D1.1 LAUNCH VERSION OF THE ASHVIN PLATFORM

Mirko Teodorović, Mainflux

@AshvinH2020
ASHVIN H2020 Project
www.ashvin.eu

ASHVIN has received funding from the European Union’s Horizon 2020 research and innovation programme under Grant Agreement No 958161. This document reflects only the author’s view and the Commission is not responsible for any use that may be made of the information it contains.
## Project Title
Assistants for Healthy, Safe, and Productive Virtual Construction Design, Operation & Maintenance using a Digital Twin

## Project Acronym
ASHVIN

## Grant Agreement No
958161

## Instrument
Research & Innovation Action

## Topic
LC-EEB-08-2020 - Digital Building Twins

## Start Date of Project
1st October 2020

## Duration of Project
36 Months

## Name of the deliverable
Launch version of the ASHVIN platform

## Number of the deliverable
D1.1

## Related WP number and name
WP 1 IoT driven digital twin platform

## Related task number and name
T1.1 IoT Platform

## Deliverable dissemination level
PU

## Deliverable due date
01-12-2020

## Deliverable submission date
31-12-2020

## Task leader/Main author
Mirko Teodorović (MFL)

## Contributing partners
Timo Hartmann (TUB), Rahul Tomar (DTT), Ilias Koulalis (CERTH), Jason Pridmore (EUR), Ken Gavin (NGEO), Rolando Chacón (UPC)

## Reviewer(s)
Timo Hartmann (TUB), Rahul Tomar (DTT)

### ABSTRACT
MFL deployed, in the cloud managed by Digital Ocean, an open-source, secure and scalable IoT platform that ensures device provisioning, connectivity and management, data accumulation and consumption. The platform is based on the microservices combined with Kubernetes orchestration mechanisms. The platform is available at https://iot.ashvin.eu. A set of user and technical manuals is provided in the appendix of this report. Deliverable 1.1. represents the launch version of the Ashvin platform that will be further improved throughout the project and a final version will be described in D1.6 Full Version of the Ashvin Digital Twin Platform.

### KEYWORDS
IoT platform, microservices, cloud, user manuals, ABAC, edge computing, protocols, data formats
REVISIONS

<table>
<thead>
<tr>
<th>Version</th>
<th>Submission date</th>
<th>Comments</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>28.12.2020</td>
<td>Initial draft</td>
<td>Mirko Teodorović (MFL)</td>
</tr>
<tr>
<td>V0.2</td>
<td>29.12.2020</td>
<td>Editing and formatting</td>
<td>Timo Hartmann (TUB)</td>
</tr>
</tbody>
</table>

DISCLAIMER

This document is provided with no warranties whatsoever, including any warranty of merchantability, non-infringement, fitness for any particular purpose, or any other warranty with respect to any information, result, proposal, specification or sample contained or referred to herein. Any liability, including liability for infringement of any proprietary rights, regarding the use of this document or any information contained herein is disclaimed. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by or in connection with this document. This document is subject to change without notice. ASHVIN has been financed with support from the European Commission. This document reflects only the view of the author(s) and the European Commission cannot be held responsible for any use which may be made of the information contained.
**ACRONYMS & DEFINITIONS**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IIoT</td>
<td>Industrial Internet of Things</td>
</tr>
<tr>
<td>CERTH</td>
<td>Centre of Research &amp; Technology - Hellas</td>
</tr>
<tr>
<td>DTT</td>
<td>Digital Twin Technology GmbH</td>
</tr>
<tr>
<td>FAS</td>
<td>Przedsiębiorstwo Robót Elewacyjnych Fasada sp.zo.o.</td>
</tr>
<tr>
<td>ABAC</td>
<td>Attribute-based access control</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, read, update and delete</td>
</tr>
<tr>
<td>AIP</td>
<td>Ashvin Internet of Things Platform</td>
</tr>
</tbody>
</table>
ASHVIN aims at enabling the European construction industry to significantly improve its productivity, while reducing cost and ensuring absolutely safe work conditions, by providing a proposal for a European wide digital twin standard, an open source digital twin platform integrating IoT and image technologies, and a set of tools and demonstrated procedures to apply the platform and the standard proven to guarantee specified productivity, cost, and safety improvements. The envisioned platform will provide a digital representation of the construction product at hand and allow to collect real-time digital data before, during, and after production of the product to continuously monitor changes in the environment and within the production process. Based on the platform, ASHVIN will develop and demonstrate applications that use the digital twin data. These applications will allow it to fully leverage the potential of the IoT based digital twin platform to reach the expected impacts (better scheduling forecast by 20%; better allocation of resources and optimization of equipment usage; reduced number of accidents; reduction of construction projects). The ASHVIN solutions will overcome worker protection and privacy issues that come with the tracking of construction activities, provide means to fuse video data and sensor data, integrate geo-monitoring data, provide multi-physics simulation methods for digital representing the behavior of a product (not only its shape), provide evidence based engineering methods to design for productivity and safety, provide 4D simulation and visualization methods of construction processes, and develop a lean planning process supported by real-time data. All innovations will be demonstrated on real-world construction projects across Europe. The ASHVIN consortium combines strong R&I players from 9 EU member states with strong expertise in construction and engineering management, digital twin technology, IoT, and data security / privacy.
# TABLE OF CONTENTS

1 IOT PLATFORM ........................................................................................................... 8
   1.1 Ashvin IoT platform in the context of the Ashvin toolkit .......................................... 9
   1.2 Platform features .................................................................................................... 10

1.3 Platform Access ...................................................................................................... 10
   1.3.1 System management ............................................................................................ 10
   1.3.2 Messaging ........................................................................................................... 11

2 FINE-GRADED ACCESS ............................................................................................ 12
   2.1 Current administration system ................................................................................ 12
   2.2 New administration system (work in progress) ....................................................... 12
   2.3 Administration system development roadmap ....................................................... 13
   2.4 State of development .............................................................................................. 13

3 EDGE COMPUTING MECHANISMS ......................................................................... 15
   3.1 Edge connectivity ..................................................................................................... 15
   3.2 Edge communication mechanisms .......................................................................... 15
   3.3 Rules engine ............................................................................................................ 15
   3.4 Network bandwidth and storage costs .................................................................... 16
   3.5 IoT data analytics .................................................................................................... 16
   3.6 Ongoing research .................................................................................................... 16

4 DATA GATHERING AND EXCHANGE MECHANISMS ................................................. 17
   4.1 Data exchange mechanisms ..................................................................................... 17
   4.2 Data gathering mechanisms .................................................................................... 17

5 MEETINGS .................................................................................................................. 18

6 CONCLUSION ............................................................................................................. 19
### APPENDIX 1: ABOUT ASHVIN IOT PLATFORM

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 What is Ashvin</td>
<td>21</td>
</tr>
<tr>
<td>7.2 What is Ashvin IoT platform (AIP)?</td>
<td>21</td>
</tr>
<tr>
<td>7.3 AIP Address</td>
<td>22</td>
</tr>
</tbody>
</table>

