℃ bin.
- MOVING_AVERAGE – this function is much simpler to understand than binning, but here we would just allow the policy to define a time window (5 minutes, 6 hours) and rather than emitting the raw 23.2 value instead we would emit a moving average defined over the given time interval, which in this case might be 21.9.

In addition to ensure that only the desired recipient can see this data, after the filtering or transformative operations have been performed on the data, we also plan to use the strong encryption provided by Zenroom by giving each policy a public/private key pair meaning that the recipient will only be able to decrypt and use the data if they not only know the identifier of a policy, but also are in possession of the corresponding private key by which data has been encrypted.

2.4 Proposed Encryption Scheme

The encryption algorithm we chose for the IoT pilot is an algorithm offered by Zenroom which allows us to use the same encryption and decryption scripts uniformly across any components that require encryption or decryption.

The algorithm chosen is called AES-GCM [8], which is an AES (Advanced Encryption Standard) based block cipher [9] using counter-mode and a random public initialization vector (IV).

To inspect the encryption and corresponding decryption scripts please see the appendices 8.1.1 and 8.1.2.

2.5 Design Iterations

The goal of the IoT pilot was to design a system that allowed a user to securely encrypt data at source and give the user direct control over that encryption meaning they had the power to choose what data to collect and where it should go. Once data was encrypted it could then be safely published to any untrusted repository as only entities with a valid credential would be able to decrypt and understand the data.

The system shown in Figure 2 - Original system schematic shows our original design for the system.

In the system shown there are physical IoT devices that were intended to be distributable to participants in the project. This device was planned to include some virtual sensors that would generate simulated IoT data.

The concept then was that devices could be "claimed" by a user by means of a cryptographic protocol called IRMA [12], and then once claimed the users would make use of an application on their mobile device (described as a "wallet") which would perform the dual job of both securely storing cryptographic attributes that proved the user owned the device, but also provide a UI by which the user could exercise autonomy over their data.

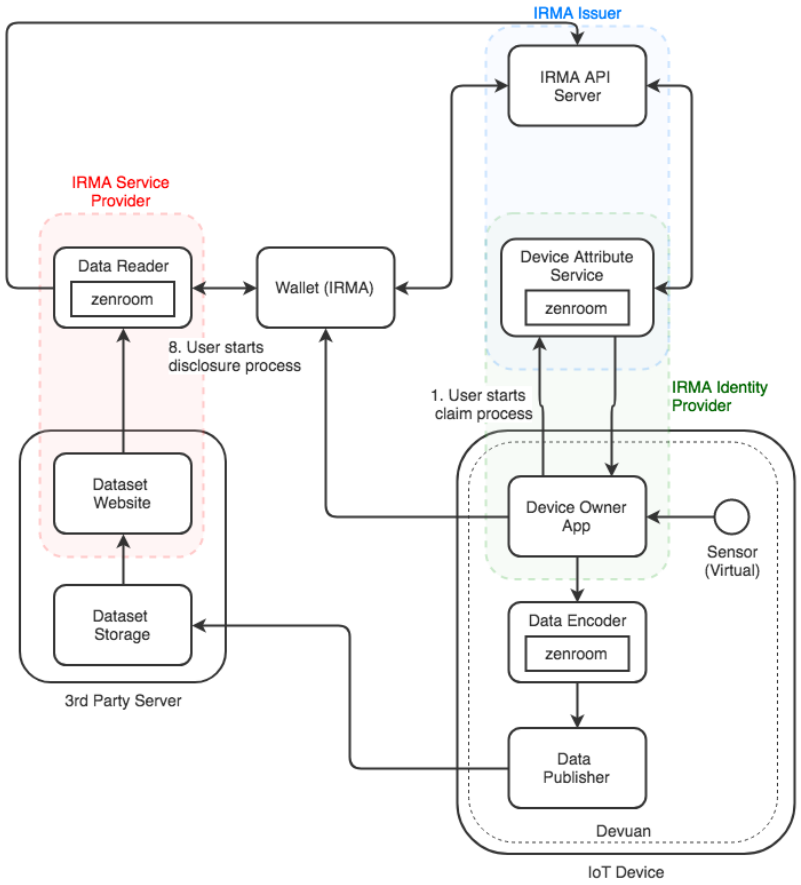

Figure 2 - Original system schematic

This design required the project to have access to a sufficient number of IoT devices such that we could distribute these devices to individuals wishing to participate in the DECODE project, as well as requiring that DECODE partners themselves recruit sufficient participants to make the project worthwhile. Unfortunately, it was not possible for the project to obtain sufficient devices on the required timescale, so the project decided to take a slightly different approach, by making use of the expertise of SmartCitizen in terms of both their existing infrastructure and expertise in distributing IoT devices to citizens, and in recruiting participants to take part in citizen engagement events.

The revised system schematic we proposed looked like the system shown in Figure 3 - IoT pilot scale model. In this model which was described as a scale model, we aimed to take advantage of SmartCitizen's existing infrastructure for collecting data from IoT devices over a messaging protocol called MQTT, and rather than have an encoder and configuration system deployed directly onto the IoT devices; instead we could subscribe to SmartCitizen's MQTT stream and simulate the behaviour of the system as if the encoding and filtering of data was taking place directly on the device. This had two key advantages – we could be much flexible in terms of the software that will

implement the encoding of the raw data; if we were required to update the firmware on devices every time we needed to make an update we would be required to think about much longer timescales to build and test the components.

In this revised configuration the "wallet" application would still be the primary user facing component, as It would be this that the user would use in order to claim devices and manage encrypted streams.

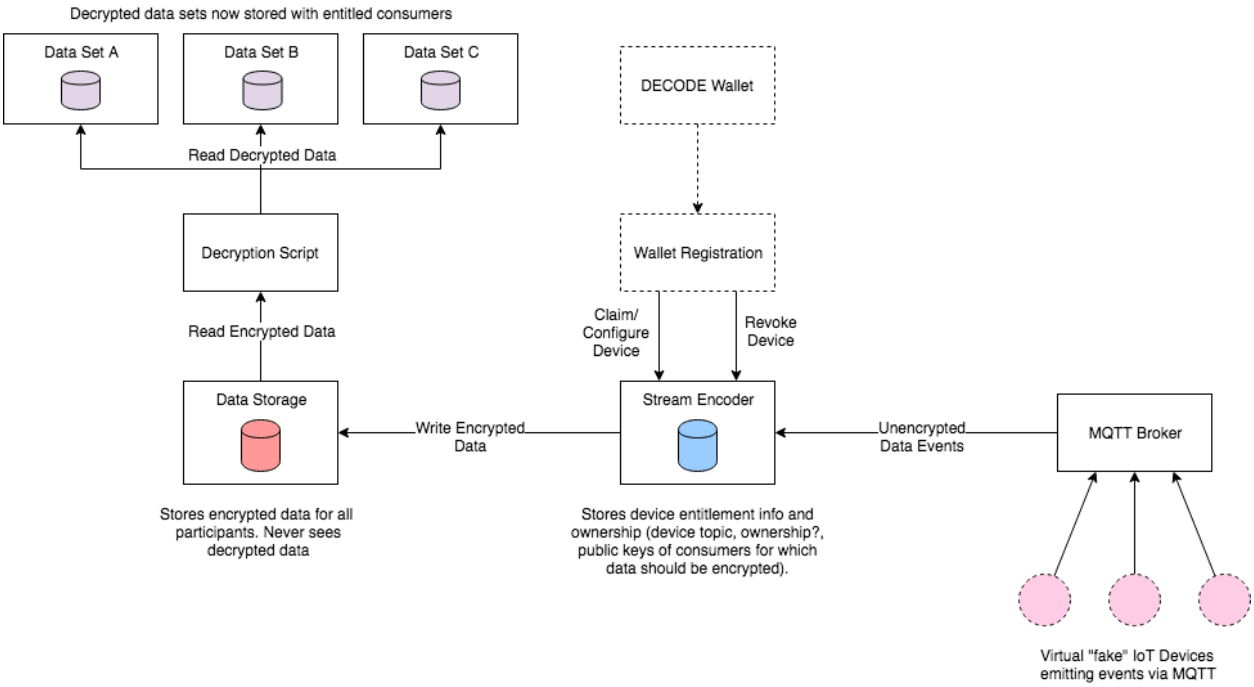

Figure 3 - IoT pilot scale model

The main problem with the system described above is that the mechanism by which the initial device registration and subsequent claiming of devices was supposed to work was very poorly defined, with no clarity as the mechanism by which this was supposed to occur.

However, at this stage Thingful did build some initial implementations of the Encrypted datastore, the Stream encoder, and the Device registration component. Before too much work was done on this system, we were able to have a focused collaboration day with all the participants of the IoT pilot in which we were able to collectively refine the design shown above to come up with a simpler and more coherent model that eliminated the uncertainties in the design. This final system design will be described in detail in section 3.

