



## International Partnership on Innovation

### SAMS - Smart Apiculture Management Services

Deliverable N°4.1

#### Report on Data Management

Work package 4 Decision Support System

Horizon 2020 (H2020-ICT-39-2017)

Project N°780755











This project has received funding from the European Union's Horizon 2020 research and innovation programme under **grant agreement N° 780755**. The sole responsibility for the content of this document lies with the authors. It does not necessarily reflect the opinion of the EU.

Project information		
<b>Lead partner for the deliverable</b>	Latvia University of Life Sciences and Technologies, Faculty of Information Technologies, Department of Computer Systems. Lead researcher and Project lead: Asoc.prof. Dr. Aleksejs Zacepins	
<b>Document type</b>	Report	
<b>Dissemination level</b>	Public	
<b>Due date and status of the deliverable</b>	31.08.2019	<i>Incl. status/Date of upload</i>
<b>Author(s)</b>	Dr. Aleksejs Zacepins, Dr. Vitalijs Komasilovs, Armands Kviesis, Olvija Komasilova	
<b>Reviewer(s)</b>	GIZ	

This document is issued by the consortium formed for the implementation of the SAMS project under Grant Agreement N° 780755.

### SAMS consortium partners

Logo	Partner name	Short	Country
	Deutsche Gesellschaft für Internationale Zusammenarbeit (GIZ) GmbH (Coordinator)	GIZ	Germany
	University of Kassel	UNIKAS	Germany
	University of Graz (Institute for Biology)	UNIGRA	Austria
	Latvia University of Life Sciences and Technologies	UNILV	Latvia
	ICEADDIS – IT-Consultancy PLC	ICEADDIS	Ethiopia
	Oromia Agricultural Research Institute, Holeta Bee Research Center	HOLETA	Ethiopia
	University Padjadjaran	UNPAD	Indonesia

	Commanditaire Vennootschap (CV.) Primary Indonesia	CV.PI	Indonesia
-----------------------------------------------------------------------------------	-------------------------------------------------------	-------	-----------

## List of Abbreviations

---

API	Application Programming Interface
Back-end	Data access layer (server side) of an application
Collection	MongoDb stores documents (records) in collections. Collections are analogous to tables in relational databases
DBMS	Database Management System
DSS	Decision Support System
DW	Data Warehouse
Front-end	Presentation layer (usually Web based) of an application
HIVE	HIVE measurement system (WP3)
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token, an open, industry standard RFC 7519 method for representing claims securely between two parties.
OAuth 2.0	Open standard for access delegation. OAuth 2.0 provides specific authorization flows for web applications, desktop applications, mobile phones, and smart devices.
POST	HTTP request method
RegExp	Regular expression, sequence of characters that define a search pattern.
REST	Representational State Transfer
SQL	Structured Query Language
UI	User Interface

## Summary of the project

---

SAMS is a service offer for beekeepers that allows active monitoring and remote sensing of bee colonies by an appropriate and adapted ICT solution. This system supports the beekeeper in ensuring bee health and bee productivity, since bees play a key role in the preservation of our ecosystem, the global fight against hunger and in ensuring our existence. The high potentials to foster sustainable development in different sectors of the partner regions are they are often used inefficient.

### Three continents - three scenarios

(1) In Europe, consumption and trading of honey products are increasing whereas the production is stagnating. Beside honey production, pollination services are less developed. Nevertheless, within the EU 35% of human food consumption depend directly or indirectly on pollination activities.

(2) In Ethiopia, beekeepers have a limited access to modern beehive equipment and bee management systems. Due to these constraints, the apicultural sector is far behind his potential.

(3) The apiculture sector in Indonesia is developing slowly and beekeeping is not a priority in the governmental program. These aspects lead to a low beekeeper rate, a low rate of professional processing of bee products, support and marketing and a lack of professional interconnection with bee products processing companies.

Based on the User Centered Design the core activities of SAMS include the development of marketable SAMS Business Services, the adaption of a hive monitoring system for local needs and usability as well as the adaption of a Decision Support System (DSS) based on an open source system. As a key factor of success SAMS uses a multi stakeholder approach on an international and national level to foster the involvement and active participation of beekeepers and all relevant stakeholders along the whole value chain of bees.