### APPENDIX 2: ASHVIN IOT PLATFORM ENTITIES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Things</td>
<td>23</td>
</tr>
<tr>
<td>8.2 Channels</td>
<td>23</td>
</tr>
<tr>
<td>8.3 Messages</td>
<td>24</td>
</tr>
<tr>
<td>8.4 Users</td>
<td>24</td>
</tr>
<tr>
<td>8.5 Twins</td>
<td>25</td>
</tr>
<tr>
<td>8.5.1 Twin’s Anatomy</td>
<td>26</td>
</tr>
</tbody>
</table>

### APPENDIX 3: ASHVIN IOT PLATFORM RELATED TOOLS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Create, read, update, and delete (CRUD) entities</td>
<td>28</td>
</tr>
<tr>
<td>9.1.1 Ashvin IoT GUI</td>
<td>28</td>
</tr>
<tr>
<td>9.1.2 Postman</td>
<td>28</td>
</tr>
<tr>
<td>9.2 Send and receive messages</td>
<td>28</td>
</tr>
<tr>
<td>9.2.1 Mosquitto</td>
<td>28</td>
</tr>
</tbody>
</table>

### APPENDIX 4: ASHVIN IOT PLATFORM GUI

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Login screen</td>
<td>29</td>
</tr>
<tr>
<td>10.2 Dashboard</td>
<td>29</td>
</tr>
<tr>
<td>10.3 Things</td>
<td>30</td>
</tr>
<tr>
<td>10.4 Channels</td>
<td>35</td>
</tr>
<tr>
<td>10.5 Twins</td>
<td>37</td>
</tr>
</tbody>
</table>

### APPENDIX 5: POSTMAN

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 User authentication request</td>
<td>42</td>
</tr>
<tr>
<td>11.2 Postman environments</td>
<td>45</td>
</tr>
</tbody>
</table>
11.3 Create thing request .................................................................................. 47
11.4 Retrieve thing request ................................................................................ 50
11.5 Requests specifications ................................................................................ 51
  11.5.1 Automatic request generation ................................................................. 53
  11.5.2 Ashvin Collection .................................................................................. 54

12 APPENDIX 6: MOSQUITTO ............................................................................. 56
  12.1 MQTT client .................................................................................................. 56
  12.2 Subscribe and publish .................................................................................. 56
  12.3 SenML .......................................................................................................... 57
1 IOT PLATFORM

Ashvin IoT platform is based on Mainflux, a messaging middleware geared towards the Internet of things. Mainflux is a scalable, secure, open-source, and patent-free IoT cloud platform written in Go. It accepts connections over various network protocols (i.e. HTTP, MQTT, WebSocket, CoAP).

Mainflux IoT platform consists of multiple microservices with separate and well-defined responsibilities such as management of users and authentication concerns, management of things and channels - Mainflux IoT platform primitives used to build complex IoT topologies – and authorization concerns. These microservices can be run in many different ways according to the use case needs. They can be run locally or in the cloud, on premise or off-premise. They can be run as a composition of Docker containers (a lightweight, standalone operating system abstraction that includes everything needed to run an application) or as standalone applications on the local host computer.

Mainflux IoT as a core of the Ashvin IoT platform has a complex architecture. Blue boxes on the picture bellow show Mainflux primitive entities, such as users, things, adapters, etc. Green cylinders are databases where the messages are persisted. Green cylinders are databases where the messages are persisted. White cylinders are Mainflux internal message buses. They centralize different message pathways (data traveling via different protocols of communication). NGINX acts as Mainflux internal domain name server (reverse proxy) and forwards requests coming from the “outside world”, i.e. from platform clients, to the corresponding services (users, things, etc.). Finally, the cloud icon refers to clients of the IoT platform - devices (data produces) and applications (data consumers).
For the first version of the Ashvin platform, we have chosen to install Mainflux IoT platform as a **Docker container composition orchestrated by Kubernetes.** Mainflux is installed in the cloud managed by [DigitalOcean](https://www.digitalocean.com), a cloud infrastructure provider with data centres worldwide, including Europe. The servers of the Ashvin IoT platform are located in [Frankfurt](https://www.frankfurt.de), but this can be changed easily. **NB: we will use Ashvin IoT platform and Mainflux IoT platform interchangeably in this report.**

### 1.1 Ashvin IoT platform in the context of the Ashvin toolkit

The Ashvin IoT platform (based on Mainflux) is the messaging (communication) core of the Ashvin toolkit. It is meant to connect data producers – usually, physical devices on the construction site – to the data consumers – Ashvin toolkit applications processing the data. Its role is also to store the data. The image below shows the Ashvin toolkit “stack”. The IoT platform is in the middle of the stack, thus connecting the “south” end of the stack (data producers) with the “north” end of the stack (data consumers).
1.2 Platform features
The main and fully functional features of the deployed platform are following:

- **User management**, where the user represents the real (human) user of the system or an organization.
- **System provisioning**
  - Provisioning things, where things represent devices or applications connected to the platform.
  - Provisioning channels, where channels represent communication pathways between devices and/or applications. Channel serves as a message topic that abstracts away complexities of the underlying communication protocols. It is used by things to send and receive messages.
- **Messaging**
  - with supported protocols
    - HTTP
    - MQTT
    - WS
    - CoAP
    - LoRaWAN
    - OPC UA
  - Mutual TLS Authentication with X.509 Certificates over HTTPS, and MQTTs.
- **Storage** - platform supports various databases for message persistence (storing): - CassandraDB - MongoDB - InfluxDB - PostgreSQL

1.3 Platform Access

1.3.1 System management
Platform user can access the platform in multiple ways. For user management and system provisioning and storage, we use an HTTP RESTful API.

Some of the well-known RESTful API clients include command line tools such as CURL and GUI tools such as Postman. The use of these kinds of tools is documented in the Ashvin IoT platform user manual.

Mainflux IoT platform - a backbone of the Ashvin IoT platform - provides also a custom command line tool - `mainflux-cli` - as well as a custom GUI tool – Ashvin IoT platform GUI. The use of the latter is documented in the Ashvin IoT platform user manual ().

All of these tools are already fully functional and are ready for use in production as of this writing. Ashvin IoT platform experimental GUI can be accessed at the address [https://iot.ashvin.eu/](https://iot.ashvin.eu/). The address `iot.ashvin.eu` itself can be used with any of the
tools in order to use the Mainflux IoT platform for user management, system provisioning or messaging.

1.3.2 Messaging
Mainflux IoT platform is a messaging middleware. It is akin to a post office in the sense that it collects messages on the so-called south end or edge (fog) – where data producers reside – and relays messages to the so called north end (cloud) – where data consumers reside.