3 System Design

The final system architecture we arrived at after several iterations is shown in: Figure 4 - Final system schematic.

In this final formulation of the system we have three key components to be developed by Thingful:

- Policy store
- Encrypted datastore
- Stream encoder

In addition, the following components need to be developed by partners in the project:

- DECODE app (wallet)
- Onboarding Application
- Data collector
- Dashboard

These components are then combined to participate in a slightly complex set of interactions, including SmartCitizen's onboarding flow when devices are brought into the system and the corresponding authentication flow to the BarcelonaNow dashboard created by Eurecat when users authenticate their identity to the dashboard, in order to implement the proposed system functionality.

3.1 Final System Architecture

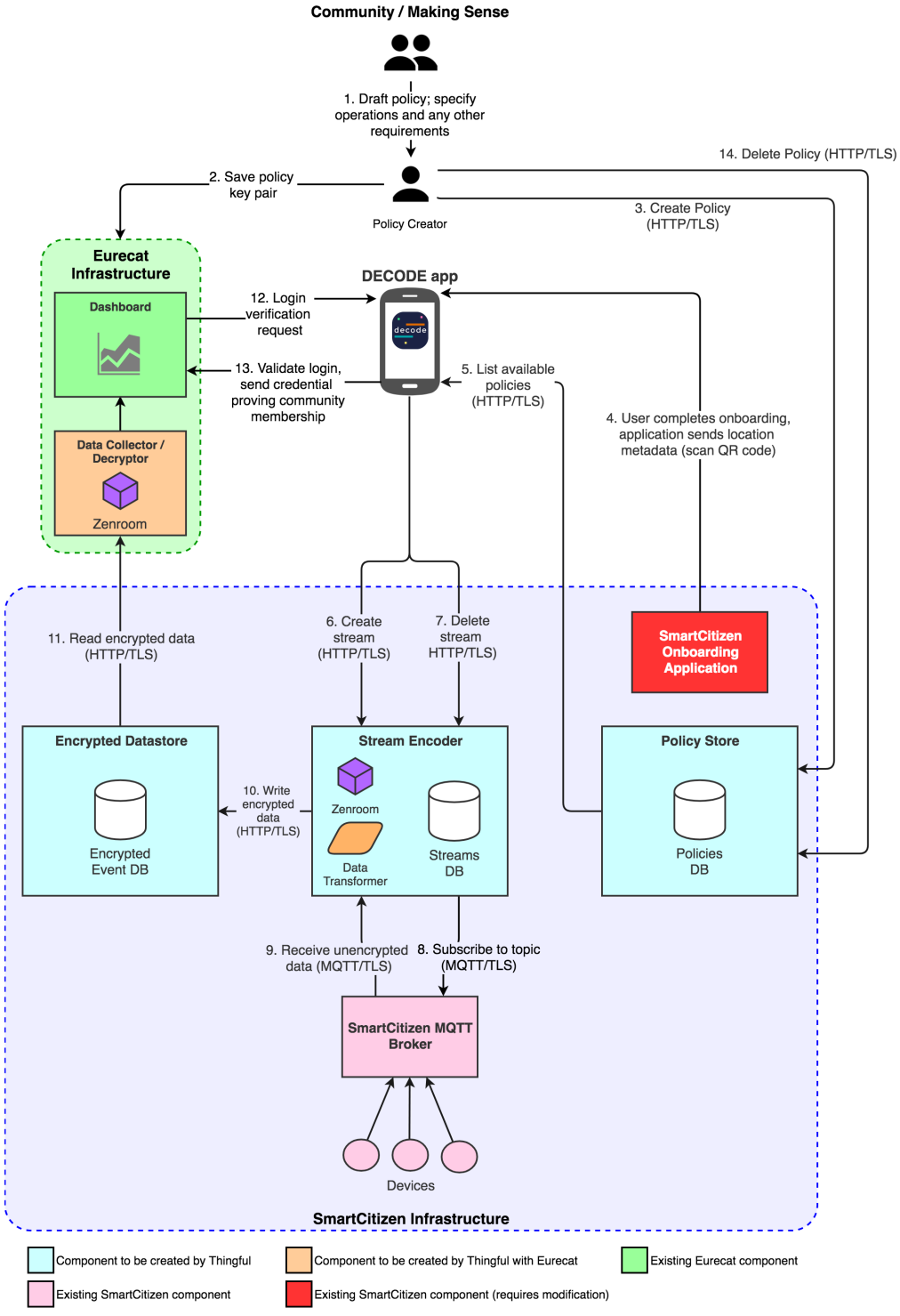

Figure 4 - Final system schematic

3.1.1 Policy store

The Policy store is a simple datastore which is designed to store definitions of "policies", where a "policy" is defined as being the combination of a cryptographic key pair (which will be used to encrypt data being collected for that policy), a descriptive label by which the policy can be identified, and then an optional list of operations to be applied to the raw data of any devices that have chosen to use this policy. The Policy store should expose a simple API which allows policies to be created or deleted by callers with appropriate permissions.

3.1.2 Encrypted datastore

This component is a simple datastore that is able to reliably store encrypted data events received from devices. This datastore can be thought of as existing on a public untrusted network as all data stored within it is encrypted using Zenroom, meaning that only entities with the correct private key will be able to read this data. The Encrypted datastore needs to expose an API which allows encrypted events to be written to the datastore, and then read back by a client. When reading data, the datastore should be able to handle paginating through potentially large lists of events.

3.1.3 Stream Encoder

This component is the component that is responsible for subscribing and consuming the events published by SmartCitizen over MQTT. This encoder component is also responsible for applying any filtering or transformative operations defined for the data, and then encrypting the data using Zenroom, and then writing the encrypted events into the secure datastore. The Stream encoder should expose an API which focuses on creating or deleting streams for callers using their DECODE app in order to manage their device's data.

3.1.4 DECODE app (wallet)

This component was intended to be developed by Thoughtworks, as it is the same component that is planned to be used for the other Barcelona based pilot that is part of DECODE. It is a React Native application that is designed to be able to take part in the various cryptographic interactions defined for DECODE, including things like being able to sign petitions (for the Decidim pilot), and interact with the IoT pilot components in terms of generating keys, and storing attributes securely within it that prove the user's ownership of IoT devices.

3.1.5 Onboarding application

The onboarding application is the bridge between SmartCitizen's components and the new components being developed for DECODE. SmartCitizen already have a sophisticated onboarding procedure that they wish to keep using in order to allow users to configure their devices. In order to minimise the impact on the system an approach was suggested whereby SmartCitizen continue to use their existing system, but at the end of that process, they create a QR code that captures the final output of that onboarding flow. The suggestion is that at this point the user will be able to scan that QR code from their DECODE app, and this will pull all the required attributes into the wallet ready to be used to create or update streams.

3.1.6 Data Collector

The Data Collector is a sub component of the BarcelonaNow dashboard and has been singled out only to discuss the usage of Zenroom again. In the Data collector that Eurecat will be implementing we will be able to use Zenroom via its Python bindings in order to decrypt encrypted events in a straightforward way.

3.1.7 Dashboard

The BarcelonaNow dashboard is the visible output that participants in the project expect to be able to use in order to see the data they have been collecting. The dashboard already exists and has some visualizations; however, the dashboard will be extended in order to handle the IoT sensor data that will be generated for participating devices. In addition, the dashboard is required to be able to respond to authentication requests made by users and handle the response to ensure that only authenticated users are able to access protected resources.

3.2 System User Flow

We'll talk briefly through how the components developed work together in order to deliver the required functionality. The numbers shown here relate to the numbers shown in Figure 4 - Final system schematic.

1. A core requirement for DECODE is to involve the participants in authoring the data entitlement policies they wish to make available to everyone participating in the project, so step 1 in the diagram represents a community engagement meeting at which a set of policies are logically defined. These policies are comprised of the different combinations of the operations described in section 0.

