TagItSmart
Smart tags driven service platform for enabling ecosystems of connected objects

Grand agreement 688061

Integrated Extension Prototype

Deliverable ID:	D1.2
Deliverable Title:	Integrated Extension Prototype
Project acronym:	TagItSmart!
Project title:	Smart Tags driven service platform for enabling ecosystems of connected objects
Extension acronym:	TagItLedger
Extension title:	Tag Driven Supply Chain Ledger Service for Enhanced Product Transparency
Project website address:	http://tagitsmart.eu/

European Commission

Contents

1 Introduction ***5***

2 TagItLedger Architecture ***6***

2.1 Nxt Blockchain **6**
2.1.1 Private Blockchain Creation 6
2.1.2 Docker Compose Approach 7
2.1.3 Nxt Native Client 8

2.2 EVRYTHNG Platform **9**
2.2.1 MQTT Pub/Sub 9
2.2.2 EVT Virtual Entity 10

2.3 TagItLedger Middleware **10**

3 TagItLedger Extension Implementation ***12***

3.1 Scanning phase simulation **12**

3.2 TagItLedger Android Application **12**

3.3 Application Programming Interface (API) **14**

4 TagItLedger in the frame of TagItSmart Use Cases ***15***

4.1 A Digital Product/Lifecycle Management **15**
4.1.1 Usage Example 17

4.2 Dynamic Pricing **18**
4.2.1 Usage Example 18

5 Conclusions ***21***

6 References ***22***

7 Appendix A ***23***

8 Appendix B ***26***

European Commission

Figures

Figure 1 - Nxt Client UI 9
Figure 2 - TagItLedger Architecture 10
Figure 3 - TagItLedger App UI 13
Figure 4 - Updated scanning flowchart for consumer 16
Figure 5 - Asset Issue Transaction 17
Figure 6 - Asset ID 17
Figure 7 - Nxt property with asset id 17
Figure 8 - Asset Transfer 17
Figure 9 - Transaction info 18
Figure 10 - Notification Message 19
Figure 11 - The new transaction in the NXT blockchain 19
Figure 12 - Transaction Info 20
Figure 13 - Transaction Data Content 20

Tables

Table 1 - Nxt API calls 14

1 Introduction

This document is the second deliverable of the TagItLedger project. TagItLedger is a Category B (new extensions - modules) deliverable aiming to extend TagItSmart platform functionality with components related to transactional validation and traceability capabilities through blockchain functionalities in TagItSmart product lifecycle. This deliverable D1.2 of TagItLedger project reports the integrated extension prototype and is an extension of D1.1. The major goal is to produce the extension prototype of the D1.1 design as well as interconnection modules with the TagItSmart platform and relevant software documentation.

The rest of this deliverable is organised as follows:

- Section 2 focuses on the high-level TagItLedger architecture, on how to integrate TagItSmart components with the TagItLedger module and on how this architecture is implemented.
- Section 3 outlines the extension implementation of TagItLedger, as well as the used blockchain API.
- Section 4 gives a general overview of the TagItLedger module in the frame of TagItSmart use cases, along with examples.
- Finally, Section 5 concludes the deliverable.

Last but not least, in the end of this document we provide two Appendixes with the Docker Compose and Android application source code snippets, as well as with the relevant explanation.

2 TagItLedger Architecture

In this section, the final TagItLedger architecture is presented starting from a high-level perspective explaining components and services. TagItLedger is based on top of TagItSmart architecture. TagItLedger extends the TagItSmart architecture by adding blockchain functionality that will enhance the system's ability and intelligence to identify and detect cases of fraudulent behavior as well as indirectly keep an overall history record of the product, enhancing transparency towards the end user. More specifically, the section describes in detail, the architecture, components and the process flow of the implementation. EVRYTHNG platform and Nxt blockchain will be used in order to integrate with the respective platform for obtaining scanned information. Third parties can be use the Nxt blockchain by firing up Nxt nodes, following the instructions provided in the current document. As already pointed out in D1.1, TagItLedger middleware architecture, consist of two major components:

- NXT blockchain and
- EVRYTHNG platform

2.1 Nxt Blockchain

2.1.1 Private Blockchain Creation

Nxt provides a creation kit [kit2018] that allows blockchain developers to create and launch their own private blockchain platform, which is an easy and fast approach to launch a new blockchain as service (BaaS). This package [kit2018] is intended to allow easy creation of new blockchain projects based on Nxt, satisfying the requirements of the Jelurida Public License version 1.1 for the Nxt Public Blockchain Platform, focusing on developers and not to the end users. In order to read/write in the blockchain a Nxt account is necessary. Any entity can be considered as the asset of the blockchain, mapping to the concept and exploiting the functionality, and in this particular case the granularity should be placed on the product instance ID level. However, for such an approach to minimize user involvement (and increase self-management from the system side), one needs to map and unify data notifications coming from the products side (such as product classes and Thng catalogue coming from EVRYTHNG, information coming from the scanning process in TagItSmart) to the typical blockchain management concepts. In this subsection we will provide instructions and the followed steps on how we created our private Nxt blockchain.

To actually start a new blockchain project using this kit the Nxt clone creation kit, we have first to create a genesis block. More specifically, the genesis block parameters and accounts must be defined in the genesisParameters.json and genesisAccounts.json files respectively. The number of created nodes is independent from the number of participants, however the number of Nxt accounts should be the same number of the participants (minus the end user)

to be able to perform transactions. In our scenarios the number of stakeholders is known from the beginning, however this is easily extensible since new accounts can be added to the blockchain anytime is needed on demand. This can give access also to the consumers with the condition to create first an account.

- The genesisParameters.json file defines the timestamp of the genesis block for the new blockchain, and the public key of the genesis account.

```
{
"genesisPublicKey":"1259ec21d31a30898d7cd1609f80d9668b4778e3d97e941044b39f0c44d2e51b",
  "epochBeginning":"2017-05-24 12:00:00 +0000"
}
```

genesisParameters.json

- The genesisAccounts.json file reason contains contain the list of accounts to be created in the genesis block of the new blockchain, and their corresponding balances and public keys.

```
{
   "balances": {
      "NXT-NZKH-MZRE-2CTT-98NPZ": 30000000000000000,
      "NXT-X5JH-TJKJ-DVGC-5T2V8": 30000000000000000,
      "NXT-LTR8-GMHB-YG56-4NWSE": 30000000000000000
    },
    "publicKeys": [
      "bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37",
      "39dc2e813bb45ff063a376e316b10cd0addd7306555ca0dd2890194d37960152",
      "011889a0988ccbed7f488878c62c020587de23ebbbae9ba56dd67fd9f432f808"
    ]
}
```

genesisAccounts.json

Last but not least, the blockchain service is particularly tailored for containerized environments such as Docker, as well as any other container format adopting the Open Container Specification.