Once a channel is provisioned and a thing is connected to it, the latter can start publishing and receiving messages over the former. HTTP, MQTT, WS and CoAP are supported out of the box and don’t require any special configuration. LoRaWAN requires that a device be connected to a LoRaWAN server. Once device is connected to the server, we can connect it with the Ashvin IoT platform. OPC UA works out of the box, but it is still in the experimental stage.

You can use command line tools such as CURL and GUI tools such as Postman to send messages via HTTP, the mosquitto_pub and mosquitto_sub MQTT clients to send and receive messages over MQTT. To send and receive messages over CoAP, you can use Copper CoAP user-agent. Naturally, those are only some of the well-known tools and are recommended by Mainflux Labs. The use of these tools with the Mainflux IoT platform will be documented in the Ashvin IoT platform user manual.
2 FINE-GRADED ACCESS

Mainflux team started an implementation of the fine-graded access based on the Attribute-based access control (ABAC) model. The ABAC model enables easy creation and management of users, channels and things groups as well as per group and per entity (user, channel or thing) permissions.

2.1 Current administration system
Currently, Mainflux users are represented by email and password. The latter are used as platform access credentials in order to obtain an access token, i.e. as a means of user authentication. Once logged into the system, the user can manage resources in a CRUD manner and define things-channels access control policies (authorization) by connecting things and channels. One should not confuse user authentication with the thing authorization. The former is used to authenticate a user and the latter is used to check whether a given thing can send messages to and receive messages from the given channel.

2.2 New administration system (work in progress)
The development of the new administration system is ongoing and is nearly done. In the new system there is a root admin. It is created by default and, so, there is always a root admin and there is a possibility to create additional super admins. This system is similar to UNIX root and regular users system.

The main role of the root admin and super admins is to organize users into groups to match the structure of business applications and to organize the access rights. User groups are further divided into subgroups so there is a hierarchical organization of parent, children and sibling super- and subgroups of an arbitrary depth.

User groups should be matched with things (devices and apps) groups. When adding devices or applications in the system, they should be organized so that organization matches the structure of company organization or some other real world organization structure (for example devices can be organized in flats, buildings, streets, cities, etc.).
2.3 Administration system development roadmap
Mainflux users and things groups have a hierarchy and various metadata. Groups can have parents and children. This opens up the possibility and need for complex queries of users, things and channels along the lines of metadata and hierarchical structure. Therefore, queries for traversing relationship trees are on the development roadmap.

2.4 State of development
The current state of development of the new administration system can be followed anytime at the address https://github.com/mainflux/mainflux/pull/1246 as well as at the address https://github.com/mainflux/mainflux/pull/1313. This development is nearly done, so as of this reading, it may be very well merged into the main version that can be found anytime here https://github.com/mainflux/mainflux.
Once the new version of Mainflux is out, it will be tested on an Ashvin demo deployment (found also on Digital Ocean) and if everything goes well, it will deployed on the [https://iot.ashvin.eu/](https://iot.ashvin.eu/). Once deployed, all the existing users of the platform will be migrated to use the new administration system.
3 EDGE COMPUTING MECHANISMS

“Edge”, “fog” or “south-end” is an Industrial Internet of Things (IIoT) terminology used to refer to the very physical site where data sources are located. Data sources are various and tend to be numerous, even on a single production site. Even a single machine can use several protocols or means (such as CSV files) to emit measurement data and to receive control data.

This means two things. Firstly, there is a plethora of communication methods that need to be addressed by the IoT platform. Secondly, there is a high quantity of produced data. This data is collected by a gateway - a south-end hardware device that hosts an edge IoT platform and serves as a single point of collection of the production site messages. Usually, a gateway forwards this data to the cloud applications.

3.1 Edge connectivity
One of the special challenges of the Mainflux IoT platform is enabling the connectivity of the various types of devices over the gateway. Since there is no “one size fits all” solution, various services need to be developed that enable device-gateway communication. The Mainflux IoT platform supports several protocols - MQTT, WS, CoAP, HTTP, LoRaWAN and OPC UA - out of the box. If a device supports one of these protocols, this means that it can be directly connected to the platform. However, if a device is not connected to the internet or the device supports a specific communication protocol, a special software service (a “daemon”) must be written in order to enable one-way or two-way device-gateway communication.

3.2 Edge communication mechanisms
Mainflux IoT platform has, as an add-on, export service. The latter can send messages from one Mainflux cloud to another via MQTT, or it can send messages from edge gateway to Mainflux Cloud. The development of the export service is ongoing and can be found here https://github.com/mainflux/export.

Mainflux IoT Agent is a communication, execution and software management agent for Mainflux IoT platform that runs on the gateway. The development of the Agent is ongoing and can be found here https://github.com/mainflux/agent.

3.3 Rules engine
Rules engine is a production rule system. The rule-based approach is dependent on a knowledge representation that pertains to a specific domain. A rule has a condition and an action. Rules engine uses knowledge representation to encode a current state of the system. Based on the current state, the rules engine evaluates rules and takes appropriate actions if the rule conditions associated with these
actions are met. This enables a real-time processing of production data on the IIoT edge and specifically, on a gateway. This is important for two reasons. Firstly, we want to save network bandwidth and storage costs. Secondly, we want to be able to do some kind of basic data analytics on the data production site.

### 3.4 Network bandwidth and storage costs

The amount of production site data is usually so huge that there are significant network bandwidth and storage costs associated with data fog-cloud transfer. One of the aims of rules engine is to transform and filter data in order to save network bandwidth and reduce storage requirements.

### 3.5 IoT data analytics

Edge lightweight IoT data analytics software such as rules engine is used for predictive maintenance and to improve system safety. By analysing and processing data from different data sources, various failures related to operation and safety can be predicted and a further insight into the system operation can be gained.

### 3.6 Ongoing research

We are currently analysing and testing two candidates for the rules engine that will be used on the edge. The candidates are Grule and EMQ X Kuiper. The idea is to integrate one of these two rules engines into the existing Mainflux IoT platform. Both of the rules engines are written in the programming language Go and are lightweight and are thus well adapted for the resource constrained edge devices such as gateways. The projected end of research is the end of December 2020. We should start the integration of the rules engine with the Mainflux IoT platform in the January 2021. However, if we decide that none of the existing rules engine matches the special needs of the Ashvin use cases and the internals of the Mainflux architecture, we will start the development of the custom in-house rules engine.
4 DATA GATHERING AND EXCHANGE MECHANISMS

4.1 Data exchange mechanisms
One of the problems that need to be solved is the data exchange mechanism between Mainflux IoT platform consumers - Ashvin tools, parts of the Ashvin toolkit - and the platform itself. Two things need to be taken into account. Firstly, we need to determine compatible data formats. Secondly, we need to address the multitude of protocols used to transfer messages or data.

Mainflux is collaborating closely with CERTH and DTT on defining the data format and the data exchange mechanisms. For now, the preferred way to transfer data is via the RESTful mechanisms and the preferred data format is JSON. Mainflux IoT platform has already well defined RESTful API that is thoroughly documented by means of OpenAPI specification documents and is ready to use.