2. A cryptographic key pair is generated for each policy (using Zenroom). The private key must be kept secret, and only known to the entity that is the desired recipient of the data (here it would be the BCNNow dashboard being administered by Eurecat (step 2 in the diagram).
3. A call is then made to the Policy store to create a new representation of this policy within the Policy store. The call must contain the public key part of the key pair, a label identifying the policy, as well as the definition of all the operations the policy contains. This is step 3 on the diagram. This create operation returns a unique ID for the policy as well as a secret token which the caller will require should they wish to delete the policy (step 14).
4. We now jump to a user who has been given a device which they wish to add to the DECODE ecosystem at one of the events organised by SmartCitizen. SmartCitizen have an existing onboarding application that cleverly guides the user through installing and configuring a device. The concept is that at the end of that onboarding flow SmartCitizen will modify their onboarding application to generate a QR code which contains some metadata about the device. In step 4 the user scans this QR code using the DECODE app. This process reads these attributes into the app which is the procedure by which the user "claims" ownership of the device.
5. The DECODE app now needs to present a list of available policies to the user, so it makes a call to the Policy store to read a list of all policies currently in existence. The Policy store returns this list of policies, then it is the responsibility of the DECODE app to present this information in a UI that allows them to make an informed decision about where to choose to send their data.
6. Once the user has made a decision, they choose the appropriate policy or policies in the DECODE app, and then in step 6 the DECODE app makes a call to the Stream encoder. This message must include the following information: the device identifier, the metadata received from SmartCitizen in relation to the device (location, exposure), as well as a copy of the definition of the policy's operations. If this create operation is successful a new stream will be created within the encoder and the server will return a unique ID as well as a secret token that would have to be stored within the DECODE app.
7. Should the user wish to delete a stream, the DECODE app will also provide a UI for viewing a list of created policies, and from those policies allowing the user to choose one to delete, which would cause the DECODE app to send a delete request to the Stream encoder (step 7 in the diagram).
8. Internal to the Stream encoder when a create stream request is received, the Stream encoder is responsible for creating a subscription to SmartCitizen's MQTT broker (step 8). Once this subscription is in place, the Stream encoder will start listening for events broadcast for the device in question.

9. As events are generated by the device within the user's home, these events are emitted over MQTT, and then rebroadcast via SmartCitizen's MQTT broker (step 9).
10. The Stream encoder is then responsible for receiving that event, loading the policy in question, and then first processing the received data packet to first create an enriched view of the data where we enhance the received data with metadata about the device (i.e. its location and exposure), as well as metadata about the sensors (i.e. the sensor name, type and unit). The encoder then applies any of the filtering operations defined within the policy (moving average or binning the data). If no operations are defined the Stream encoder is designed to broadcast on all sensor fields in unchanged form. Finally, once we have the message packet we want to write to the datastore, the Stream encoder is responsible for using Zenroom, and applying an encryption script to the data before writing this encrypted message to the Encrypted datastore (step 10).
11. The data collector of the BCNNow dashboard is then responsible for reading data from the Encrypted datastore. This operation is to be performed on a regular interval (i.e. hourly) for all policies currently registered with the data collector. The data collector uses Zenroom (via the Python bindings described in section 5.5) to decrypt the data (the decryption script is shown in appendix 8.1.2), and this data is written to the dashboard's data store (step 11).
12. Now the data exists within the dashboard, all that remains is for the user to be able to prove to the BCNNow dashboard that they have the right to view the data for that policy. This interaction happens between the dashboard and the DECODE app and also makes use of Zenroom to facilitate a cryptographically secure proof which results in the user being able to share a credential with the dashboard that logs them in (steps 12 and 13).

4 Implementation

4.1 Technology Choices

Here we could talk about:

- Golang – language
- PostgreSQL
- Redis
- Zenroom
- Twirp/Protocol Buffers

4.1.1 Go (language)

The language we chose to implement the components was a language called Go [13]. Go is a programming language originally created at Google, which allows us to create high performance network servers with a robust and modern toolchain that fulfils our requirements for deployment strategies in that it compiles down to a single binary which contains everything it needs in order to operate.

4.1.2 Docker

Docker [7] is a tool for operating system virtualization that provides a relatively simple way for developers to package their applications inside very light weight virtual machines (typically called containers), that can then be deployed and run on a wide variety of platforms. We decided to use Docker as a primary delivery method for our components as SmartCitizen are already comfortable with operating applications packaged in this way, and in addition it is an effective tool to work with as a developer.

4.1.3 PostgreSQL

The primary database we selected for the components within the IoT pilot is a database system called PostgreSQL. This is a very mature, performant and reliable relational database that is very widely used to support many internet applications. A possible reason not to choose PostgreSQL is that it does not have as good support as modern non-relational database for building flexible clusters for the data. However we determined that for the amount of data likely to be generated for the IoT pilot PostgreSQL should be easily able to cope.

4.1.4 Redis

The secondary database system we propose using is one called Redis. Redis is a non-relational datastore, that provides a number of high performance data structure persistence. Typically, it is used for data that has lower permanence requirements than the data typically written to PostgreSQL, e.g. caching, but it also provides a unique data structure called a sorted set that we can use in order to generate moving averages of received values without having to implement complicated moving average logic within our application.

4.1.5 Zenroom

Zenroom is a tool created as part of DECODE by Dyne [14], that aims to create a virtual machine for performing cryptographic operations. There are more details in this in

4.1.6 Twirp / Protocol Buffers

When planning how we might implement the components defined for the IoT pilot we chose to use a technology called Twirp [15]. Twirp is an RPC framework developed at Twitch that makes use of a technology called Protocol Buffers [16].

Protocol Buffers are a language neutral, platform neutral way of defining interfaces for serializing/deserializing and transmitting structured data. In contrast to the prevailing paradigm for HTTP interfaces which commonly use something called REST (or Representational State Transfer), Protocol Buffer interfaces use a style called RPC (or Remote Procedure Call). In REST the metaphor is that we are passing around representations of the state of objects using the set of "verbs" defined in HTTP (i.e. GET, POST, PUT, PATCH, DELETE). In contrast RPC interfaces are more akin to calling functions, passing parameters to those functions and receiving data back as the return values from those functions.

Fundamentally there isn't a huge difference between RESTful interfaces and RPC interfaces in that can be used to perform exactly the same operations, however the big advantage we saw in using Protocol Buffers is that we could define our interfaces precisely via Protocol Buffer specification, and this specification could then generate a client library as well as a server stubs which would speed up our development cycle.

4.2 Development Methodology

The DECODE consortium as a whole using an agile methodology, attempt to iterate quickly on components, share with partners through regular weekly "stand-up" style meetings to elicit any feedback. We also had the benefit of those intense review days

where we were able to take advantage of Thoughtworks expertise in this area to refine and clarify the design.

5 Results and discussion

This section contains specific details of each of the components developed for the IoT demonstrator.

5.1 Policy Store

5.1.1 Description

As previously described in section 3.1.1 the Policy store is a simple component that provides a read/write/delete API for storing policies. Policy creators will use the write/delete functions, while consumers will read policies which can then be applied to devices.

The primary interface offered by the Policy store is a Protocol Buffers over HTTP interface generated using Twirp (described in 4.1.6), however the server also exposes a JSON over HTTP interface that is callable from any standard HTTP client. This allows us to use the more performant Protocol Buffer interface where we can, but also supports clients in other languages which aren't able to generate Protocol Buffer bindings.

5.1.2 Source Repository

The source repository for the Policy store can be found at the following GitHub URL:

https://github.com/DECODEproject/iotpolicystore

A binary build of the Policy store component can be downloaded from the releases page of the software repository:

https://github.com/DECODEproject/iotpolicystore/releases

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/DECODEproject/iotpolicystore/issues

5.1.3 Docker Repository

A Docker image containing the Policy store binary can be downloaded from the following Docker hub URL:

https://cloud.docker.com/u/thingful/repository/docker/thingful/policystore-amd64

At the time of writing the latest tagged version is: v0.1.0.

5.1.4 Dependencies

The server requires the following runtime dependencies:

- PostgreSQL (tested with PostgreSQL 10 but should work with all versions after 9)

5.1.5 Runtime Configuration

The binary generated for this application is called **`policystore`**. It has the following four subcommands:

- **`help`** - displays help information
- **`migrate`** - allows database migrations to be created and applied
- **`server`** - the primary command that starts up the server.

For operational use the **`server`** subcommand is the only one that is generally required.

Configuration for **`server`** subcommand

Flag	Environment variable	Description	Default value	Required
--addr or -a	POLICYSTORE_ADDR	The address and port to which the server binds	0.0.0.0:8082	No
--cert-file or -c	POLICYSTORE_CERT_FILE	Path to a TLS certificate file to enable TLS		No
--database-url or -d	POLICYSTORE_DATABASE_URL	URL at which Postgres is listening		Yes
--encryption-password	POLICYSTORE_ENCRYPTION_PASSWORD	Password used to encrypt secrets in the DB		Yes
--hashid-length or -l	POLICYSTORE_HASHID_LENGTH	Minimum length of generated IDs	8	No
--hashid-salt	POLICYSTORE_HASHID_SALT	Salt value used when generating IDs		Yes
--verbose		Flag that if present enables verbose logging	false	No

Flag	Environment variable	Description	Default value	Required
	SENTRY_DSN	Optional DSN string to enable Sentry error reporting		No

Table 1 - Policy store runtime configuration / flags

The above information is also available by running: **`policystore --help`**

5.1.6 API Client and Documentation

The protocol buffer definition for the Policy store service can be seen at the following URL:

https://github.com/thingful/decode-protorepo/blob/master/policystore/policystore.proto

This protocol buffer definition is the canonical source of truth that is used to define the server interface, and when using Go we are able to generate server stubs as well as a working client binding using this protocol buffer definition. A copy of the protocol buffer definition has also been included in appendix 8.2.1.