2.1.2 Docker Compose Approach

TagItLedger project follows the microservice architectural paradigm. Such a paradigm commands the decomposition of a single service into a set of services, which are

independent but inter-communicate. The more decentralized the blockchain becomes, the greater the immutability of its records. Client management will be based on typical Dockerized micro-service rationale. Docker will help to deploy fast and easy as many Nxt nodes. Docker Compose is used towards supporting this approach. Docker Compose is a tool for defining and running multi-container Docker applications. A YAML file is used to configure the blockchain nodes and only a single command is needed in order to create and start the Nxt nodes.

A two-step process is needed [unicornd4.1]. First, the environment of the application is defined in a Dockerfile. This file is a text file with a list of steps to perform in order to create an image. Configuration of the operating system, installation of the Nxt node or coping of the files to the right place could be included in the aforementioned steps. When the Dockerfile is built an image is created which is a template for creating the environment for the application. A running instance of this image constitutes the container where the Nxt node is deployed. In the second step a docker-compose.yml file is created, where the Nxt node orchestration is defined. In that way the services can be run together in an isolated environment. Snippets of the Dockerfile used for the compile and deployment of Nxt node and the docker-compose.yml file along with the descriptions of their contents can be found in Appendix A.

2.1.3 Nxt Native Client

After the blockchain creation, the stakeholders will be able to access the blockchain, through the Nxt native client. Each stakeholder will have a private Nxt account, where he will be able to see all the transactions and the assets he owns. Currently we have created three accounts for demo purposes in the genesis block, but it easily to add new accounts from the Nxt native client UI, where can be supported as many accounts as need on demand after the creation of the genesis block. Figure 1, below shows the Nxt client UI.

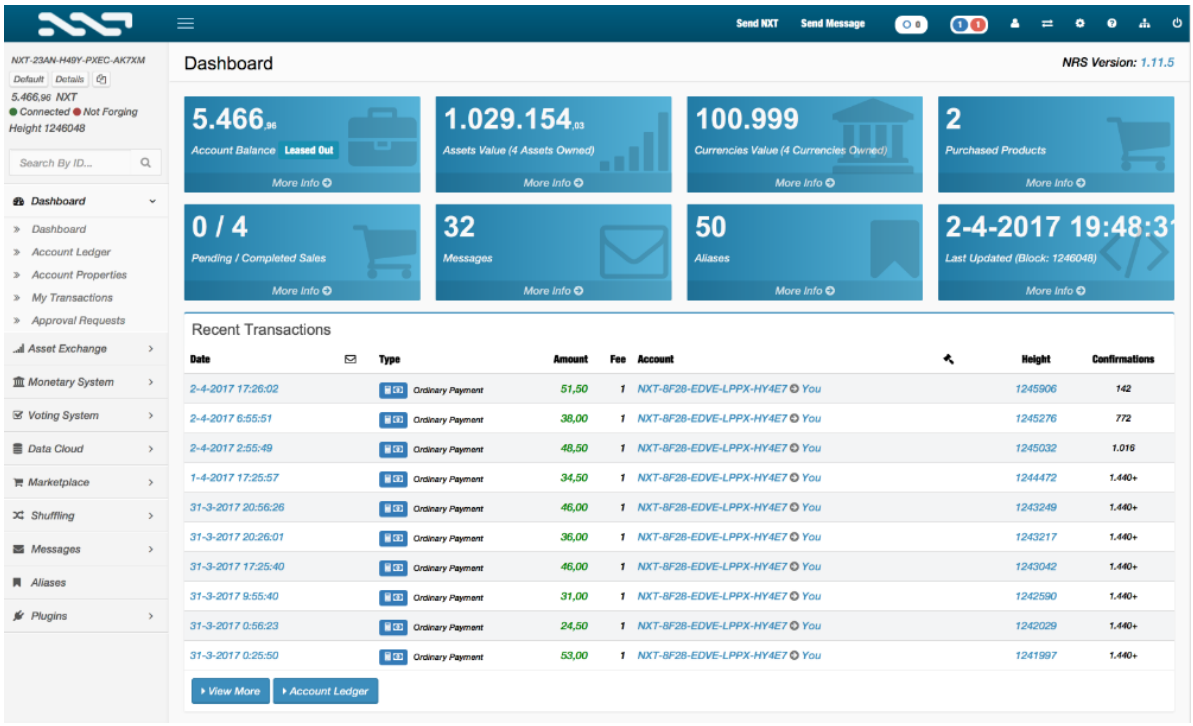

Figure 1 - Nxt Client UI

2.2 EVRYTHNG Platform

2.2.1 MQTT Pub/Sub

TagItSmart is deployed on EVRYTHNG platform, where the tags are registered and all tags get their virtual entity from here. The virtual entity validates the tags existence and is used throughout the process. EVRYTHNG platform supports Message Queuing Telemetry Transport (MQTT) protocol. MQTT is a publish/subscribe messaging protocol for constrained Internet of Things (IoT) devices and low-bandwidth, high-latency or unreliable networks. Because MQTT specializes in low-bandwidth, high-latency environments, it is an ideal protocol for machine-to-machine (M2M) communication. All MQTT topics and the JSON payloads of subscription or publication messages are exactly the same as EVRYTHNG's REST API, allowing easily to swap between the protocols. However, for TagItLedger architecture, MQTT protocol is used due to performance reasons for IoT devices.

In order to check EVT's MQTT Pub/Sub, MQTTBox has been used for data simulation. MQTTBox can be used to create MQTT clients to publish or subscript topics, create MQTT virtual device networks, load test MQTT devices or brokers and much more. More specifically, MQTTBox is a cross platform application available on Linux, MAC and Windows [mqttbox].

2.2.2 EVT Virtual Entity

This service provides the interface to work with the virtual entities, registration and query, as well consulting the available models. Also, there are API endpoints to create Virtual Entities that have SmartTags associated. EVRYTHNG platform provides backend support with free access to their IoT Smart Products Platform. In terms of TagItLedger, we utilize EVRYTHNG's Virtual Entity service and store the occurred value changes as a blockchain transaction.