4.2 Data gathering mechanisms
The exact details of the demo sites are still unknown. However, in close collaboration with FAS, Mainflux Labs created a detailed questionnaire for the demo site owners in order to gather as much details as possible about the data sources (various data producing devices such as sensors) existing on the demo sites. Also, there is an ongoing effort with the representatives of the Zadar airport to determine the best network and device setup for the demonstration site in question.

Mainflux also launched an independent research on the sensors typically used on construction sites and in buildings. The reason behind this research is not only to predict possible challenges on construction sites, but also to make a recommendation document that details how to connect, or enrich with connectivity, construction sites which are not connected, or are not sufficiently connected, to some sort of network (LAN, WAN, private, public, etc.) or are not sufficiently equipped with sensors.
5 MEETINGS

Mainflux Labs organized two meetings related to the work package 1. The first meeting was an initial meeting. The meeting was attended by the representatives of TUB, DTT, EUR, FAS and UPC. The second meeting, related to the planning and development of WP-1 and related WP-3 tasks was attended by DTT and UPC. Meeting minutes can be found in the https://nextcloud.ashvin.eu/ > Documents > WP1 > Meeting Minutes. Representatives of Mainflux Labs also participated in the numerous meetings organized by various Ashvin partners.
6 CONCLUSION

This accompanying report briefly describes Deliverable 1.1 of the Ashvin project – the launch version of the Ashvin IoT platform available at https://iot.ashvin.eu. The platform serves as basis for all technical development of the Ashvin project. Having an initial version of the platform available at the start of the project already allows for data collection from demonstration projects. The platform will be further improved throughout the project and a final version will be described in D1.6 Full Version of the Ashvin Digital Twin Platform.
D1.1 Launch Version of the Ashvin Platform
7 APPENDIX 1: ABOUT ASHVIN IOT PLATFORM

7.1 What is Ashvin

*Ashvin* strives to develop a solution that will place the European construction industry on the international stage as the number one example to follow. It will do this by employing the power of digitalization and the cutting-edge Digital Twin Technology.

Digitizing and transforming the European construction industry

7.2 What is Ashvin IoT platform (AIP)?

Ashvin IoT platform is a modern, scalable and secure IoT platform. It can be deployed in cloud or on premise. It is meant to be run on Linux machines, but can be also deployed on Mac and Windows machines natively. AIP accepts connections over various network protocols (i.e. HTTP, MQTT, WebSocket, CoAP) making a seamless bridge between devices and applications.

While Ashvin project aims at providing the *Ashvin platform*, i.e. “an open source digital twin platform integrating IoT and image technologies, and a set of tools and demonstrated procedures to apply the platform” (*About Ashvin*), *Ashvin IoT platform (AIP)* is the underlying layer of connectivity that enables the real-time device to device, app to app, device to app and app to device communication. Thus, AIP plays the role of a low-level enabler of the higher level (more abstract) and end-user oriented aspects of the Ashvin platform. In a nutshell, AIP is a messaging middleware backbone of the Ashvin platform. AIP is based on *Mainflux*, an open-source and patent-free IoT messaging middleware written in Go.
Mainflux messaging middleware written in GO

AIP is made and maintained by Mainflux Labs.

7.3 AIP Address

Ashvin IoT platform can be accessed at iot.ashvin.eu internet address. In order to use it, you need a valid user, i.e. you need access credentials. To get the credentials, please write an email to ashvin@mainflux.com.
8 APENDIX 2: ASHVIN IOT PLATFORM ENTITIES

AIP features two basic entities used to represent physical devices or software applications and their communication. Firstly, **things** represent hardware such as gateways and devices (sensors and actuators) as well as software such as visualization tools, rules engines, etc. Secondly, **channels** represent means of communications between things.

Things exchange **messages** via channels. Things can send or publish messages over channels. Things can receive or listen to channel for the incoming messages. A Thing has to be connected to a channel in order to send and receive messages via the respective channel. Messages sent over channels are persisted in a **database**.

![Diagram of AIP entities and communication](image)

*Ashvin IoT platform architecture*

8.1 Things

AIP Thing represents any data **source** or producer (provider). It can be a sensor based physical device, cloud based app that generates data or any edge located data source such as gateway.

Thing represents also any data **destination** or consumer. It can be an actuator based physical device, a cloud based app that receives data - e.g. data visualization tool such as Grafana - or an edge located app such as machine software, gateways etc.

8.2 Channels

Channels are **communication** pathways or data transmission mediums. AIP things communicate via channels by a) **sending** messages to and b) **receiving** messages
from channels. Channels should be best thought of as pipes which let messages flow in both directions.

Channels abstract away complexities of low-level communication protocols (MQTT, HTTP, etc.) handling and offer a unified and easy to use interface to exchange messages.

### 8.3 Messages

Message consists of communication payload - information of interest - accompanied by a metadata such as who sent the message, when the message was sent, etc. AIP saves every message into a database of choice. DBs supported out of the box are MongoDB and InfluxDB.

AIP implements publish and subscribe model (pubsub model). Things publish messages over channels by means of HTTP, WebSocket, MQTT and CoAP protocols (other protocols are also supported). Things subscribe to channels by means of WebSocket, MQTT and CoAP protocols. Any thing can publish message to any channel and any thing can receive message from any channel. It is important to keep in mind that sending thing does not know anything about receiving things and vice versa, receiving things do not know anything about the sending thing - except for the sending thing’s id. Things are generally only “aware” of channels and messages.

This is how a typical Mainflux message looks like when received in the JSON format:

```json
{
    "channel": "038774f6-1d37-4d37-9107-487adc3466a4",
    "subtopic": "room3.occupancy",
    "publisher": "6d2d8ad1-87b1-4b72-9b6f-2c00ae8755a1",
    "protocol": "http",
    "name": "occupancy3",
    "time": 1608542793.263,
    "bool_value": false
}
```

The field publisher refers to the sending thing’s AIP internal id and time refers to Unix time. Instead of bool_value message can also have a float, string or binary value.

### 8.4 Users

Users are AIP tenants and represent physical persons and organizations. Users own channels and things. Users provide authentication mechanisms to AIP, i.e. a secure access to things and channels and their manipulation.

User is represented via e-mail and password. The latter are used as platform access credentials in order to obtain an access token. Once logged into the system, user can manage his resources (i.e. things and channels) in CRUD fashion and define access control policies by connecting them.

There is an important distinction to keep in mind here. While user provides access rights to
and allows one to perform CRUD operations on channels, things and their connections, thing’s ID and key are used as credentials - a sort of username and password - to send and receive messages over channels.

8.5 Twins

_Not to be confused with DTT’s digital twin._

In the IoT terminology, twin refers to a digital representation of a real world data system consisting of possibly multiple data sources/ producers and/or destinations/ consumers (data agents).