The Go client bindings for the server can be seen here:

https://github.com/thingful/twirp-policystore-go

API documentation for the Go client bindings can be seen here:

https://godoc.org/github.com/thingful/twirp-policystore-go

For clients that aren't able to generate bindings automatically you can view API documentation for the JSON/HTTP API the Policy store also exposes at the following URL:

https://decodeproject.github.io/iot-policystore-docs/

5.1.7 Software License

The component has been released under the terms of the GNU Affero GPL License v3.0. See https://github.com/DECODEproject/iotpolicystore/blob/master/LICENSE for details.

5.2 Encrypted Datastore

5.2.1 Description

As previously described in section 3.1.2 the Encrypted datastore is a simple component that provides a read/write API for storing encrypted data events. Data generators creators will use the write function, while consumers will read events which they will then be able to decrypt provided they have the correct decryption keys.

The primary interface offered by the Encrypted datastore is a Protocol Buffers over HTTP interface generated using Twirp (described in 4.1.6), however the server also exposes a JSON over HTTP interface that is callable from any standard HTTP client. This allows us to use the more performant Protocol Buffer interface where we can, but also supports clients in other languages which aren't able to generate Protocol Buffer bindings.

5.2.2 Source Repository

The source repository for the Encrypted datastore can be found at the following GitHub URL:

https://github.com/deCODEproject/iotstore

A binary build of the Encrypted datastore component can be downloaded from the releases page of the software repository:

https://github.com/DECODEproject/iotstore/releases

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/DECODEproject/iotstore/issues

5.2.3 Docker Repository

A Docker image containing the Encrypted datastore binary can be downloaded from the following Docker hub URL:

https://hub.docker.com/r/thingful/iotstore-amd64

At the time of writing the latest tagged version is: v0.3.10.

5.2.4 Dependencies

The server requires the following runtime dependencies:

- PostgreSQL (tested with PostgreSQL 10 but should work with all versions after 9)

5.2.5 Runtime Configuration

The binary generated for this application is called **`iotstore`**. It has the following four subcommands:

- **`delete`** – can be used to delete old data from the database (by whoever is operating the service)
- **`help`** - displays help information
- **`migrate`** - allows database migrations to be created and applied
- **`server`** - the primary command that starts up the server.

For operational use the **`server`** subcommand is the only one that is generally required.

Configuration for **`server`** subcommand

Flag	Environment variable	Description	Default value	Required
--addr or -a	IOTSTORE_ADDR	The address and port to which the server binds	0.0.0.0:8080	No
--cert-file or -c	IOTSTORE_CERT_FILE	The path to a TLS certificate file to enable TLS		No
--key-file or -k	IOTSTORE_KEY_FILE	The path to a TLS key file to enable TLS		No
--database-url or -d	IOTSTORE_DATABASE_URL	Connection string for a Postgres database		Yes
--verbose		Flag that if set enables verbose logging	false	No
	SENTRY_DSN	Optional DSN string to enable Sentry error reporting		No

Table 2 - Encrypted datastore runtime configuration / flags

5.2.6 API Documentation

The protocol buffer definition for the Encrypted datastore service can be seen at the following URL:

https://github.com/thingful/decode-protorepo/blob/master/datastore/datastore.proto

This protocol buffer definition is the canonical source of truth that is used to define the server interface, and when using Go we are able to generate server stubs as well as a working client binding using this protocol buffer definition. A copy of the protocol buffer definition has also been included in appendix 8.2.2.

The Go client bindings for the server can be seen here:

https://github.com/thingful/twirp-datastore-go

API documentation for the Go client bindings can be seen here:

https://godoc.org/github.com/thingful/twirp-datastore-go

For clients that aren't able to generate bindings automatically you can view API documentation for the JSON/HTTP API the Encrypted datastore also exposes at the following URL:

https://decodeproject.github.io/iot-datastore-docs/

5.2.7 Software License

The component has been released under the terms of the GNU Affero GPL License v3.0. See https://github.com/DECODEproject/iotstore/blob/master/LICENSE for details.

5.3 Stream Encoder

5.3.1 Description

As previously described in section 3.1.3 the Stream encoder is a component that provides a read/write API for creating encrypted streams. Device owners will be the entities that create or delete streams as it is this operation that is them exercising control over the destination of their data. The action of creating a stream causes the server to subscribe to SmartCitizen's MQTT broker then as events are received, they are processed and encrypted and written to the encrypted datastore.

The primary interface offered by the Encrypted datastore is a Protocol Buffers over HTTP interface generated using Twirp (described in 4.1.6), however the server also exposes a JSON over HTTP interface that is callable from any standard HTTP client. This allows us to use the more performant Protocol Buffer interface where we can, but also supports clients in other languages which aren't able to generate Protocol Buffer bindings.

5.3.2 Source Repository

The source repository for the Stream encoder can be found at the following GitHub URL:

https://github.com/DECODEproject/iotencoder

A binary build of the Stream encoder component can be downloaded from the releases page of the software repository:

https://github.com/DECODEproject/iotencoder/releases

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/DECODEproject/iotencoder/issues

5.3.3 Docker Repository

A Docker image containing the Encrypted datastore binary can be downloaded from the following Docker hub URL:

https://cloud.docker.com/u/thingful/repository/docker/thingful/iotenc-amd64

At the time of writing the latest tagged version is: v0.1.10.

5.3.4 Dependencies

The server requires the following runtime dependencies:

- PostgreSQL (tested with PostgreSQL 10 but should work with all versions after 9)
- Redis (tested with Redis 5, but should work with any version that includes sorted sets)
- Encrypted datastore (tested with the latest v0.3.9 build described above)

5.3.5 Runtime Configuration

The binary generated for this application is called **`iotenc`**. It has the following three subcommands:

- **`help`** - displays help information
- **`migrate`** - allows database migrations to be created and applied
- **`server`** - the primary command that starts up the server.

For operational use the **`server`** subcommand is the only one that is generally required.

Configuration for **`server`** subcommand

Flag	Environment variable	Description	Default value	Required
--addr or -a	IOTENCODER_ADDR	The address and port to which the server binds	0.0.0.0:8080	No
--broker-addr or -b	IOTENCODER_BROKER_ADDR	Address at which the MQTT broker is listening	tcp://mqtt.smartcitizen.me:1883	No
--cert-file or -c	IOTENCODER_CERT_FILE	The path to a TLS certificate file to enable TLS		No
--key-file or -k	IOTENCODER_KEY_FILE	The path to a TLS key file to enable TLS		No
--database-url	IOTENCODER_DATABASE_URL	Connection string for a Postgres database		Yes
--datastore or -d	IOTENCODER_DATASTORE	Address at which the datastore component is listening		Yes
--encryption-password	IOTENCODER_ENCRYPTION_PASSWORD	Password used to encrypt secrets we write to Postgres		Yes
--hashid-length or -l	IOTENCODER_HASHID_LENGTH	Minimum length of generated IDs for streams	8	No
--hashid-salt	IOTENCODER_HASHID_SALT	Salt value used when generating IDs for streams		Yes
--redis-url	IOTENCODER_REDIS_URL	URL at which		Yes

Flag	Environment variable	Description	Default value	Required
		Redis is listening		
--verbose		Flag that if set enables verbose logging	false	No
	SENTRY_DSN	Optional DSN string to enable Sentry error reporting		No

Table 3 - Stream encoder runtime configuration / flags

5.3.6 API Documentation

The protocol buffer definition for the Stream encoder service can be seen at the following URL:

https://github.com/thingful/decode-protorepo/blob/master/encoder/encoder.proto

This protocol buffer definition is the canonical source of truth that is used to define the server interface, and when using Go we are able to generate server stubs as well as a working client binding using this protocol buffer definition. A copy of the protocol buffer definition has also been included in appendix 8.2.3.

The Go client bindings for the server can be seen here:

https://github.com/thingful/twirp-encoder-go

API documentation for the Go client bindings can be seen here:

https://godoc.org/github.com/thingful/twirp-encoder-go

For clients that aren't able to generate bindings automatically you can view API documentation for the JSON/HTTP API the Encoder also exposes at the following URL:

https://decodeproject.github.io/iot-encoder-docs/

5.3.7 Software License

The component has been released under the terms of the GNU Affero GPL License v3.0. See https://github.com/DECODEproject/iotencoder/blob/master/LICENSE for details.