2.3 TagItLedger Middleware

Figure 2, depicts the TagItLedger middleware architecture. The mobile devices at the left represent TagItSmart users that scan a product and publish events to the MQTT broker. By utilizing Pub/Sub stream processing of EVRYTHNG platform, TagItLedger can subscribe (TagItLedger Subscriber) to retrieve the relevant events such as Thng changes. The TagItLedger Subscriber, pushes the retrieved events to the blockchain through Nxt's REST API. Data stored to the blockchain could be the contextual data retrieved from the scanning process, such as geolocation, device type, user ID, time of scanning process, or even the QR code image. More specifically, we leverage the emerging Nxt blockchain to build a decentralized, trusted, transparent and open architecture for TagItSmart, on top of the largely adopted Pub/Sub infrastructure. In addition, Nxt blockchain clients/stakeholders will be able to connect the blockchain to validate the transactions.

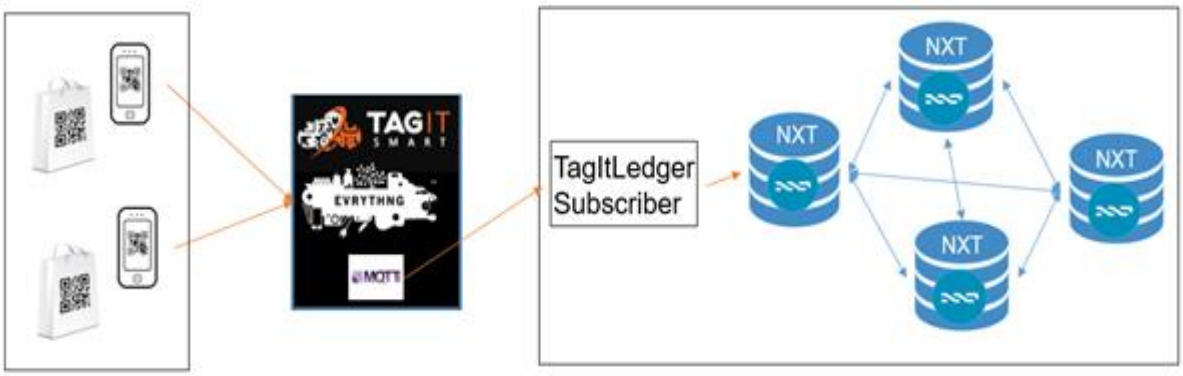

Figure 2 - TagItLedger Architecture

Publishers and subscribers communicate to MQTT Broker of EVRYTHNG platform, which intercept the operations and transmit them to NXT blockchain, while operations are recorded in the blockchain transparently and abstract to the TagItSmart participants. A direct link between the Pub/Sub Broker, from which the basic information regarding a product and its cycle will be acquired and feed the blockchain oriented functional components. The latter will subscribe to the aforementioned systems and automatically translate the information to

additions in the chain, thus enhancing the blockchain abstraction and extending the self-organization aspect of the system. TagItLedger blockchain creation will be directly integrated in TagItSmart, while the blockchain technology itself is asset agnostic and applied to a multiplicity of application domains.

3 TagItLedger Extension Implementation

In this section, we provide details on the simulation and implementation of TagItLedger middleware.

3.1 Scanning phase simulation

As already pointed out, in TagItLedger module we assume that the scanning phase has already taken place. For that reason, we simulate such phase in order to continue with the TagItLedger part. More specifically, MQTTBox has been used for data simulation. MQTTBox can be used to create MQTT clients to publish or subscript topics, create MQTT virtual device networks, load test MQTT devices or brokers and much more. More specifically, MQTTBox is a cross platform application available on Linux, MAC and Windows [mqttbox]. In addition, various scan scenarios based on the use cases have added (for simulation reason) as a feature in the implemented Android application.

3.2 TagItLedger Android Application

In this subsection, we will analyze the TagItLedger Android application. The three main functionalities of TagItLedger application are:

- Subscription to EVRYTHNG's Pub/Sub broker
- Receive notification on value changed
- Create a new transaction in the blockchain
    - Issue Asset
    - Transfer Asset
    - Upload Data

The created Android application is written in Java language. Various source code snippets and their description can be found in Appendix B. The application designed in a way to be easily adapted to different scenarios. Moreover, the source code, as well as the docker compose file for the blockchain are provided in this repository (https://github.com/sofiannasof/TagItLedger). For the implementation, we used the Model-View-Presenter (MVP) pattern which is known pattern for Android development. In MVP, the Presenter contains the UI business logic for the View. All invocations from the View delegate directly to Presenter. The Presenter is also decoupled directly from the View and talks to it through an interface. This is to allow mocking of the View in a unit test. One common attribute of MVP is that there has to be a lot of two-way dispatching. An example usage of the application is found is in Figure 3. The screen shows a recycler view where all the scanned actions are listed, while on top there is a menu where we can select which scenario we want to run. The fab button on the bottom, can be used to simulate the scan data phase.

TagItLedger middleware is represented by the implemented Android application, however this is not restrictive. The same idea can also implement as a backend application, that receives all the scanned information and then forwards that information to the blockchain. We chose to implement an Android application as the TagItLedger middleware, for possible seamless integration with the existing TagItSmart mobile application. In order to use the mobile app a blockchain account is necessary. In addition, in order to avoid multiple forwarded data to the blockchain from the multiple instances of the TagItLedger application, we selected to forward only data from the device that scans the product, in our case the simulated scanned data. All devices, will receive the relevant subscribed notifications upon changes, however only the device that scanned the product will forward the data to the blockchain upon message received.