A machine can use multiple protocols such as MQTT and OPC UA, and a regularly updated, machine hosted CSV file to send measurement data - such as flow rate, material temperature, etc. - and state data - such as engine and chassis temperature, identity of the current human operator, etc. - as well as to receive control (messages) - such as, turn on/off light, increment/decrement borer speed, etc. On the cloud end of the IoT spectrum, an application can consume, but it can also generate data. There can be a whole series of logically interconnected applications that produce data.

Twin is an abstract digital replica of a real world system such as the machine we have just described or a series of logically interconnected applications, or both. It is used to store system’s state snapshots, to compare system states over a given period of time - so-called diffs or deltas - as well as to control agents composing the system.

Twin entity is built on top of the Ashvin IoT platform and relies on its architecture and entities, more precisely, on users, things and channels. Ashvin IoT twin consists of three parts:

- General data about twin itself, i.e. twin’s metadata,
- History of twin’s definitions, including current definition,
- History of twin’s states, including current state.

All twins and their states are persisted in the database. Currently, Ashvin IoT system supports MongoDB for twin persistence out of the box. (Messages can be persisted in MongoDB as well as in InfluxDB.)
AIP architecture with twin service

8.5.1 Twin’s Anatomy

Twin’s general information stores twin’s owner email - owner is represented by an Ashvin IoT platform user -, twin’s ID (internal unique identifier) and name (not necessarily unique), twin’s creation and definition update times as well as twin’s revision number - the sequential number of twin’s definition.

Twin’s definition is a semantic representation of system’s data producers and consumers (data agents). Each data agent is represented by means of an attribute. An attribute consists of data agent’s name, channel and subtopic. Nota bene: each attribute is uniquely defined by the combination of channel and subtopic and we cannot have two or more attributes with the same channel and subtopic in the same definition.

A typical twin’s definition might look like this:

```
{
  "id": 6,
  "created": "2020-12-15T09:05:19.445Z",
  "attributes": [
    {
      "name": "room2 > temperature",
      "channel": "32628244-4569-45fa-84c8-a4ac4a77f258",
      "subtopic": "room2.temperature",
      "persist_state": true
    },
    {
      "name": "room2 > humidity",
      "channel": "32628244-4569-45fa-84c8-a4ac4a77f258",
      "subtopic": "room2.humidity",
      "persist_state": true
    },
    {
      "name": "room2 > occupancy",
      "channel": "32628244-4569-45fa-84c8-a4ac4a77f258",
      "subtopic": "room2.occupancy",
```
"persist_state": true
},
"delta": 1000
}

A typical twin's state might look like this:

{
  "twin_id": "8905064a-af8d-4705-9970-bf6f6d49fe83",
  "id": 3501,
  "definition": 6,
  "created": "2020-12-15T12:13:55Z",
  "payload": {
    "room2 > humidity": 66.51826718978143,
    "room2 > occupancy": false,
    "room2 > temperature": 28.917830716862856
  }
}
9 APPENDIX 3: ASHVIN IOT PLATFORM RELATED TOOLS

Ashvin IoT platform (AIP) is at its core a messaging middleware. Except for the well-defined and comprehensive IoT entities CRUD and messaging oriented application programming interface, it does not offer much in terms of a user interface. However, there are many open source tools that can use AIP application programming interface in order to

- log in,
- perform create, read, update, and delete (CRUD) operations on AIP basic entities,
- send and receive messages.

Every interaction with the API requires a valid Ashvin IoT platform user.

9.1 Create, read, update, and delete (CRUD) entities

9.1.1 Ashvin IoT GUI

Ashvin IoT platform has a dedicated GUI for the platform operators. GUI is under development and can be used for all CRUD operations. GUI is not supposed to be used for messaging. In other words, it is used to create, read, update, and delete things and channels as well as their connections (for more information, please see entities). This is the easiest way to perform CRUD operations in AIP. You can get more information here.

9.1.2 Postman

Postman is the API Client that enables you to send HTTP requests and receive HTTP responses. Postman enables you to do manually what your browser is doing automatically for you. That means that you can use Postman to gain a greater and fine grain control over AIP CRUD operations. You can get more information here.

9.2 Send and receive messages

9.2.1 Mosquitto

Eclipse Mosquitto is an open source message broker that implements the MQTT protocol. The Eclipse Mosquitto project also hosts the development of the popular mosquitto_pub and mosquitto_sub command line MQTT clients. You can get more information here.
10 APPENDIX 4: ASHVIN IOT PLATFORM GUI

To operate AIP in an intuitive way, we use AIP dedicated graphical user interface (GUI). To be clear, this GUI is not a platform itself. It is only an external tool used to operate a platform. To understand why, please keep in mind that the AIP is at its core a messaging middleware that exposes a well-defined API to be used with various protocols such as HTTP, MQTT and CoAP.

AIP GUI leverages a RESTful web service design. In practice, this means that you can use HTTP methods, such as GET, HEAD, POST, PUT, etc. to perform create, read, update, and delete (CRUD) operations on the AIP entities and that is exactly what AIP GUI enable us to do.

NB: AIG GUI is not meant to be used to send messages. You can use Mosquitto client to send and receive messages over AIP.

10.1 Login screen

AIP GUI is located at iot.ashvin.eu. Once you go there, you are presented with the login screen. To use it, you need to be a valid user, i.e. you need access credentials. Please see here how to get the credentials.

10.2 Dashboard

Once you are logged in, you are presented with the dashboard.

GUI dashboard

In the central panel you can see the description and some useful links, amongst others, link to this documentation.

The menu on the right shows you the top-level options. The majority of these options reflect
AIP entities and these options let you operate, in a CRUD fashion, these entities.

10.3 Things

If you select the Things menu option, you will be presented with the following screen:

Things list

This is the list of the things that belong to your user, i.e. to the currently logged in user. To add a new thing, click on the plus sign, enter thing’s name and click the check mark sign:

Add thing
The thing’s ID will be automatically generated. Now you can click on the details icon (a magnifying glass) to see the details of your thing. You will be presented with the details screen:

![Thing's details](image)

**Thing’s details**

On the details screen, you can see thing’s ID and thing’s key. The former is the internal Ashvin IoT platform identifier. The latter is best understood as thing’s password. You need this in order to send and receive messages with this thing. The thing’s key is an AIP abstraction: it abstracts away complexities of different protocol authorization procedures.

As things abstract away complexities of protocol authorization, channels abstract away complexities of protocol message “routing”. In order to exchange messages via the thing in question, we need to connect the thing to the appropriate channel:
Connect **thing** and **channel**

In the upper-right corner of the screenshot, we see the possible channels listed in the drop down menu.

In this particular example, the channel represent single machine’s communication “route”. Our hypothetical machine is endowed with many sensors represented by AIP things and all this sensors communicate with a cloud application via this single channel. (Naturally, there are other ways to represent the real data system by using AIP things and channels.) This single channel could stand for (and abstract away complexities of) an mqtt host and topic, for example.