The aim of SAMS is to:

- enhance international cooperation of ICT and sustainable agriculture between EU and developing countries in pursuit of the EU commitment to the UN Sustainable Development Goal (SDG N°2) “End hunger, achieve food security and improved nutrition and promote sustainable agriculture”
- increases production of bee products
- creates jobs (particularly youths/ women)
- triggers investments and establishes knowledge exchange through networks..

## Project objectives

---

The overall objective of SAMS is to strengthen international cooperation of the EU with developing countries in ICT, concentrating on the field of sustainable agriculture as a vehicle for rural areas. The SAMS Project aims to develop and refine an open source remote sensing technology and user interaction interface to support small-hold beekeepers in managing and monitoring the health and productivity in their own bee colonies. Highlighted will be especially the production of bee products and the strengthening of resilience to environmental factors.

- Specific objectives to achieve the aim:
- Addressing requirements of communities and stakeholder
- Adapted monitoring and support technology
- Bee related partnership and cooperation
- International and interregional knowledge and technology transfer
- Training and behavioural response
- Implementation SAMS Business cooperation

## Contents

---

SAMS consortium partners.....	2
Summary of the project .....	4
Project objectives .....	4
Contents.....	6
List of tables / figures .....	6
<b>1. Background .....</b>	<b>8</b>
1.1 Scope of the Deliverable .....	8
<b>2. Data Warehouse Concept .....</b>	<b>8</b>
<b>3. Database management system .....</b>	<b>10</b>
<b>4. Data flows .....</b>	<b>11</b>
<b>5. User Interface .....</b>	<b>13</b>
<b>6. WebApi.....</b>	<b>14</b>
6.1 Nodes .....	14
6.2 Sensor mapping .....	16
6.3 Authentication and authorization .....	19
<b>7. DW Core.....</b>	<b>22</b>
7.1 Swamp.....	23
7.2 Vaults.....	23
7.3 Reports .....	27
7.4 Core internal components .....	29
<b>8. Deployment components and infrastructure .....</b>	<b>30</b>
<b>9. Further development.....</b>	<b>32</b>

---

## List of tables / figures

---

Figure 1. SAMS Data warehouse architecture .....	9
Figure 2. Demonstration of hardware data-in package schema .....	11
Figure 3. Example of WEB system dashboard.....	14
Figure 4. Different types of DW nodes .....	14
Figure 5. Example of user nodes in the system .....	15
Figure 6. Metadata of the user node .....	15
Figure 7. Example of sensor mapping.....	17
Figure 8. An example of node with its source mapping .....	18
Figure 9. Example of non-interactive flow for device authentication .....	20
Figure 10. Auth0 login page .....	21
Figure 11. Example of interactive flow for authentication.....	21
Table 1. Summary of potential records for different time periods .....	24
Figure 12. List of available reports .....	28

Figure 13. Report creation.....	28
Figure 14. Example of the raw temperature measurements .....	28
Figure 15. Conceptual diagram of the DW Core .....	30
Figure 16. Deployment structure of the DW .....	31

# 1. Background

The Latvia University of Life Sciences and Technologies develops the Data Warehouse (DWH) system within the SAMS project. The DW is important part of the project as acts as a data storage and analysis unit for all beehive data collected by the SAMS HIVE measurement systems developed by the University of Kassel (UNIKAS). All collected data is sent from the HIVE systems via Wi-Fi / Internet to the data warehouse system for storage and further processing. At the end, processed and analysed data will be used to support local beekeepers with information about the bee colony health status, productivity level and inform about some deviations from normal colony state.

## 1.1 Scope of the Deliverable

This report describes the development process of the SAMS data warehouse from all the beginning to the functioning prototype. As well data management procedures within the DW are described in this report.

# 2. Data Warehouse Concept

The following chapter explains the conceptual design of data management solution for the SAMS project. The solution is designed upon several major functional requirements:

- receive data from a various hive measurements systems (hardware configurations);
- store and process received data according defined rules;
- provide data output facilities.

In general computing systems a **data warehouse** (DW) can be considered as a universal system, which is able to operate with different data inputs and have flexible data processing algorithms. By the definition data warehouse is like an intermediate layer between data provider systems and data consumer systems or end-users. DW provides customizable facilities for data storage management, processing, analysis and output.