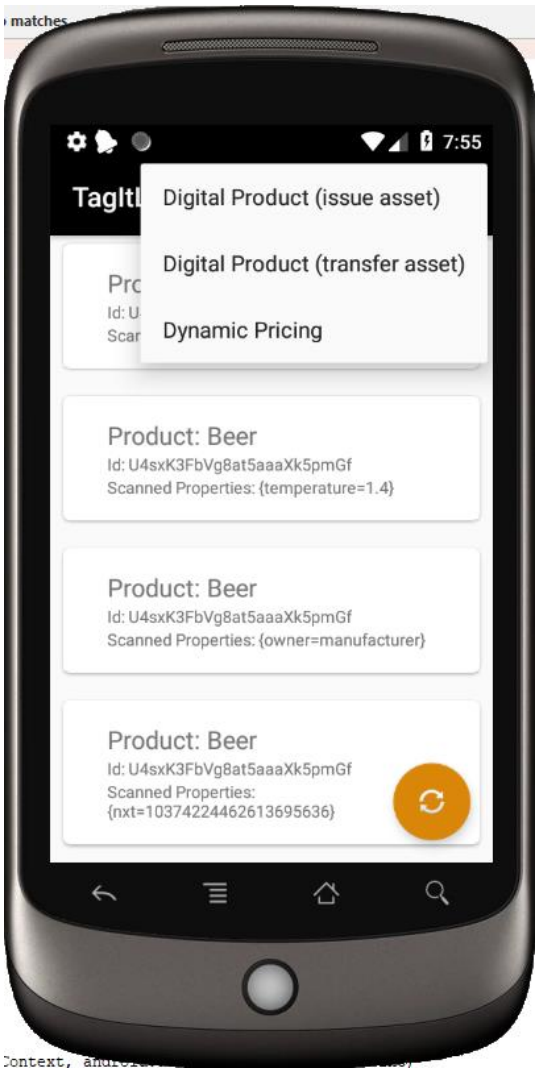

Figure 3 - TagItLedger App UI

3.3 Application Programming Interface (API)

In this subsection the reference APIs of blockchain TagItLedger is presented providing a uniform documentation on designed functionalities to be used by other authorized domains, architectures and developers. Developers need to be able to create applications and connect with the Nxt blockchain API with little effort. This reference APIs provide the foundations for the implementation of each component for the use cases in the underlying IoT platforms. These APIs are provided here as a short version. A full specification is available in the official Nxt blockchain API [nxtapi]. Table 1 below list the API calls used to interact with Nxt blockchain for the use cases.

Table 1 - Nxt API calls

Operation	Description
POST /nxt? requestType=issueAsset& publicKey=bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37& name=U4sxK3FbVg8at5aaaXk5pmGf& description=This+is+DigitalBeer& quantityQNT=100& deadline=60& feeNQT=1& broad=false	Issues an asset with the name of EVRYTHNG thng ID. After the transaction, the TagItLedger App, publish a new EVRYTHNG property with the NXT asset ID.
POST /nxt? requestType=transferAsset& secretPhrase=0& recipient=NXT-X5JH-TJKJ-DVGC-5T2V8& asset=U4sxK3FbVg8at5aaaXk5pmGf& quantityQNT=1& feeNQT=1& deadline=60	Transfer an asset with the name of EVT thng ID to the next stakeholder in the lifecycle
POST /nxt? requestType=uploadTaggedData& data=”[{"value": 1.1}]”& name=Digital Beer publicKey=bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37& description=This+is+DigitalBeer& tags=product, beer& secretPhrase=0& deadline=60& feeNQT=100000000&	Upload raw data in the blockchain, easily accessible through a tag

4 TagItLedger in the frame of TagItSmart Use Cases

This section specifies the application of TagItLedger to TagItSmart use cases and to show how the use cases are extended with a blockchain as a service. The use cases that can capture the concepts of TagItLedger are the digital product, the lifecycle management and dynamic pricing. The digital product use case is a kind of umbrella use case including the whole value chain from manufacturer to transport, retail, consumer and recycling, while the other cases concentrate more in details to one or two stages of the lifecycle of the product. The use cases are described more in detail in the following subsections, focusing on the TagItLedger incorporation, in order to define the usefulness of the blockchain service.

In all the use cases we omitted to present the details regarding how the end-user can have access to the blockchain since it is the last owner and it is not necessary. However, if needed the end-user can easily create an account to the private blockchain and access all the necessary information throw the web-based UI provided from the Nxt blockchain. Moreover, PoS algorithms like the one used in Nxt blockchain in general improve network scalability. Nxt network back in 2013 could process up to 367,200 transactions per day [nxtwhite], however some more recent results achieved a scale up to 100 transactions per second in a private blockchain [results], which is a good improvement, considering applications with many clients such as the TagItLedger middleware. Unfortunately, a large-scale stress testing is time and resource consuming and we didn't manage to include it in this deliverable in order to have a more realistic results regarding the blockchain scalability.

4.1 A Digital Product/Lifecycle Management

A digital beer is considered as the digital product to be mapped as an asset in the blockchain. The product is labeled with a smart tag containing a sensor with reversible thermochromic ink. The first use of the sensor and the smart tag is during the production of the product itself, where the sensor indicates that the temperature of the product is below a certain temperature. When used by the retailer of consumers, the sensor data values are sent to the server for storage and further processing. More specifically, when a consumer scans the code on a bottle, a message based on the backend response about the temperature of the bottle, information about the beer and when and a timeline of the items life. The timeline has all events that have happened to the bottle so far: created, added to storage, and sent to retail [D5.3].

In terms of TagItLedger, we assume that the scanning phase has already taken place and through the Pub/Sub broker the TagItLedger app will get access to the virtual entity and store it on the Nxt blockchain. Moreover, we will focus on the mapping of Evrythng virtual entity to blockchain asset during product lifecycle. The manufacturer will be able to track the "owner" of the asset. This use case can easily be extended to the lifecycle management use case, in the same way.

When a consumer scans a code on a bottle, the reader application sends the information to the backend. The backend will use the data matrix code to resolve the correct virtual entity from Evrythng platform. Sensor information from the virtual entity is used to interpret the sensor reading. TagItLedger App, subscribed to the Pub/Sub broker, notified by the scanned value and creates a new transaction on the blockchain. The first time the manufacturer will be able to issue a blockchain asset, while the ownership of the asset can be transferred upon further scanning. The box code identifier is used to find the sales order information for the bottle. The consumer will be shown a message based on the backend response about the temperature of the bottle, information about the beer and when and a timeline of the items life. More specifically, when an asset is transferred from stakeholder A to stakeholder B, not only A has to issue the “transfer asset” command, but also B has to accept the asset transfer. Actually, the acceptance process will be triggered when B scans the product. When the new ownership confirmed the current scanned data are pushed to the blockchain. The timeline has all events that have happened to the bottle so far: created, added to storage, and sent to retail. Figure 3, summarises the aforementioned flow for the consumer.