To connect a thing to a channel, simply select the channels and click on the connect button. You can also edit the thing’s metadata:
**Metadata**

Metadata is used to add an arbitrary data to a thing. We normally add a type of the thing. For example, a type can be device or an application. You can have more precise types like drilling machine, etc. This kind of data is called metadata because - unlike thing’s key and thing’s ID which are AIP internal data necessary for the proper functioning of the AIP -, AIP does not understand anything about the metadata structure and semantics. It is a responsibility of a user - be it a real person or another application built on top of AIP - to figure out the organization and meaning of the metadata.

To add a metadata, simply type a valid JSON object in the Metadata card and click on the edit button. If your JSON is valid, which you can check beforehand, for example, [here](#), you’ll get a metadata edit success notification.

Finally, in the card Messages, you can see all messages sent or received by the current thing:
D1.1 Launch Version of the Ashvin Platform

Thing’s messages in table format

You can get a “raw” message list - messages formatted as a list of JSON objects - by clicking on the table dropdown menu and selecting JSON:

Thing’s messages in JSON list format

A thing can be subscribed to multiple channels, and many things can be subscribed to one channel. To tell if a thing is a receiver or a sender of a message, just compare the publisher field of a particular message with thing’s ID. If they are the same, than the current thing is the publisher of the message.
10.4 Channels

The interface part for CRUD operations on channels follows the same logic as the interface part related to things. Firstly, you can list channels that you (or your user) have by simply clicking on the Channels in the left hand side menu.

Channels list

From there on, you can create the new channel in the same way you create a new thing. Just click on the plus sign, enter channel’s name and click the check mark sign. By clicking on the magnifying glass icon, you can visit particular channel’s details page.

Ashvin IoT platform supports one-to-one, one-to-many, many-to-one and many-to-many connections between arbitrary things and channels. In other words, you can connect any thing to any channel and vice versa. You can connect things and channels either on thing’s details page or on channel’s details page. The procedure is the same:
Connect thing and channel

You can edit channel’s metadata in the same way you edit things metadata. Just enter a valid JSON object and click on the Edit button. If the JSON object is valid, you will get a success notification:

Channel’s metadata

As with things, metadata is not related to internals of Ashvin IoT platform. Rather, it enables IoT system designers to make an IoT data system with a custom semantics and structure. Finally, you can also get a list of messages send or received over the channel in question by simply inspecting the card Messages:
D1.1 Launch Version of the Ashvin Platform

Channel’s messages in table format

Obviously, channel attribute of a particular message will always be the same as the channel ID of the current channel.

10.5 Twins

Mainflux twin is a digital representation of a real world data system consisting of multiple data sources/producers and/or destinations/consumers. Its aim is to facilitate the logical or semantic grouping of physical devices and applications.

The interface part for CRUD operations on twins follows the same logic as the interface part related to things. Firstly, you can list twins that you (or your user) have by simply clicking on the Twins option in the left hand side menu.
Twins list

You can create a new twin in the same way you create a new thing: click on the plus sign, enter twin’s name and click the check mark sign. By clicking on the magnifying glass icon, you go to the particular twin’s details page.

Twin’s details

Twin info card shows you the name and internal ID of the twin, owner and when it was created as well as when it was updated. To understand what twin update means, one needs to understand the anatomy of a twin. In a nutshell, the main part of a twin is it’s definition which consists of a set of attributes, where each attribute is uniquely defined by a combination of channel and subtopic.
Definition card shows you the current definition of a twin. By clicking on the button History, you get a chronological list of successive twin’s definitions.

**Twin’s definition**

The field persisted tells us whether an attribute is persisted in the database. This leads us to the question of twin’s state. Twin’s state is a snapshot in a certain point in time of a data agent represented by a twin. It stores the values of attributes in a certain point in time. The new snapshot occurs when any of the attributes change and if the time difference, delta, between the last snapshot and the snapshot being taken is greater than a set amount of time.

**State on the twin’s details page**

**Twin’s details**
shows us the current values of the attributes whether they are persisted in a database (currently only MongoDB is supported) or not. By clicking on the History button, you go to the chronological list of states:

**Twin’s states**

You see a total count of states and from and to fields, where you can enter the range of states (max range spans 100 states) to be displayed in the lower list. Each state has an ID - which is simply state’s ordinal number - and a Definition field corresponding to a certain revision number of a twin, i.e. to a certain definition of twin. The date refers to the time when the state snapshot was made. Finally, the column payload shows us the attribute names and their corresponding values.

In the bottom of the twin’s details page, you can find an attribute editor:
In the area outlined by rectangle 1, you can see an individual attribute editor. You can give a name, select a channel, enter a subtopic and select whether you want to persist an attribute. By clicking on the Add/Update Attribute button, you add an attribute to an attribute list (or update it, if the attribute was already there). An attribute list itself, the area outlined by the rectangle 2, shows you the list of attributes and their properties. This list represents a set of attribute that will compose the future (next) definition of a twin. You can also set the time difference delta (see above). By clicking on Update definition button, you update the twin’s definition, i.e. you turn the attribute list from the attribute editor into the current twin’s definition.
11 APPENDIX 5: POSTMAN

**Postman** is, amongst other, the API Client that enables you to send HTTP requests and receive HTTP responses. Postman enables you to do manually what your browser is doing automatically for you. This means that you can use Postman to gain a greater and fine grain control over AIP CRUD operations.

### 11.1 User authentication request

Let us now see how to create an HTTP request in Postman. Every Postman request is a part of a collection of requests. So we need to make a collection first.

**Postman Collection**

Once you make a collection, click on the _Add request s_ link and you will be presented with the request metadata (we will enter request HTTP data later) editor:
Postman Request

Enter the request name as “User authentication” and save the request to the collection. The first thing you want to do when interacting with the AIP is to get your user token. This token is then used in all other requests that you send to the AIP. Token is used by AIP to verify the authenticity of your requests. So let us set the Postman request that we have just made to request the user token. Go to the Params tab (it should be open by default), replace GET method by POST and enter the `iot.ashvin.eu/tokens` in the URL field.

Method selection

Every request that you make via Postman is directed towards a certain address. Every request addressed to the AIP begins with `iot.ashvin.eu/`. We use the POST method of request sending in order to post our username and password in the request body.
To set a request body, go to Body tab, select raw body and from the dropdown menu select JSON. Then enter

```json
{
   "email": "<email>",
   "password": "<password>"
}
```

in the body text field, replacing `<email>` and `<password>` with the email and password that you used to create a user (pay attention to keep " around your email and password in order to respect how JSON format specifies strings).