Within SAMS project DW is developed with aim to help beekeepers run the apiary more effectively by utilising higher amount of available data and accumulated data interpretation knowledge.

Architecture of the developed DW is demonstrated in Figure 1.



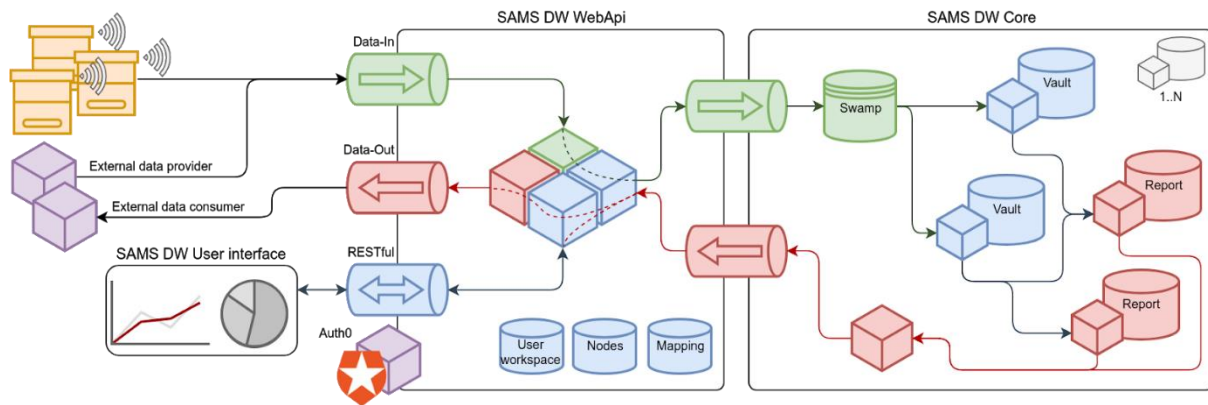


Figure 1. SAMS Data warehouse architecture

DW consists of three modules:

1. **Core** – main data storage and processing module; it receives data about various beekeeping objects in predefined format and distributes it through number of vaults and reports, which apply needed transformation to the data (e.g. aggregation, modelling, decision making);
2. **WebApi** – intermediary module between “outer world” and DW Core; it provides number of HTTP interfaces for machine-to-machine interaction with external systems via Internet; main functions of the unit include request authentication and authorization, user private workspace management, data-in and data-out interface configuration and data conversion to/from DW Core supported formats;
3. **Graphical user interface** – single-page web application provides user convenient way for managing the sources of incoming data (e.g. hives with measurement devices) and getting insights into produced outputs (e.g. reports).

Architecture of DW is developed considering flexibility and extensibility of each main data management stage:

- data **input** functionality from various data sources, for example data files uploaded manually via custom user interface or via automated (scheduled) scripts, a hive measurement system configured to send data in accepted format, or third-party services providing needed data (e.g. weather station);
- data **storage** suitable for different measurement types, for example temperature, weight or audio recordings, intermediate aggregation and modelling results; and suitable for different data time granularity, for example minutely, hourly or daily recordings;
- data **processing** is organized using modular approach which allows building flexible aggregation and modeling pipelines, where raw incoming data is pushed through a number of transformation steps resulting in useful derivations and reports for end users.

DW is built in the way that incoming data are processed almost immediately by involving different models for data aggregation and reporting. Modular architecture of the solution ensures isolation boundaries both for reliability reasons, maintenance and development considerations.

Mentioned modules are described in details in the following chapters of the document.

### 3. Database management system

SAMS data warehouse is essentially a solution for beekeeping data recording, transformation and reporting according to developed rules and schemas. However low level data storage topic is out of the project scope (e.g. data organization on disks). For the purpose of such low level data storage ready database management system (DBMS) should be used.

UNILV previously had experience with hive temperature recordings and used MySQL DBMS as a data storage. Overall the experience with MySQL can be considered successful, the solution recorded data about local test apiary for 7 years. However few shortcomings arose during these years.

- Over the years the volume of raw measurement table have significantly grown. Taking into account data model (normalized, one record per measurement) and indexing needs database maintenance was complicated and performance degraded. It was improved to some extent by introducing table partitions and optimized vendor-specific SQL queries.
- Majority of data usage cases (data-out queries) were oriented on aggregated values (e.g. temperature changes over a week, last measurements from particular device, etc). Due to data model such queries were not effective and workarounds with triggers were introduced.
- Data model was designed to store measurements coming from sensors (which represent particular hives). However over the years due to hardware failures or apiary relocations sensors were changed between devices and hives, leading to overcomplicated and untrackable device-sensor-hive data model.