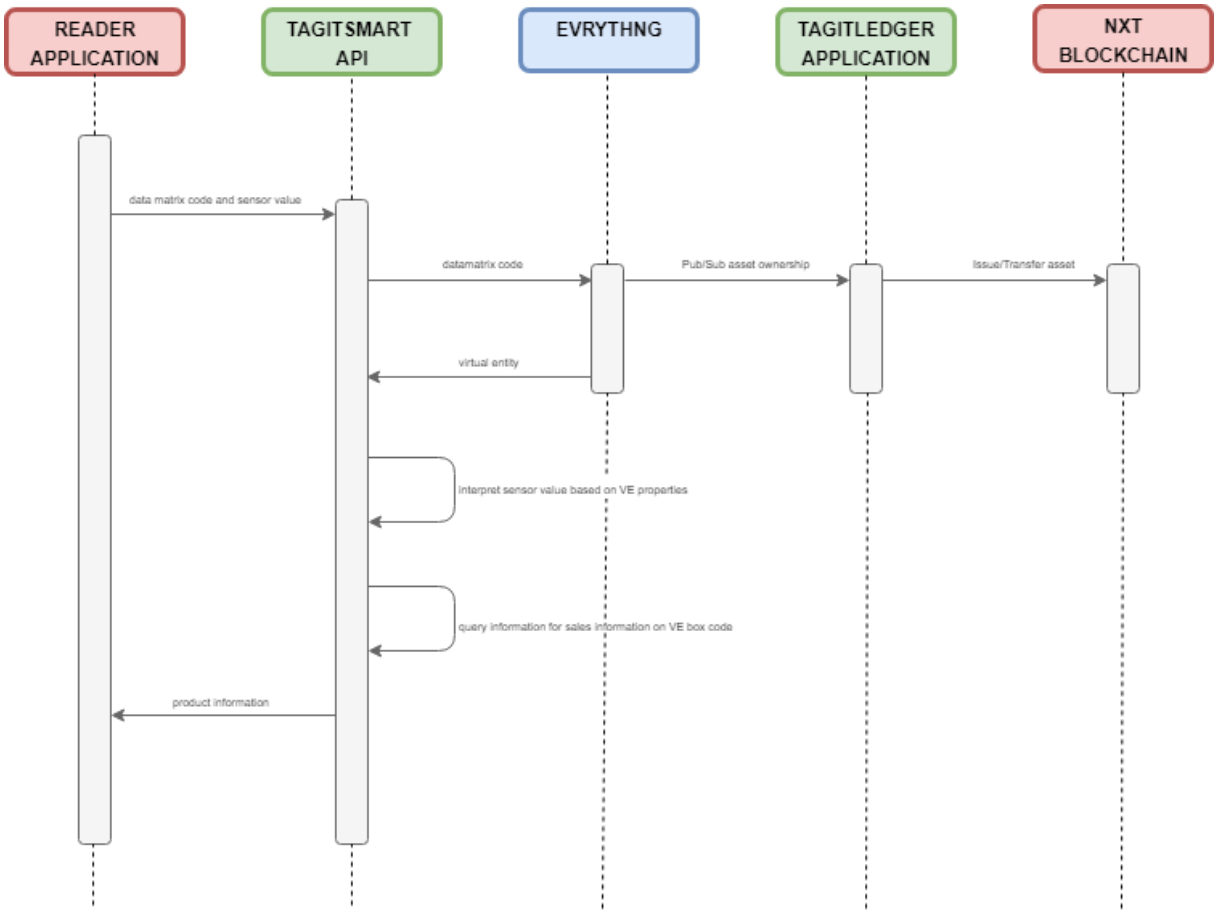

Figure 4 - Updated scanning flowchart for consumer

4.1.1 Usage Example

In this example, we simulate the first part of the scanned data and we notify the app user with the new data value through the EVT Pub/Sub. More specifically, the first time the manufacturer upon scanning issues an asset on the blockchain. The name of the asset should match the name of the thng virtual entity. Figures below shows the transaction of issuing a new asset and the new created asset id.

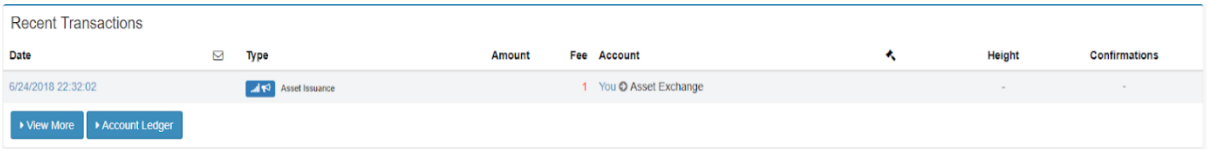

Figure 5 - Asset Issue Transaction

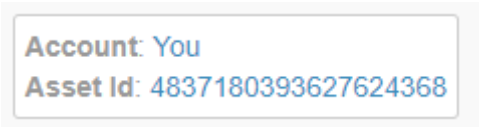

Figure 6 - Asset ID

After the completion of the transaction, the TagItLedger application will publish a new property “nxt” in the blockchain with the transaction ID, which is the same with the asset ID, for mapping purposes.

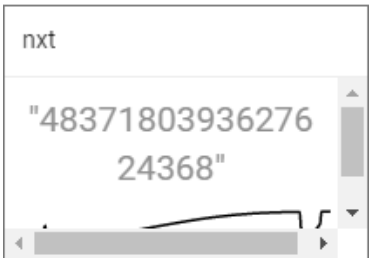

Figure 7 - Nxt property with asset id

Upon a second scan the user transfers the asset to the next “owner”, by utilizing the “nxt” property. Figures below, shows the transaction of asset transfer and the transaction info.