5.4 Zenroom-go

5.4.1 Description

To facilitate the usage of Zenroom from Go, we developed a very minimal Go wrapper which allows us to use the Zenroom binary easily from Go. It uses a feature of Go called CGO which allows Go to invoke functionality provided by C libraries directly.

Using CGO makes it slightly harder to compile and distribute binaries of your program, but the advantage of this is that we can instantly get access to all the functionalities provided by Zenroom.

5.4.2 Source Repository

The source for Zenroom-go can be found at the following GitHub URL:

https://github.com/DECODEproject/zenroom-go

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/DECODEproject/zenroom-go/issues

5.4.3 API Documentation

API documentation for Zenroom-go can be seen here:

https://godoc.org/github.com/DECODEproject/zenroom-go

5.4.4 Installation

In accordance with standard Go library installation practice the module can be installed by the standard **`go get`** command to install directly from GitHub.

```
$ go get github.com/DECODEproject/zenroom-go
```

Table 4 - zenroom-go installation command

5.4.5 Usage Example

An example of using the library is shown below:

```
package main

import (
  "fmt"
  "log"
```

```
  zenroom "github.com/DECODEproject/Zenroom-go"
)

func main() {
  script := `
-- define data schema
msg = SCHEMA.Record {
  msg = SCHEMA.String
}

-- read and validate the data
data = read_json(DATA, msg)

-- read keys without validating
keys = read_json(KEYS)

-- now import recipient public key
recipient_key = ECDH.new()
recipient_key:public(base64(keys.recipient_public))

-- now import our own private key (we are the data subject)
own = ECDH.new()
own:private(base64(keys.own_private))

-- encrypt the fields
out = {}
out = LAMBDA.map(data,function(k,v)
  header = MSG.pack({key=k, pubkey=own:public()})
  enc = ECDH.encrypt(own,recipient_key,str(v), header)
  oct = MSG.pack( map(enc,base64) )
  return str(oct):base64()
end)

-- print out result
print(JSON.encode(out))
`

  keys := `
{
  "own_private": "DYgWghvJuClxvHVCQSAfDpWQmeMQ4zh1/mGoNjM8UX0=",
```

```
  "own_public":
"BAXRFKfMNSMge11U/cP+mCW2al166qIAY/cETmToEGmqe+4JnMdhmJ1FURvtUU+gA4QiEP+C7QFy/eoH+FDSR
bw=",
  "recipient_public":
"BCj962CsLq0Ey9Ibe6DEFSak4KqnQ5FhbNMv7MaMr6OzZZsnncUVOTrFK4Ym9WItAEMbpkGIOIjgfPESblAKl
bw="
}
`

  data := `{"msg":"top secret"}`


  result := zenroom.Exec(script, zenroom.WithKeys(keys), zenroom.WithData(data))


  // Here we'd actually do something with the output
  fmt.Println(result)
}
```

Table 5 - zenroom-go usage example

5.5 Zenroom-py

5.5.1 Description

To facilitate the usage of Zenroom from Python, we developed a very minimal Python wrapper which just attempts to give the generated Zenroom binary a slightly more Pythonic API and make it a little easier to install.

5.5.2 Source Repository

The source for Zenroom-py can be found at the following GitHub URL:

https://github.com/DECODEproject/zenroom-py

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/DECODEproject/zenroom-py/issues

5.5.3 API Documentation

API documentation for Zenroom-py can be seen here:

https://zenroom-py.readthedocs.io/en/latest/

5.5.4 Installation

The library has been published as a package to the standard Python Package Index (https://pypi.org/), so can be installed via the following command:

```
$ pip install zenroom
```

Table 6 - zenroom-py installation command

The package is listed here: https://pypi.org/project/zenroom/

5.5.5 Usage Example

An example of using the library is shown below:

```
from zenroom import zenroom

script = b"""
-- define data schema
msg = SCHEMA.Record {
   msg = SCHEMA.String
}

-- read and validate the data
data = read_json(DATA, msg)

-- read keys without validating
keys = read_json(KEYS)

-- now import recipient public key
recipient_key = ECDH.new()
recipient_key:public(base64(keys.recipient_public))

-- now import our own private key (we are the data subject)
own = ECDH.new()
own:private(base64(keys.own_private))


-- encrypt the fields
out = {}
out = LAMBDA.map(data,function(k,v)
  header = MSG.pack({key=k, pubkey=own:public()})
```

```
  enc = ECDH.encrypt(own,recipient_key,str(v), header)
  oct = MSG.pack( map(enc,base64) )
  return str(oct):base64()
end)

-- print out result
print(JSON.encode(out))
"""


keys = b"""
{
  "own_private": "DYgWghvJuClxvHVCQSAfDpWQmeMQ4zh1/mGoNjM8UX0=",
  "own_public":
"BAXRFKfMNSMge11U/cP+mCW2al166qIAY/cETmToEGmqe+4JnMdhmJ1FURvtUU+gA4QiEP+C7QFy/eoH+FDSR
bw=",
  "recipient_public":
"BCj962CsLq0Ey9Ibe6DEFSak4KqnQ5FhbNMv7MaMr6OzZZsnncUVOTrFK4Ym9WItAEMbpkGIOIjgfPESblAKl
bw="
}
"""


data = b'{"msg":"top secret"}'


result = zenroom.execute(script, keys=keys, data=data)


# Here we'd actually do something with the output
print(result)
```

Table 7 - zenroom-py usage example

5.6 Python Datastore Client

5.6.1 Description

As stated in the section about Twirp, we are able to generate client bindings for languages from the Protocol Buffer definition provided a generator has been implemented for that language. Python is one of the supported languages for Twirp, so we were able to generate a client which will be used by the data collector in order to pull encrypted data from the datastore before decrypting it using Zenroom and storing the decrypted data into the dashboard's database.

5.6.2 Source Repository

The source code for the datastore client can be found at the following GitHub URL:

https://github.com/thingful/decode-datastore-client-py

Any issues related to the project can be reported via the issues page of the software repository:

https://github.com/thingful/decode-datastore-client-py/issues

5.6.3 Installation

The library has been published as a package to the standard Python Package Index (https://pypi.org/), so can be installed via the following command:

```
$ pip install decode-datastore-client
```

Table 8 - decode-datastore-client installation command

The package is listed here: https://pypi.org/project/decode-datastore-client/

5.6.4 Usage

An example of using the library is shown below:

```
from datetime import datetime, timedelta

from datastore_client.datastore_pb2 import ReadRequest

# we create a start_time of 1 hour ago
start_time = datetime.utcnow() - timedelta(hours=1)

# obtain the public key we are requesting data for
public_key = 'BGBgTKU7ZJHRBl...'

# construct our read request
read_request = ReadRequest()

# set the public key we are requesting data for
read_request.public_key = public_key

# set the start time from our datetime object
read_request.start_time.FromDatetime(start_time)
```

```
# set the page size we want to work with (max 1000)
read_request.page_size = 100

# now make the first request
response = client.read_data(read_request)

while True:
    for event in response.events:
        print(event.data) # encoded data requiring decryption
        print(event.event_time.ToJsonString())

    if response.next_page_cursor == "":
        break

    read_request.page_cursor = response.next_page_cursor
    response = client.read_data(read_request)
```

6 Conclusions

We feel the process of developing these components for the IoT pilot proceeded fairly successfully. The approach we took of defining a set of interfaces and sharing early with consortium partners allowed them to identify flaws or unnecessary complexities in the design that we have managed to eliminate.

Choosing to use Zenroom for all cryptographic work was also a great help in pushing this development forward. One of the key architectural principles of DECODE is to provide reusable components that can be composed in different ways. The IoT pilot is a solid example of this principle in practice, in that it allows a clean separation between the cryptographic expertise of certain partners within the consortium with the more mundane IoT component development and integration work of others. Zenroom provided this integration point between the two realms of expertise that allowed advanced cryptographic functions to be used by all parties with a minimal learning curve for all implementing developers.

6.1 Deployment

Work is just now starting on trying to get the components deployed and running in order to support the active state of the pilot as it moves from the development stage to a stage where the components are hopefully in active use. It will be interesting to see the reality of how the components interact together once we start seeing some significant flow of data between the components.

6.2 Future work

The original conception of the IoT pilot for DECODE was intended to make use of a distributed ledger implementation called Chainspace [17] created by other consortium partners to provide an immutable record proving that a data consumer had accepted the terms by which a data producer (the citizen) had agreed to share their data. This procedure aimed to invert the normal way that these interactions typically take place, i.e. rather than the user having to agree to the terms specified by a corporate entity, the corporate entity would have to accept the terms offered by the user. Once this agreement had been created and stored into the distributed ledger, the encoder would then query the ledger in order to read this state, and then create all specified encrypted streams until or unless the data producer chooses to revoke that agreement.