**Request body**

When your done, you are almost ready to send a request. You need to put the information in your request that you are sending a JSON content in the request body. Go to Headers tab and enter Content-type in the first row and the KEY column and application\json in the first row and the Value column.
Request headers

Now click on the Send button and, if everything went well, you should get something similar to this as a response body:

```json
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MDg2NzU5MDQsImlhdCI6MTYwODYzOTkwNCwiXNzIjoibWFpbmZsdXguyXV0aG4ilCIjdcIiIwZXRhci5wZXRyb3ZpY0BlbWFpbC5uZXQiLCJzdWIiOiJwZXRhci5wZXRyb3ZpY0BlbWFpbC5uZXQiLCJpc3N1ZXJfaWQiOiJhOWM1MWE1MC1jMzQwLTQ5YTc2NjczZDUwNmM3NTM1ZTQiLCJ0eXBlIjowfQ.B-I464fhsneLAOJZdjRgCJLi1aDvymVr1dpWAKw-Fg"
}
```

Your token is this long cryptic word, in our case

```text
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MDg2NzU5MDQsImlhdCI6MTYwODYzOTkwNCwiXNzIjoibWFpbmZsdXguyXV0aG4ilCIjdcIiIwZXRhci5wZXRyb3ZpY0BlbWFpbC5uZXQiLCJzdWIiOiJwZXRhci5wZXRyb3ZpY0BlbWFpbC5uZXQiLCJpc3N1ZXJfaWQiOiJhOWM1MWE1MC1jMzQwLTQ5YTc2NjczZDUwNmM3NTM1ZTQiLCJ0eXBlIjowfQ.B-I464fhsneLAOJZdjRgCJLi1aDvymVr1dpWAKw-Fg
```

We will use it to create our first thing. So note it down.

### 11.2 Postman environments

Better yet, let us use Postman itself to store the token (see above). This is a preferred alternative, since we will be able to reuse our stored token in multiple requests. When we start a new session and get a new token, we will simply update the value stored in Postman and we will not have to change any of our requests (see below).

Postman use environments to store logically related sets of values. To create an environment, click on the icon marked by 1 in the image below.
**Postman environment**

After that, click on the button Add (2 on the image above) and you will be presented with the following window:

Enter the environment name (1), enter token as a variable name (you can enter any name you like), fill in the initial and the current value of the variable - in our case, simply paste the token value in both fields (2) - , and finally click on the Add button (3).

To use your environment, simply select it from the dropdown list (1) and to visualize it and edit it, click on the eye icon button (2):
Postman environment

11.3 Create thing request

Now that we have set the environment, we can use our environment variable token to create another request. We will create a request that will instruct AIP to create a thing (a platform entity mostly used to represent a real world device or a software application). Add a new request to the collection:

Postman request

And give it a name Adds new thing (you can give it any name you like). Select POST as a
request method and enter iot.ashvin.eu/things in the URL text field. In the headers tab, you need to add two rows: one with the key Content-type and the value application/json and the other with the key Authorization and the value {{token}}. The {{variable_name}} syntax tells to Postman to look for the value of token in the currently chosen environment - the Ashvin docs environment, in our case.

**Postman request**

Finally, switch to the tab Body, select raw as a body type and select JSON as a body type specifier from the dropdown list. In the Body text field, enter the name and the metadata of the thing you want to create, e.g.

```json
{
    "name": "borer temperature sensor",
    "metadata": {
        "min heat": 15,
        "max heat": 55
    }
}
```

You can normally enter any kind of metadata you like as long as it follows JSON format. Also, the thing’s name is arbitrary.

We should be all set now. If you click the Send button, you should get a response with a Status: 201 Created. Of special interest are the response headers of the response:
Response headers

It is important to note the thing’s id which you get in the Location header. It’s a long number in a uuid format that follows the /things/ prefix, e.g. aa5edfa6-d362-462c-8270-4c0caff44aa8. Please note it down, or better yet, let’s add it to our Postman Ashvin docs environment. Click on the eye icon, and then on the button Edit and enter the variable name, initial and current value:

Postman environment
11.4 Retrieve thing request

Now we are ready to fetch our created thing and see its properties. In order to do so, create another request and name it, for example, Retrieve thing_info. Set requests method to GET. Our URL, `iot.ashvin.eu/things/:thingid`, is a bit unusual this time, because it contains `:thingid`. This means that this part of the URL will be replaced by the value we put in the parameter row with a key `thingid`.

**Parameters**

We will put our environment variable `{{thingid}}` in the value column. We need to put our token environment variable in the `Headers` tab as we did for the create thing request:

**Headers**
As explained (see above), this is needed in order to authenticate the platform user. And we are ready to send a request. By clicking on the button Send, you should get a response with a body similar to this one:

```
{
    "id": "aa5edfa6-d362-462c-8270-4c0caff44aa8",
    "name": "borer temperature sensor",
    "key": "635c2084-60da-491f-b448-4df5ab6abfef",
    "metadata": {
        "max heat": 55,
        "min heat": 15
    }
}
```

You can see your thing’s id, name, key and metadata.

### 11.5 Requests specifications

We have added three requests so far, so one might ask how do we know all the data needed to create requests. What is more, how do we know what requests we can make? This is a general question, since it is not related specifically to Postman. Postman is just one of the tools we can use to make HTTP requests addressed to the the Ashvin IoT Platform (AIP). (One of the popular choices is curl, an open-source CLI HTTP client.)

AIP is based on Mainflux, a “scalable, secure, open-source, and patent-free IoT platform”. Mainflux exposes a well defined HTTP API and that is the API that the Ashvin IoT platform is using. This API itself is described in the following files:

- users
- things
- twins

For a non-programmer, those files are rather cryptic. So, select the entire text in an individual file, copy it and head to the online Openapi specification editor. (Openapi is the name of the specification standard followed by the AIP (Mainflux) HTTP API specifications.) If everything went well, you should get a document similar to this one:
Openapi specification

All the possible requests you can make are listed there. You can now unfold any of the request in order to see the data needed to construct a request (in Postman, or in any other HTTP client, such as `curl`):

Openapi specification request

In the upper image, I have unfolded the request to add a new thing. We can see that request takes one parameter, `Authorization`. This parameter takes as a value our user token in the form of a `uuid` string. Furthermore, the request expects a JSON body with a following schema:
When we created a thing (see above), we have actually used the following JSON body:

```json
{
    "name": "borer temperature sensor",
    "metadata": {
        "min heat": 15,
        "max heat": 55
    }
}
```

We left out key field and we, thus, let AIP to generate a thing key for us.

### 11.5.1 Automatic request generation

Postman allows us to automate the process of request creation by importing directly Openapi specifications. Click on the Import button, switch to Link, enter one of the URLs from the upper list and click on the button Continue.

**Import Openapi specification**

Finally, click on the button Import and Close, and you should get a collection of requests generated based on the Openapi specification. These requests are more or less ready to use. You should go through each request you want to use and fill in the missing info, such as the URL prefix, your user token, thing id, etc.
11.5.2 Ashvin Collection

If you don't want to generate the requests yourself, you can go here and download the request collection and the associated environment.