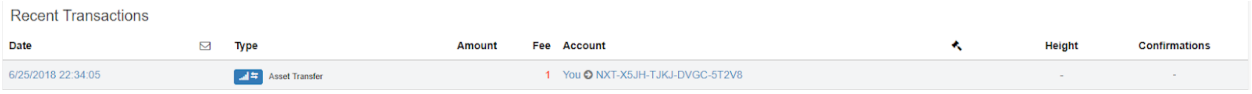

Figure 8 - Asset Transfer

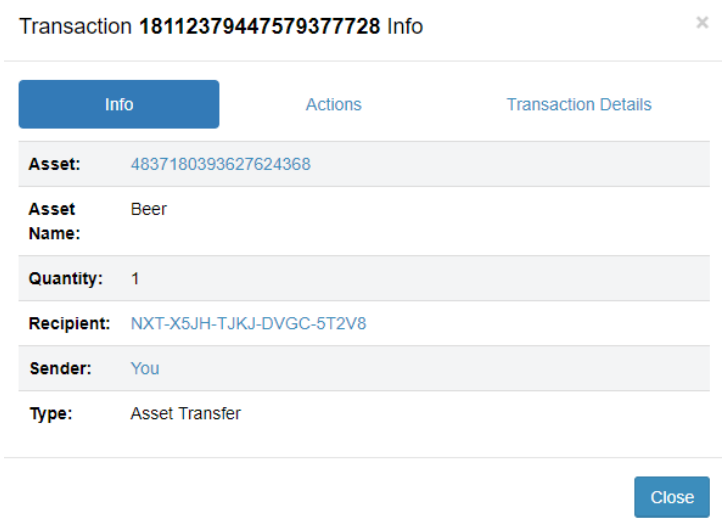

Figure 9 - Transaction info

4.2 Dynamic Pricing

This use case will use an example product that can benefit from conditions dependent pricing. More specifically, SmartTags printed with functional ink are designed to react when the temperature reaches the threshold of more than 7 degree Celsius for longer than 30 minutes. Users were able to claim the discount on the product if they scan the SmartTag with the installed TagItSmart application on their mobile device. The main benefits of the Dynamic Pricing use case are the ability to monitor cold supply chain on the item level, from the production place to the moment consumers interact with the product in a shop and the ability to provide additional, potentially personalized services to consumers whenever they use the product, again in the same way as the previous use case.

In terms of TagItLedger, we assume again that the scanning phase has already taken place and through the Pub/Sub broker, the TagItLedger app will get access to the scanned value (e.g. the temperature) and store it on the Nxt blockchain.

4.2.1 Usage Example

In this example, again we simulate the first part of the scanned data and we notify the app user with the new data value through the EVT Pub/Sub. More specifically, Figure 10 shows the notification showed to a TagItLedger user upon new value on a thng. In our case is the temperature property with value 10.1 as show in the mobile app. The subscribed Android TagItLedger app receives the appropriate notification. As we can see in Figure 5, the topic follows a specific path to utilize the EVRYTHNG's platform Pub/Sub broker "/thngs/U4sxK3FbVg8at5aaaXk5pmGf/properties/temperature", while the published value is in json format "[{"value": 10.1}]".

Upon receiving the notification, the Android app creates the new transaction to the Nxt blockchain. The new created transaction, contains all the data received through the EVRYTHNG Pub/Sub broker. Nxt blockchain supports tags, for easily search of transaction for a specific topic. In this example we used the tag “tag” for demonstration purposes.

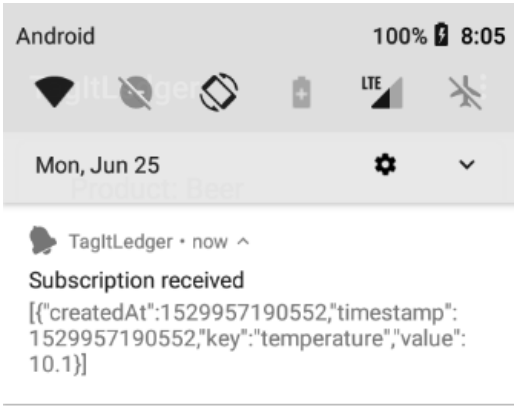

Figure 10 - Notification Message

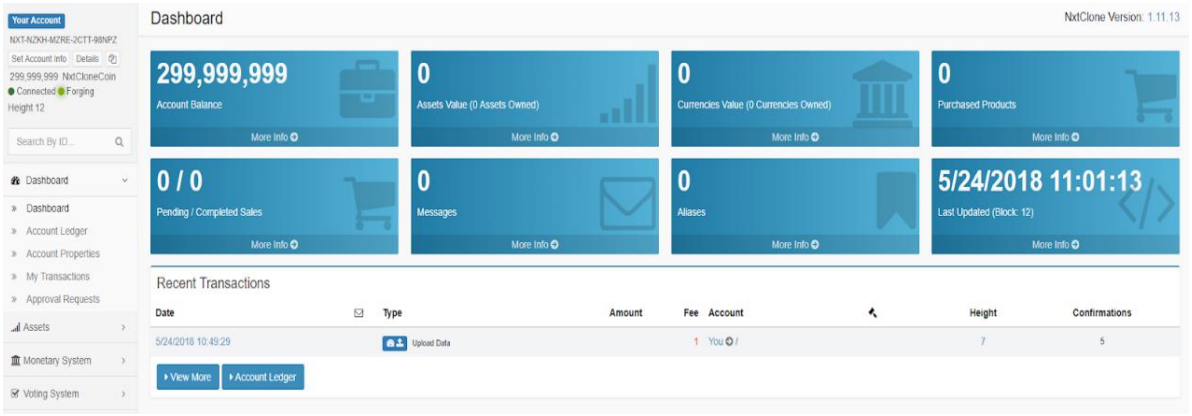

Figure 11 - The new transaction in the NXT blockchain

Figure 12, depicts some transaction information such as the hash, the type and the used tag.

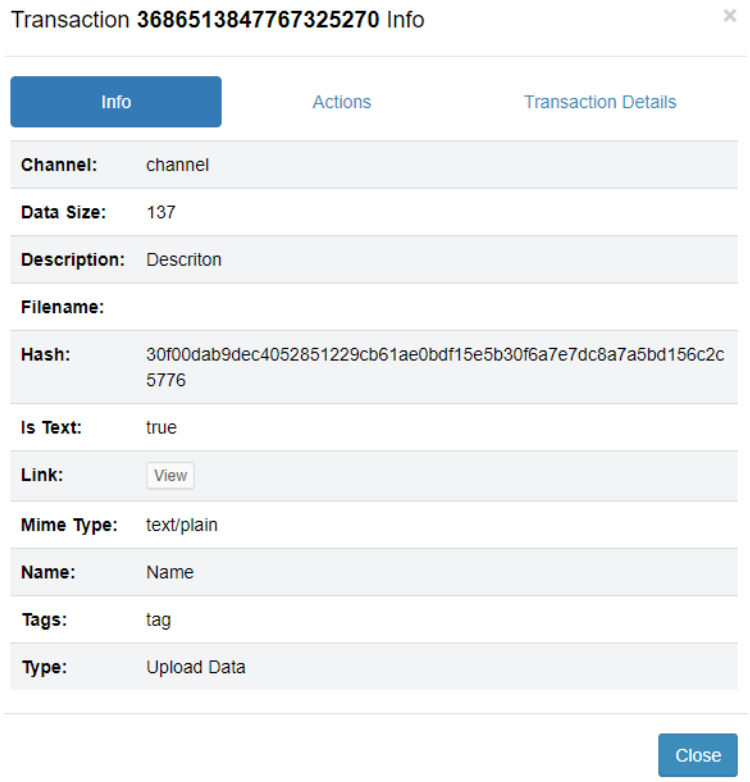

Figure 12 - Transaction Info

Figure 13, depicts how the data stored inside the transaction, among the timestamp, the changed property (e.g. temperature) and the value of that property.