Precise details of how this ledger based interaction would work are yet to be finalised, but the basic idea is not that the chain based storage would provide the primary storage for the entitlement policies, or even the encrypted data, rather it would expose an API in the form of a smart contract that would allow both parties to record immutably their acceptance of one or more entitlement policies. The stream encoder would then use a query API to Chainspace (again in the form of a smart contract), to read this state from the ledger, and use this information to create all required encrypted streams for the user's data.

The availability of project resources allowing we hope to be able to implement this updated functionality within the project, however it's important to stress that adding the above described distributed ledger functionality would not change the end user experience in terms of how they will interact with the visible components, so we feel this current omission does not in any way damage the integrity of the pilot.

7 References

[1] "Vibrator maker ordered to pay out C$4m for tracking users' sexual activity," Guardian, 14th March 2017. [Online]. Available: https://www.theguardian.com/technology/2017/mar/14/we-vibe-vibrator-tracking-users-sexual-habits. [Accessed 13th January 2019].

[2] T. Hunt, "Data from connected CloudPets teddy bears leaked and ransomed, exposing kids' voice messages," 28th February 2017. [Online]. Available: https://www.troyhunt.com/data-from-connected-cloudpets-teddy-bears-leaked-and-ransomed-exposing-kids-voice-messages/. [Accessed 13th January 2019].

[3] W. Meers, "Hello Barbie, Goodbye Privacy? Hacker Raises Security Concerns," Huffington Post, 1st December 2015. [Online]. Available: https://www.huffingtonpost.com.au/entry/hello-barbie-security-concerns_us_565c4921e4b072e9d1c24d22. [Accessed 13th January 2019].

[4] A. Hern, "Fitness tracking app Strava gives away location of secret US army bases," Guardian, 28th January 2018. [Online]. Available: https://www.theguardian.com/world/2018/jan/28/fitness-tracking-app-gives-away-location-of-secret-us-army-bases. [Accessed 13th January 2019].

[5] W. Richter, "Our Dental Insurance Sent us "Free" Internet-Connected Toothbrushes. And this is What Happened Next," 14th April 2018. [Online]. Available: https://wolfstreet.com/2018/04/14/our-dental-insurance-sent-us-free-internet-connected-toothbrushes-and-this-is-what-happened-next/. [Accessed 13th January 2019].

[6] S. Biddle, "FOR OWNERS OF AMAZON'S RING SECURITY CAMERAS, STRANGERS MAY HAVE BEEN WATCHING TOO," The Intercept, 10th January 2019. [Online]. Available: https://theintercept.com/2019/01/10/amazon-ring-security-camera/. [Accessed 13th January 2019].

[7] "Docker," Docker Inc, 2018. [Online]. Available: https://www.docker.com/. [Accessed 13th January 2019].

[8] "Institut Municipal d'Informatica," IMI, [Online]. Available: http://ajuntament.barcelona.cat. [Accessed 16th January 2019].

[9] "Making Sense," Making Sense Consortium, [Online]. Available: https://making-sense.eu. [Accessed 12th January 2019].

[10] "Eurecat, Centre Tecnològic de Catalunya," [Online]. Available: https://eurecat.org/. [Accessed 24th January 2019].

[11] "Galois/Counter Mode," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Galois/Counter_Mode. [Accessed 15th January 2019].

[12] "Block cipher mode of operation," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. [Accessed 15th January 2019].

[13] "IRMA technical documentation," Privacy by Design Foundation, [Online]. Available: https://credentials.github.io/docs/irma.html. [Accessed 11th January 2019].

[14] "Go Programming Language," [Online]. Available: https://golang.org/. [Accessed 13th January 2019].

[15] "Dyne.org," Dyne.org, [Online]. Available: https://www.dyne.org/. [Accessed 12th January 2019].

[16] "Twirp Introduction," Twitch TV, [Online]. Available: https://twitchtv.github.io/twirp/docs/intro.html. [Accessed 13th January 2019].

[17] "Protocol Buffers," Google, [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed 13th January 2019].

[18] A. S. S. B. D. H. G. D. Mustafa Al-Bassam, "Chainspace: A Sharded Smart Contracts Platform," no. arXiv:1708.03778, 2017.

8 Appendices

8.1 Zenroom

Zenroom (https://zenroom.dyne.org/) is a deliberately small virtual machine created by Dyne [14] for fast cryptographic operations on Elliptic Curves. It has no external dependencies, includes a cutting-edge selection of C99 libraries and builds a small executable ready to run on: desktop, embedded, mobile, cloud and browsers (webassembly).

Rather than being a fully turing complete virtual machine, Zenroom only executes a scripting language called Zencode, which is a domain specific language (DSL) that exposes a compact set of cryptographic primitives and related operations whose design was informed by the pilot use cases for DECODE.

API documentation on Zenroom is available at the following URL:

https://zenroom.dyne.org/api

There is an online demo of Zenroom that is available here:

https://zenroom.dyne.org/demo

For more details on Zenroom please visit the website linked above.

8.1.1 Encryption Script

The final encryption script for the IoT pilot is included in Table 9 - Zencode encryption script.

```
-- Encryption script for DECODE IoT Pilot
curve = 'ed25519'

-- data schema to validate input
keys_schema = SCHEMA.Record {
  device_token     = SCHEMA.String,
  community_id     = SCHEMA.String,
  community_pubkey = SCHEMA.String
}

-- import and validate KEYS data
keys = read_json(KEYS, keys_schema)
```

```
-- generate a new device keypair every time
device_key = ECDH.keygen(curve)

-- read the payload we will encrypt
payload = {}
payload['data'] = DATA

-- The device's public key, community_id and the curve type are transmitted in
-- clear inside the header, which is authenticated AEAD
header = {}
header['device_pubkey'] = device_key:public():base64()
header['community_id'] = keys['community_id']

-- encrypt the data, and build our output object
output = ECDH.encrypt(
  device_key,
  base64(keys.community_pubkey),
  MSG.pack(payload), MSG.pack(header)
)

output = map(output, base64)
output.zenroom = VERSION
output.encoding = 'base64'
output.curve = curve

print(JSON.encode(output))
```

Table 9 - Zencode encryption script

8.1.2 Decryption Script

The corresponding decryption script for the IoT pilot is shown in Table 10 – Zencode decryption script.

```
-- Decryption script for DECODE IoT Pilot

-- data schemas
keys_schema = SCHEMA.Record {
  community_seckey = SCHEMA.String
}
```

```
data_schema = SCHEMA.Record {
  header   = SCHEMA.String,
  encoding = SCHEMA.String,
  text     = SCHEMA.String,
  curve    = SCHEMA.String,
  zenroom  = SCHEMA.String,
  checksum = SCHEMA.String,
  iv       = SCHEMA.String
}

-- read and validate data
keys = read_json(KEYS, keys_schema)
data = read_json(DATA, data_schema)

header = MSG.unpack(base64(data.header):str())

community_key = ECDH.new()
community_key:private(base64(keys.community_seckey))

payload, ck = ECDH.decrypt(
  community_key,
  base64(header.device_pubkey),
  map(data, base64)
)

print(JSON.encode(MSG.unpack(payload.text:str())))
```