![Screenshot of Postman interface]

**Github**

Click on the Download zip button, unpack the zip archive. In the ashvin folder you will find two files, Ashvin.postman_environment.json and Ashvin.postman_collection.json. In order to import a collection, click on the Import button and then on the Upload files button.

![Screenshot of Postman interface]

**Import Openapi specification**
Select `Ashvin.postman_collection.json` and follow the procedure. In order to import an environment, click on the environments button and then on the `Import` button.

**Import Openapi environment**

Click on the `Choose Files` button, select `Ashvin.postman_environment.json` and follow the procedure.

You use Ashvin collection in conjunction with the Ashvin environment so be sure to select Ashvin environment from the environments dropdown list in order to activate it. Once you select environment, be sure to fill in the environment variables with your desired values for the URL prefix, your user token, thing id, etc.
12 APPENDIX 6: MOSQUITTO

12.1 MQTT client

Eclipse Mosquitto is an open source message broker that implements the MQTT protocol. Since the Ashvin IoT platform uses an MQTT message broker, amongst other brokers, to send and receive messages, You can use the AIP to send and receive messages over MQTT. Eclipse Mosquitto features mosquitto_pub and mosquitto_sub command line MQTT clients for sending and receiving messages over MQTT.

Since mosquitto_pub and mosquitto_sub are command line tools, you will need a terminal (or a console) in order to send and receive messages. Every major operating system comes with a terminal pre-installed, so you do not need to worry about that. You can find the installation instructions for mosquitto message broker here. This will also install mosquitto publish and subscribe clients, i.e. mosquitto_pub and mosquitto_sub.

12.2 Subscribe and publish

In order to publish messages via MQTT you need at least one thing and one channel. For the sake of clarity, we will use two things: one will publish messages to a channel and the other will listen for these messages on the same channel (please do note that the same thing can be used to publish and listen for messages on the same channel). Publishing and listening things need to be connected to the channel that will be used for the message transmission. To create things and channels and connect them, you can use an the AIP dedicated GUI or any other HTTP client, e.g. Postman.

So, now you have two things - one that will publish messages over mqtt, and the other that will receive them - and a channel used for message transmission. Both things need to be connected to the channel. Please note down both things’ ID and key as well as the channel’s ID (channels do not have key since they are used to route messages and not to authenticate senders and receives). We will need to use things’ IDs and keys as means of authentication and the channel’s ID to construct an MQTT topic.

To actually receive messages, we need first to subscribe to a channel. In order to do it, you use this general format of command:

```
mosquitto_sub -u <thing_id> -P <thing_key> -t channels/<channel_id>/messages -h iot.ashvin.eu
```

Where you replace words in <> (together with <>) with concrete values. For example,

```
mosquitto_sub -u 5b8e259c-f69e-4da3-8cf8-266b4ad18d98 -P d87c90b7-0000-4774-b2ab-0900f122a3e2 -t channels/44c669a4-7ebc-40c8-ab6d-5bf9ac8398c5/messages -h iot.ashvin.eu
```

Please do notice how we do not use user’s credentials in order to subscribe to a channel. Rather, we use thing’s credentials, i.e. thing’s ID and key. We do it in order to see whether a thing with the given credentials is authorised to publish and
subscribe to a given channel. In the terminology of the AIP, we check whether the thing is connected to a given channel.

To publish a message over a channel, call the following command:

```
mosquitto_pub -u <thing_id> -P <thing_key> -t channels/<channel_id>/messages -h localhost -m '[{"bn":"some-base-name:","bt":1.276020076001e+09, "bu":"A","bver":5, "n":"voltage","u":"V","v":120.1}, {"n":"current","t":-5,"v":1.2}, {"n":"current","t":-4,"v":1.3}]'
```

For example,

```
mosquitto_pub -u aea9e669-2526-495b-bcd4-7d3d26c65f5d -P d07541b9-3089-48c5-b914-4bb90e0d9b5e -t channels/44c669a4-7ebc-abcd-5bf9ac8398c5/messages -h iot.ashvin.eu -m '[{"bn":"some-base-name:","bt":1.276020076001e+09, "bu":"A","bver":5, "n":"voltage","u":"V","v":120.1}, {"n":"current","t":-5,"v":1.2}, {"n":"current","t":-4,"v":1.3}]'
```

Mosquito

In the upper pane of the console, I have first subscribed to a channel. Afterwards, in the lower pane of the console, I have sent a message. Finally, in the upper pane of the console, in the second line, I see that the message has successfully arrived and is echoed in the console.

12.3 SenML

In order to understand the structure of the message

```
["bn":"some-base-name:","bt":1.276020076001e+09, "bu":"A","bver":5, "n":"voltage","u":"V","v":120.1}, {"n":"current","t":-5,"v":1.2}, {"n":"current","t":-4,"v":1.3}]
```
we have sent and received with the upper commands, we need to get acquainted with the SenML. SenML is a format used for representing sensor measurements and device parameters. For example, a temperature sensor, could use it to send the measurements or receive the configuration.

Ashvin IoT platform uses preferably SenML format to send, receive and persist messages although an arbitrary formatted JSON message can also be used.

SenML message is an array with a series of measurements. In our upper example, we have three measurements. bt refers to the time offset of each measurement, so in order to get a time of a particular measurement, we add t (represented in floating point as seconds) to the bt (itself expressed in Unix time also as a floating point seconds number). v is a value field used to store a float type value. There are other fields for values of other data types: vs for a string value, vb for a boolean value and vd for a binary value. Here is a full table of an individual SenML measurement fields:

<table>
<thead>
<tr>
<th>Name</th>
<th>Label</th>
<th>CBOR Label</th>
<th>JSON Type</th>
<th>XML Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Name</td>
<td>bn</td>
<td>-2</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Base Time</td>
<td>bt</td>
<td>-3</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>Base Unit</td>
<td>bu</td>
<td>-4</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Base Value</td>
<td>bv</td>
<td>-5</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>Base Sum</td>
<td>bs</td>
<td>-6</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>Base Version</td>
<td>bver</td>
<td>-1</td>
<td>Number</td>
<td>int</td>
</tr>
<tr>
<td>Name</td>
<td>n</td>
<td>0</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Unit</td>
<td>u</td>
<td>1</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Value</td>
<td>v</td>
<td>2</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>String Value</td>
<td>vs</td>
<td>3</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Boolean Value</td>
<td>vb</td>
<td>4</td>
<td>Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>Data Value</td>
<td>vd</td>
<td>8</td>
<td>String (*)</td>
<td>string (*)</td>
</tr>
<tr>
<td>Sum</td>
<td>s</td>
<td>5</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>Time</td>
<td>t</td>
<td>6</td>
<td>Number</td>
<td>double</td>
</tr>
<tr>
<td>Update Time</td>
<td>ut</td>
<td>7</td>
<td>Number</td>
<td>double</td>
</tr>
</tbody>
</table>

(Source: tools.ietf.org)