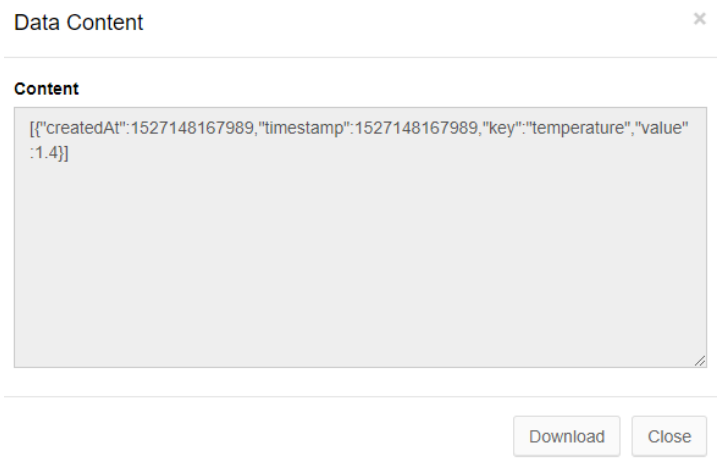

Figure 13 - Transaction Data Content

5 Conclusions

This document has reported the integrated extension prototype and the relevant software documentation. We presented thoroughly the TagItLedger Architecture, as well as potential usage with the of TagItSmart use cases, namely Digital Product/Lifecycle Management and Dynamic Pricing. We also presented the guidelines to create and setup a private Nxt blockchain and how to achieve the communication between the TagItLedger middleware and the blockchain. The implementation of the TagItLedger middleware, the android application, assist to the overall history record of the product, while at the same time enhance transparency towards the end user. In addition, this work is the result of few months so aspects of real scalability and performance, as well as security needs further to be studied and examined.

6 References

[kit2018] The Nxt Blockchain Creation Kit allows blockchain developers to create and launch their own blockchain platform based on Nxt, available at:
https://www.jelurida.com/blockchain-creation-kit
[mqttbox] MQTTBox, available at:
http://workswithweb.com/mqttbox.html
[unicornd4.1] Continuous Security and Privacy-by-Design Mechanisms – Early Release Deliverable D4.1, available at:
http://unicorn-project.eu/wp-content/uploads/2018/05/unicorn_d4.1_vfinal.pdf
[nxtapi] Nxt http API, available at:
https://demo.nxt.org/test
[TagItSmartD1.2] TagItSmart, Initial TagItSmart! Reference Architecture and API description, http://www.tagitsmart.eu/D1.2.pdf
[D5.3] Field trial implementation and evaluation on information use
[nxtwhite] Nxt community, Nxt Whitepaper, 2014
https://bravenewcoin.com/assets/Whitepapers/NxtWhitepaper-v122-rev4.pdf
[results] Nxt Blockchain - Loadtest Results, 2017. available at: https://www.jelurida.com/nxt-loadtest-results

7 Appendix A

According to the first step of the Docker Compose process, a Dockerfile for the definition of the environment that Nxt node needs was created. The code snippet below depicts snippets of such a Dockerfile. As we can see, we use as a base image of the Dockerfile the phusion/baseimage which is a standard ubuntu based image. This is an image that Nxt node is based on.

All the instructions that follow this line modify the base image and leads us to the new configured Nxt node. What follows is a RUN instruction, which installs all the prerequisites from respective Ubuntu repositories and updates the nxt.properties file. These packages include java 8 packages and other functionalities that are needed to compile the Nxt source. Another inline instruction is the CMD one. With this instruction we set the command to be executed when running the image, in our case a shell script file.

```
FROM phusion/baseimage
# start off with standard ubuntu images

#java8
RUN sed 's/main$/main universe/' -i /etc/apt/sources.list
RUN apt-get update && apt-get install -y software-properties-common python-software-
properties
RUN add-apt-repository ppa:webupd8team/java -y
RUN apt-get update
RUN apt-get install -y wget unzip
RUN echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true |
/usr/bin/debco$
RUN apt-get install -y oracle-java8-installer

# run and compile nxt
RUN mkdir /nxt
ADD . /nxt
# repo has
ADD contrib/docker_start.sh /docker_start.sh
# set nxt to listen on all interfaces
RUN echo 'nxt.allowedBotHosts=*' >> /nxt/conf/nxt.properties
RUN echo 'nxt.apiServerHost=0.0.0.0' >> /nxt/conf/nxt.properties
RUN echo 'nxt.addOns=nxt.addons.JPLSnapshot' >> /nxt/conf/nxt.properties
RUN echo 'nxt.adminPassword=123456' >> /nxt/conf/nxt.properties
RUN echo 'nxt.defaultPeers=172.18.0.1;172.18.0.2;172.18.0.3;172.18.0.4' >>
/nxt/conf/nxt.properties
RUN chmod +x /docker_start.sh
```

```
RUN cd /nxt; ./compile.sh
# both Nxt ports get exposed
EXPOSE 7874 7876
CMD ["/docker_start.sh"]
```

Dockerfile snippet file

According to the second step of the Docker Compose process, a docker-compose.yml file must be created, for the orchestration of the Nxt nodes that consist the blockchain. A very simple docker-compose.yml file is depicted below. This Compose file defines two services called “nxt1” and “nxt2”. Both services/nodes use the image that is called “nxt”.

```
version: '2.0'

services:
  nxt1:
    image: nxt1
    container_name: nxt
    build: .
    ports:
      - "7876:7876"
      - "7874:7874"

  nxt2:
    image: nxt2
    build: .
    ports:
      - "7886:7876"
      - "7884:7874"


networks:
  ee-net:
    driver: bridge
    ipam:
      config:
      - subnet: 172.18.0.0/24
        gateway: 172.18.0.1
        aux_addresses:
          nxt1: 172.18.0.2
          nxt2: 172.18.0.3
```

docker-compose.yml snippet file

The last step of the Docker Compose process is to run “docker-compose up”. Compose pulls the nxt image and runs it in order to create a container and deploy our Nxt nodes.