Table 10 – Zencode decryption script

8.2 Protocol Buffer Definitions

8.2.1 Policystore Protobuf Definition

```
syntax = "proto3";

package decode.iot.policystore;
option go_package = "policystore";

// PolicyStore is a component that is responsible for maintaining a list of
```

```
// currently active data policies that are in use within the DECODE IoT pilot.
// It exposes an API by which clients can create or delete policies from the
// system, and importantly it exposes an API by which the DECODE wallet can
// retrieve a list of active policies which allows the wallet to then present a
// UI to the end user by which they will be able to choose which policies they
// wish to take part in.
service PolicyStore {
  // CreateEntitlementPolicy is a method exposed by the service which allows a
  // new entitlement policy to be created and stored within the device
  // registration service. Once a policy has been created, users will then be
  // able to apply this policy to their devices via the wallet.
  rpc CreateEntitlementPolicy (CreateEntitlementPolicyRequest) returns
(CreateEntitlementPolicyResponse);

  // DeleteEntitlementPolicy is a method exposed by the service which allows an
  // authorized client to request that an entitlement policy be deleted.
  // Deleting a policy will not affect any existing devices that have already
  // used the policy in order to create one or more streams within the encoder,
  // however it will prevent any new applications of that policy to other
  // devices.
  rpc DeleteEntitlementPolicy (DeleteEntitlementPolicyRequest) returns
(DeleteEntitlementPolicyResponse);

  // ListEntitlementPolicies is a method exposed by the service which returns a
  // list of all policies currently defined and available within the service to
  // be applied to devices. Currently it just returns a list of all known
  // policies with no capability to filter or paginate these policies.
  rpc ListEntitlementPolicies (ListEntitlementPoliciesRequest) returns
(ListEntitlementPoliciesResponse);
}



// Operation is a message used to describe an operation that may be applied to
// a specific data type published by a SmartCitizen device. The message contains
// two required fields: the sensor_id (this is the type of data we are entitling
// over), and a specified operation to be performed on that sensor type. This
// can be one of three actions: to share the sensor without modification, to
// apply a binning algorithm to the data so we output a bucketed value, or a
// moving average calculated dynamically for incoming values.
//
// If an operation specifies an Action type of `BIN`, then the optional
// `buckets` parameter is required, similarly if an action type of `MOVING_AVG`
```

```
// is specified, then `interval` is a required field.
message Operation {
  // The unique id of the sensor type for which this specific entitlement is
  // defined. This is a required field.
  uint32 sensor_id = 1; // required

  // An enumeration which allows us to specify what type of sharing is to be
  // defined for the specified sensor type. The default value is `SHARE` which
  // implies sharing the data at full resolution. If this type is specified, it
  // is an error if either of `buckets` or `interval` is also supplied.
  enum Action {
    UNKNOWN = 0;
    SHARE = 1;
    BIN = 2;
    MOVING_AVG = 3;
  }

  // The specific action this operation defines for the sensor type. This is a
  // required field.
  Action action = 2; // required

  // The bins attribute is used to specify the the bins into which incoming
  // values should be classified. Each element in the list is the upper
  // inclusive bound of a bin. The values submitted must be sorted in strictly
  // increasing order. There is no need to add a highest bin with +Inf bound, it
  // will be added implicitly. This field is optional unless an Action of `BIN`
  // has been requested, in which case it is required. It is an error to send
  // values for this attribute unless the value of Action is `BIN`.
  repeated double bins = 3;

  // This attribute is used to control the entitlement in the case for which we
  // have specified an action type representing a moving average. It represents
  // the interval in seconds over which the moving average should be calculated,
  // e.g. for a 15 minute moving average the value supplied here would be 900.
  // This field is optional unless an Action of `MOVING_AVG` has been specified,
  // in which case it is required. It is an error to send a value for this
  // attribute unless the value of Action is `MOVING_AVG`.
  uint32 interval = 4;
}

// CreateEntitlementPolicyRequest is a message sent to the policy registration
```

```
// service to create a new entitlement policy. An entitlement policy is a
// collection of one or more "Operations". A single Operation specifies an
// functional transformation to be performed on a single data channel being
// published by a SmartCitizen device. The policy as a whole is comprised of
// one or more Entitlements.
message CreateEntitlementPolicyRequest {
  // This attribute contains the public part of a key pair created by the
  // caller. The caller must keep the private key secret as this is will be
  // required for them to be able to decrypt data.
  string public_key = 1; // required

  // This attribute is used to attach a human friendly label to the policy
  // suitable for presenting to the end user in the DECODE wallet. This is a
  // required field.
  string label = 2; // required

  // The list of operations we wish to create for the policy. This field is
  // required, and it is required that the client supplies at least one
  // Operation.
  repeated Operation operations = 3; // required
}

// CreateEntitlementPolicyResponse is a message returned by the service after a
// policy has been created. The message simply contains an identifier for the
// policy, as well as a token that the caller must protect.
message CreateEntitlementPolicyResponse {
  // This attribute contains a unique identifier for the policy that can be used
  // for later requests to either apply a policy to a specific device, or to
  // delete the policy and so prevent new instances being applied to devices.
  string policy_id = 1;

  // This attribute contains a secret generated by the service that is
  // associated with the policy. This token is required to be presented by a
  // caller when deleting a policy, so must be treated as confidential by the
  // caller.
  string token = 2;
}

// DeleteEntitlementPolicyRequest is a message that can be sent to the
// registration service in order to delete an existing policy.
//
```

```
// Deleting a policy does not affect any already existing streams configured for
// the policy, it just stops any new instances of this policy being applied to
// other devices.
message DeleteEntitlementPolicyRequest {
  // This attribute contains the unique policy identifier returned when creating
  // the policy. This is a requiredi field.
  string policy_id = 1; // required

  // This attribute contains the token returned to the creator when they
  // created the policy, and must match the value stored within the
  // PolicyStore. This is a required field.
  string token = 2; // required
}

// DeleteEntitlementPolicyResponse is a placeholder response returned from a
// delete request. Currently empty, but reserved for any fields identified for
// future iterations.
message DeleteEntitlementPolicyResponse{
}

// ListEntitlementPoliciesRequest is the message sent to the service in order
// to receive a list of currently defined entitlement policies. Currently this
// message is empty as we simply return a list of all known policies, but this
// message may be extended should a need be identified to paginate through
// policies, or apply any search or filtering techniques.
message ListEntitlementPoliciesRequest {
}

// ListEntitlementPoliciesResponse is the response to the method call to list
// policies. It simply returns a list of all currently registered and
// non-deleted policies. This is intended to be able to be fed to the DECODE
// wallet in order to allow participant to choose which entitlements to apply to
// their devices.
message ListEntitlementPoliciesResponse {
  // Policy is a nested type used to be able to cleanly return a list of
  // Policies within a single response. Each Policy instance contains the id of
  // the policy, the list of entitlements defined by the policy, as well as the
  // policy's public key.
  message Policy {
    // This attribute contains the unique identifier of the policy.
    string policy_id = 1;
```

```
    // This attribute contains a human friendly label describing the policy
    // suitable for rendering in the DECODE wallet
    string label = 2;

    // This field contains a list of the operations that define the policy.
    repeated Operation operations = 3;

    // This attribute contains the public key of the policy. This public key
    // attribute is the label applied to the bucket within the datastore which
    // will be how data can be downloaded for the entitlement policy.
    string public_key = 4;
  }

  // This attribute contains the list of all policies currently available on
  // the device registration service.
  repeated Policy policies = 1;
}
```

Table 11 - Policystore protobuf definition

8.2.2 Datastore Protobuf Definition

```
syntax = "proto3";

package decode.iot.datastore;
option go_package = "datastore";

import "google/protobuf/timestamp.proto";

// Datastore is the interface we propose exposing to implement an encrypted
// datastore for the IOT scale model and pilot for DECODE. We expose two API
// methods, one to write and one to read data.
service Datastore {
  // WriteData is our function call that writes a single encrypted data event to
  // the underlying storage substrate. It takes a WriteRequest containing the
  // actual data to be stored along with public key of the bucket for which data
  // should be persisted and the submitting user's DECODE user id. THese
  // additional attributes allow us to request the data from the bucket by
  // public key.
```

```
  rpc WriteData (WriteRequest) returns (WriteResponse);

  // ReadData is used to request data from the data store. Data is requested
  // keyed by the public key used to encrypt it (encoded as a Base64 or hex
  // string probably). In addition a read request allows the client to specify a
  // time interval so that data is only retrieved if it was recorded within the
  // interval. Pagination is supported to allow for large intervals to be
  // requested without having to return all the data in one hit.
  rpc ReadData (ReadRequest) returns (ReadResponse);
}

// WriteRequest is the message that is sent to the store in order to write
// data. Data is written keyed by the public key of the recipient, the id of
// the user, as well as an id representing the entitlement policy. Finally the
// encrypted data is sent as a chunk of bytes.
message WriteRequest {
  // Reserve these attributes as they have been removed from a previous version
  // of this specification.
  reserved 1;
  reserved "public_key";

  // The data field here is the encrypted data to be stored for the specified
  // public key/entitlement policy. From the datastore's perspective this can
  // just be a slice of bytes, however zenroom does permit this data to
  // maintain some structure. From the datastores perspective however it treats
  // this data as a completely opaque bytes.
  bytes data = 2; // required

  // A token that uniquely identifies the device. This is a required field.
  string device_token = 3;

  // A string that uniquely identifies the policy for which data is being
  // written. A recipient will not be able to decrypt the data unless they are
  // in possession of valid credentials to decrypt this data. This is a
  // required field.
  string policy_id = 4;
}

// WriteResponse is a placeholder message returned from the call to write data
// to the store. Currently no fields have been identified, but keeping this as
// a separate type allows us to add fields as we identify them.
```

```
message WriteResponse {
}

// ReadRequest is the message that is sent to the store in order to read data
// for a specific bucket. When requesting data a client must submit the public
// key and entitlement policy id which identify the bucket, then optional start
// and end timestamps. If the time attributes are included then the end time
// must be after the start time; if no end time is specified then the default is
// "now". It is an error to specify an end time without a start time.
message ReadRequest {
  // Reserve these attributes as they have been removed from a previous version
  // of this specification.
  reserved 1;
  reserved "public_key";

  // The start time represents the start of an interval for which we wish to
  // read data. It is an error for start_time to be in the future or to be
  // after end_time. This field is required.
  google.protobuf.Timestamp start_time = 2; // required

  // The end time represents the end of an interval for which we wish to read
  // data. It may be nil, in which case it defaults to "now".
  google.protobuf.Timestamp end_time = 3;

  // The page cursor is an opaque string that an implementing server can
  // understand in order to efficiently paginate through events.  The value
  // sent here cannot be calculated by the client, rather they should just
  // inspect value returned from a previous call to to `ReadData` and if this a
  // non-empty string, then this value can be sent back to the server to get
  // the "next" page of results.  This field is optional.
  string page_cursor = 4;

  // The maximum number of encrypted events to return in the response. The
  // default value is 500. Returns an error if the caller requests a larger
  // page size than the maximum.
  uint32 page_size = 5;

  // A string that uniquely identifies the policy for which data is being
  // requested. A recipient will not be able to decrypt the data unless they
  // are in possession of the correct credentials. This is a required field.
  string policy_id = 6;
```

```
}

// EncryptedEvent is a message representing a single instance of encrypted data
// that is stored by the datastore. When reading data we return lists of this
// type, which comprise a timestamp and a chunk of encoded data. From the
// datastore's perspective the encrypted data can be viewed as just an opaque
// chunk of bytes, however our encoding engine (Zenroom), does allow us to just
// encrypt the values within a JSON structure, but for the datastore's purposes
// we don't care about this.
message EncryptedEvent {
  // The time at which the event was recorded by the datastore.
  google.protobuf.Timestamp event_time = 1;

  // The opaque chunk of bytes comprising the encoded data from the device.
  bytes data = 2;
}

// ReadResponse is the top level message returned by the read operations to the
// datastore. It contains the public key for the recipient, as well as the
// entitlement policy id. The events property contains a list of encrypted
// events in ascending time order. This will not necessarily be all possible
// events for the requested time period, as we have implemented pagination for
// this endpoint. If the response contains a non-empty string for the
// next_page_cursor property, then there are more pages of data to be consumed;
// if this property is the empty string, then the response is all data available
// for the requested time period.
message ReadResponse {
  // Reserve these attributes as they have been removed from a previous version
  // of this specification.
  reserved 1;
  reserved "public_key";

  // The list of encrypted events containing the actual data being requested.
  // This list will be returned in ascending time order, and each element
  // contains a timestamp as well as the actual chunk of encrypted data. If no
  // data is available this will be an empty list.
  repeated EncryptedEvent events = 2;

  // An optional field containing a pointer to the next page of results
  // expressed as an opaque string. Clients should not expect to be able to
  // parse this string as its contents are strictly implementation specific and
```

```
  // subject to change at any time. Rather the value here should just be checked
  // to see if it is an empty string or contains a value, and if any value is
  // present, the client can pass it back in a new read request as the value of
  // the page_cursor field.
  string next_page_cursor = 3;

  // The page size that was originally requested to create this response.
  // Supplied to make it easy for the client to construct a new request for the
  // next page.
  uint32 page_size = 4;

  // A string that uniquely identifies the policy for which data is being
  // sent. A recipient will not be able to decrypt the data unless they
  // are in possession of the correct credentials.
  string policy_id = 5;
}
```

Table 12 - Encrypted datastore protobuf definition

8.2.3 Stream Encoder Protobuf Definition

```
syntax = "proto3";

package decode.iot.encoder;
option go_package = "encoder";

// Encoder is the basic interface proposed for the stream encoder component for
// DECODE. It currently just exposes two methods which allow for encoded streams
// to be created and destroyed. Creating a stream means setting up a
// subscription to an MQTT broker such that we start receiving events for a
// specific device. These events are then encrypted using the supplied
// credentials, and then written upstream to our encrypted datastore. Once a
// stream has been created it continues running indefinitely until receiving a
// call to delete the stream.
//
// Later iterations of this service will implement filtering and aggregation
// operations on the stream, but for now all data is simply passed through to
// the datastore.
service Encoder {
  // CreateStream sets up a new encoded stream for the encoder. Here we
```

```
  // subscribe to the specified MQTT topic, save the encryption keys, and start
  // listening for events. On receiving incoming messages via the MQTT broker,
  // we encrypt the contents using Zenroom and then write the encrypted data to
  // the configured datastore.
  rpc CreateStream (CreateStreamRequest) returns (CreateStreamResponse);

  // DeleteStream is called to remove the configuration for an encoded data
  // stream. This means deleting the MQTT subscription and removing all saved
  // credentials.
  rpc DeleteStream (DeleteStreamRequest) returns (DeleteStreamResponse);
}

// CreateStreamRequest is the message sent in order to create a new encoded
// stream. As a result of this method call, the stream encoder will have
// configured a stream that receives messages, applies all defined entitlement
// operations, then encrypts the data and sends it on to the configured
// datastore.
message CreateStreamRequest {
  // The token that uniquely identifies the device. This is a required field.
  string device_token = 1;

  // A unique identifier for the specific community represented by the policy
  // being applied.
  string policy_id = 2;

  // The public key of the recipient, again this is used in order to encrypt
  // outgoing data, as well as being used to signify to the datastore the bucket
  // in which data should be stored. This is a required field.
  string recipient_public_key = 3;

  // A nested type capturing the location of the device expressed via decimal
  // long/lat pair.
  message Location {
    // The longitude expressed as a decimal.
    double longitude = 1;

    // The latitude expressed as a decimal.
    double latitude = 2;
  }

  // The location of the device to be claimed.
```

```
  Location location = 5;

  // An enumeration which allows us to express whether the device will be
  // located indoors or outdoors when deployed.
  enum Exposure {
    UNKNOWN = 0;
    INDOOR = 1;
    OUTDOOR = 2;
  }

  // The specific exposure of the device, i.e. is this instance indoors or
  // outdoors.
  Exposure exposure = 6;

  // A nested type which is used to capture a list of specific operations we
  // perform the stream.
  message Operation {
    // The unique id of the sensor type for which this specific configuration is
    // defined. This is a required field.
    uint32 sensor_id = 1;

    // An enumeration which allows us to specify what type of sharing is to be
    // defined for the specified sensor type. The default value is `SHARE` which
    // implies sharing the data at full resolution. If this type is specified, it
    // is an error if either of `buckets` or `interval` is also supplied.
    enum Action {
      UNKNOWN = 0;
      SHARE = 1;
      BIN = 2;
      MOVING_AVG = 3;
    }

    // The specific action this entitlement defines for the sensor type. This is a
    // required field.
    Action action = 2;

    // The bins attribute is used to specify the the bins into which incoming
    // values should be classified. Each element in the list is the upper
    // inclusive bound of a bin. The values submitted must be sorted in strictly
    // increasing order. There is no need to add a highest bin with +Inf bound, it
    // will be added implicitly. This field is optional unless an Action of `BIN`
```

```
    // has been requested, in which case it is required. It is an error to send
    // values for this attribute unless the value of Action is `BIN`.
    repeated double bins = 3;

    // This attribute is used to control the entitlement in the case for which we
    // have specified an action type representing a moving average. It represents
    // the interval in seconds over which the moving average should be calculated,
    // e.g. for a 15 minute moving average the value supplied here would be 900.
    // This field is optional unless an Action of `MOVING_AVG` has been specified,
    // in which case it is required. It is an error to send a value for this
    // attribute unless the value of Action is `MOVING_AVG`.
    uint32 interval = 4;
  }

  // The entitlements field holds a repeated list of Operations which each
  // define a transformational function for a specific sensor id. If no
  // operations are submitted, we currently create a stream that writes
  // through all received channels without applying any processing transformations
  // to the data, but if this field contains any elements, the resulting stream
  // will only contain the specified sensor type.
  repeated Operation operations = 7;
}

// CreateStreamResponse is the message returned from the stream encoder after it
// successfully creates a stream. The device registration service should keep a
// record of this value so that it is able to delete the stream if required.
message CreateStreamResponse {
  // An identifier for the stream which can be used in order to delete a stream
  // when required.
  string stream_uid = 1;

  // A secret token passed back to the caller which it must keep secret, in
  // order to be permitted to delete the stream.
  string token = 2;
}

// DeleteStreamRequest is the message sent to the encoder in order to delete a
// configured stream. Sending this message must delete the MQTT subscription, as
// well as deleting all encryption credentials stored on the encoder.
message DeleteStreamRequest {
  // The identifier for the stream to be deleted. This is a required field.
```

```
  string stream_uid = 1; // required

  // The secret token that was returned to the caller when creating the stream.
  // This is a required field, and must match the value stored internally for
  // the stream.
  string token = 2; // required
}

// DeleteStreamResponse is a placeholder response message on a successful
// deletion of stream on the encoder.
message DeleteStreamResponse {
}
```

Table 13 – Stream encoder protobuf definition