8 Appendix B

According to the main TagItLedger Application functionalities are the MQTT Subscription, the Notification and the post data to the blockchain.

```
//MQTT
CLIENT = mPresenter.mqttSetup(getBaseContext());
mPresenter.mqttConnect();

CLIENT.setCallback(new MqttCallback() {
    @Override
    public void connectionLost(Throwable cause) {
        Log.d("token:",cause.toString());
    }

    //background notification
    @Override
    public void messageArrived(String topic, MqttMessage message)
throws Exception {
        Log.d("topic:" + topic, "message:" + message.toString());
        createNotification(message.toString());
        mPresenter.postTaggedData(message.toString());
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {
        Log.d("token:",token.toString());
    }
});
```

MQTT Subscription snippet

```
public void createNotification(String aMessage) {
    final int NOTIFY_ID = 1002;

    // There are hardcoding only for show it's just strings
    String name = "my_package_channel";
    String id = "my_package_channel_1"; // The user-visible name of
the channel.
    String description = "my_package_first_channel"; // The user-
```

```
visible description of the channel.

   Intent intent;
   PendingIntent pendingIntent;
   NotificationCompat.Builder builder;

   if (notifManager == null) {
       notifManager =
               (NotificationManager)getSystemService(Context.NOTIFIC
ATION_SERVICE);
   }

   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
       int importance = NotificationManager.IMPORTANCE_HIGH;
       NotificationChannel mChannel =
notifManager.getNotificationChannel(id);
       if (mChannel == null) {
           mChannel = new NotificationChannel(id, name, importance);
           mChannel.setDescription(description);
           mChannel.enableVibration(true);
           mChannel.setVibrationPattern(new long[]{100, 200, 300,
400, 500, 400, 300, 200, 400});
           notifManager.createNotificationChannel(mChannel);
       }
       builder = new NotificationCompat.Builder(this, id);

       intent = new Intent(this, MainActivity.class);
       intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
Intent.FLAG_ACTIVITY_SINGLE_TOP);
       pendingIntent = PendingIntent.getActivity(this, 0, intent,
0);

       builder.setContentTitle(this.getString(R.string.notification_
title))  // required
               .setSmallIcon(android.R.drawable.ic_popup_reminder)
// required
               .setContentText(aMessage)  // required
               .setDefaults(Notification.DEFAULT_ALL)
               .setAutoCancel(true)
               .setContentIntent(pendingIntent)
               .setStyle(new
NotificationCompat.BigTextStyle().bigText(aMessage))
               .setVibrate(new long[]{100, 200, 300, 400, 500, 400,
```

```
300, 200, 400});
    } else {
        builder = new NotificationCompat.Builder(this);

        intent = new Intent(this, MainActivity.class);
        intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
Intent.FLAG_ACTIVITY_SINGLE_TOP);
        pendingIntent = PendingIntent.getActivity(this, 0, intent,
0);
        builder.setContentTitle(this.getString(R.string.notification_
title)) // required
                .setSmallIcon(android.R.drawable.ic_popup_reminder)
// required
                .setContentText(aMessage)  // required
                .setDefaults(Notification.DEFAULT_ALL)
                .setAutoCancel(true)
                .setContentIntent(pendingIntent)
                .setVibrate(new long[]{100, 200, 300, 400, 500, 400,
300, 200, 400})
                .setStyle(new NotificationCompat.BigTextStyle()
.bigText(aMessage))
                .setPriority(Notification.PRIORITY_HIGH);
    } // else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    Notification notification = builder.build();
    notifManager.notify(NOTIFY_ID, notification);
}
```

Create notification snippet

```
@Override
public void issueAsset(String message) {
    if (appRemoteDataStore == null) {
        appRemoteDataStore = new AppRemoteDataStore();
    }

    subscription = appRemoteDataStore.issueAsset("issueAsset",
message, "Descriton", "0", "100000000",
"bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37",
"1","60", true)
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<IssueAssetResp>() {
                @Override
                public final void onCompleted() {
```

```
                    Log.d("nxt","completed");
                }

                @Override
                public final void onError(Throwable e) {
                    Log.e("nxt",e.getMessage());
                }

                @Override
                public void onNext(IssueAssetResp data) {
                    Log.d("nxt","transfer success");
                    view.updateNxtData(data.getAssetID());
                }
            });
}
```

Issue asset snippet

```
@Override
public void transferAsset(String asset) {
   if (appRemoteDataStore == null) {
       appRemoteDataStore = new AppRemoteDataStore();
   }

   subscription = appRemoteDataStore.transferAsset("transferAsset",
asset,"0", "100000000",
"bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37",
"1","60", "NXT-X5JH-TJKJ-DVGC-5T2V8")
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<TransferAssetResp>() {
                @Override
                public final void onCompleted() {
                    Log.d("nxt","completed");
                }

                @Override
                public final void onError(Throwable e) {
                    Log.e("nxt",e.getMessage());
                }

                @Override
                public void onNext(TransferAssetResp data) {
                    Log.d("nxt","transfer success");
                }
            });
}
```

Transfer asset snippet

```
public void postTaggedData(String message) {

    if (appRemoteDataStore == null) {
        appRemoteDataStore = new AppRemoteDataStore();
    }
    subscription =
appRemoteDataStore.uploadTaggedData("uploadTaggedData", message,
"Name", "Descriton", "tag", "channel", "0", "1",
"bf0ced0472d8ba3df9e21808e98e61b34404aad737e2bae1778cebc698b40f37",
"60")
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<TaggedDataResp>() {
                @Override
                public final void onCompleted() {
                    Log.d("nxt","completed");
                }

                @Override
                public final void onError(Throwable e) {
                    Log.e("nxt",e.getMessage());
                }

                @Override
                public void onNext(TaggedDataResp data) {
                    Log.d("nxt","transfer success");
                }
            });
}
```

Post tagged data method snippet

The main permissions used for this Applications showed below. Among them are INTERNET, ACCESS NETWORK STATE and WAKE LOCK.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="eu.ubitech.smenesidou.tagitledger">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
```

```
    <application
        android:name=".data.TagItLedgerApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service
android:name="org.eclipse.paho.android.service.MqttService" />
    </application>

</manifest>
```

AndroidManifest.xml

The front end UI screen described with this xml source.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">
```

```
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@android:drawable/ic_popup_sync" />

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:paddingTop="52dp"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</android.support.design.widget.CoordinatorLayout>
```

activity_main.xml