Database management system for SAMS DW was selected taking into account previous experience with strict SQL DBMS and addressing the aforementioned issues. UNILV had seen reasonable use-case for NoSQL DBMS, and in particular selected MongoDB document oriented database as a primary storage for SAMS DW. Considered advantages of this MongoDB database (and NoSQL technology in general) were as follows:

- data denormalization is usual practice for NoSQL databases, which opens a wide range of options for data pre-aggregations and reports;
- document oriented data storage from one hand does not have strict schema and can be flexibly changed over prototyping cycles, and from another hand it allows more efficient record organization for given SAMS use-case (more details in next chapters);
- potentially MongoDB can be shared across multiple hosts in case database grows rapidly.

From development project (source code) organization perspective whole data storage and processing logic is implemented within the application making development and deployment process faster and easier, for example, no need for data schema definitions and custom stored procedures for DBMS, possibility to run embedded DBMS for integration tests, less deployment units during releasing process.

## 4. Data flows

As mentioned before SAMS DW consists of three main modules: *Core*, *WebApi* and *User Interface*. SAMS *HIVE* measurement system (developed within WP3, responsible partner UNIKAS) also can be considered as an additional external module.

All these modules are developed as independent units and use different technology stacks. Therefore communication between these components should be platform independent. SAMS DW solution follows RESTful architecture style and relies on HTTP communication between its modules. For the message payload JSON notation is used. These technologies are widely used in modern Web systems and are supported by all major development frameworks and libraries.

DW data-in interface provides data input functionality for various data providers, mainly focusing on the *HIVE* measurement system. It should be configured accordingly and sends data in accepted format. Received data is validated and transformed within *WebApi* module according to user configuration and then forwarded to *Core*.

For the sake of communication efficiency and to address offline operation cases for *HIVE* hardware data-in package schema is as follows:

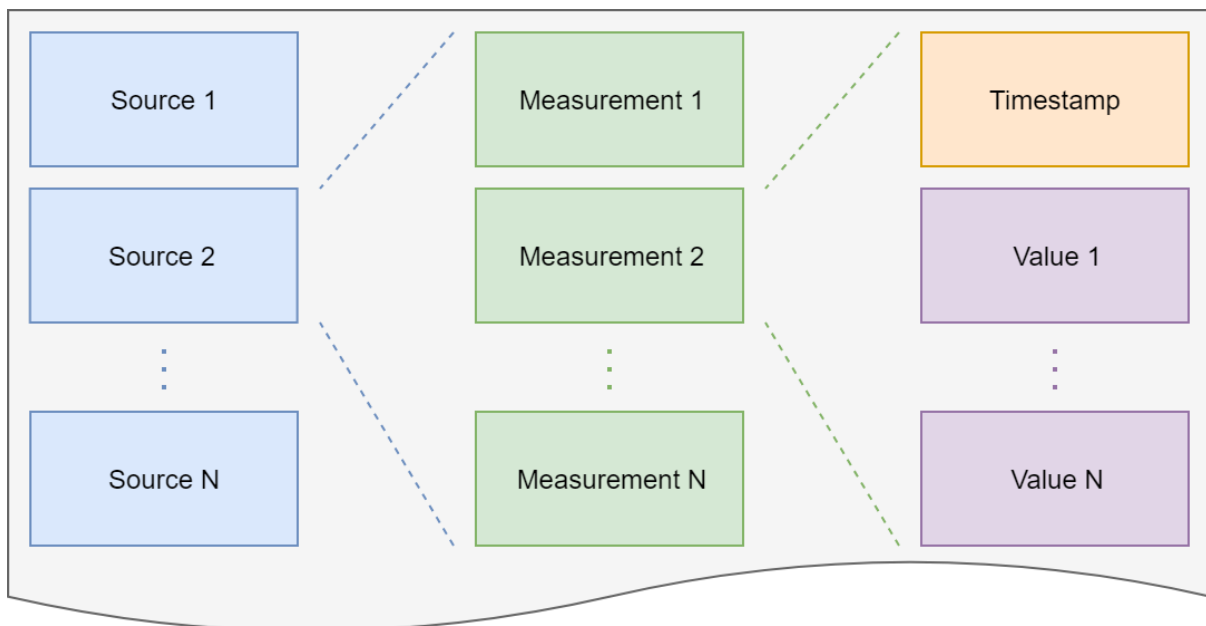


Figure 2. Demonstration of hardware data-in package schema

Such schema allows *HIVE* devices to send readings from multiple sources (usually sensors) during a single communication session. Single data package supports multiple measurements for each source, where each measurement has timestamp and one or more numeric values. This approach makes it possible for *HIVE* devices to operate in offline mode, perform sensor measurement recordings in needed frequency and communicate with DW according most efficient schedule or when connection is available.

An example of data-in package is provided below:

```
[
  {
    "sourceId": "temp-sensor-123",
```

```

        "values": [
            {
                "ts": "2019-08-13T10:15:00Z",
                "value": 32.4
            },
            {
                "ts": "2019-08-13T10:25:00Z",
                "value": 33.1
            }
        ]
    },
    {
        "sourceId": "audio-sensor-234",
        "values": [
            {
                "ts": "2019-08-13T10:17:00Z",
                "values": [0.24, 0.09, 1.42, 0.92, 0.86]
            },
            {
                "ts": "2019-08-13T09:11:00Z",
                "values": [0.19, 0.35, 1.19, 1.25, 0.53]
            }
        ]
    }
]

```

Communication link between *WebApi* and *Core* modules is expected to be more stable and reliable (single host or hosts within the same data center), therefore package schema for data exchange between these modules is straight forward. Data about each object and particular parameter is sent separately, however measurements are accepted in batches.

```

{
    "objectId": "hive-987",
    "type": "temperature",
    "values": [
        {
            "ts": "2019-08-13T10:15:00Z",
            "value": 32.4
        },
        {
            "ts": "2019-08-13T10:25:00Z",
            "value": 33.1
        }
    ]
}

```

Such package schema is flexible enough for online data streams as well as for importing historical data from files (large batches of measurements about single object).

DW data-out interface is designed to provide data output functionality for external data consumers. Currently this data flow direction is utilized only by *User interface* for reports, but potentially can be used by any other user or unmanned system as it works upon HTTP protocol and JSON notation. Report is considered as a collection of numerical values about given object mapped to time axis (e.g. raw parameter measurements, aggregated or modelled values, etc).

Report requests are proxied by *WebApi* module and forwarded to *Core* module without intermediate transformations. Report response data package is designed for processing values in column-wise approach and mainly used (but not limited) for plotting timeline charts. It provides basic report metadata, supports multiple data series and auxiliary parameters. An example of report data package is provided below:

```

{
    "code": "temperature",
    "name": "Raw temperature measurements",
    "aux": {
        "c1": 1.521,
        "ha7": 1563.528,
    },
    "data": [
        {
            "name": "timestamp",

```

```

        "values": [
            "2019-08-07T04:36:00Z",
            "2019-08-07T05:36:00Z",
            "2019-08-09T08:36:00Z",
            "2019-08-09T11:06:00Z",
            "2019-08-09T12:06:00Z"
        ]
    },
    {
        "name": "hive-111-top",
        "values": [
            22.812,
            23.312,
            24.125,
            25.5,
            25.437
        ]
    },
    {
        "name": "hive-111-bottom",
        "values": [
            21.672,
            22.113,
            21.257,
            22.645,
            23.047
        ]
    }
]
}

```

In addition there are a number of data flows supporting user interface module but they are out of the scope of current deliverable.

## 5. User Interface

By design DW provides REST API for all configuration, data-in and data-out operations. It can be used by any user or unmanned system, but primarily was created for custom user interfaces (e.g. localized mobile and Web applications).

For demonstration purposes and internal scientific usage a single page Web application was created (front-end) by UNILV. It provides a graphical interface for DW configuration maintenance, for accessing stored information and generate different reports. For front-end development Angular and Bootstrap frameworks were used.

Front-end development for DW is not required for particular deliverable, however data examples and its peculiarities shown in relevant parts of the report are based on developed user interface.

An example of dashboard shows oversee of Ethiopian apiaries and active devices and sensors:

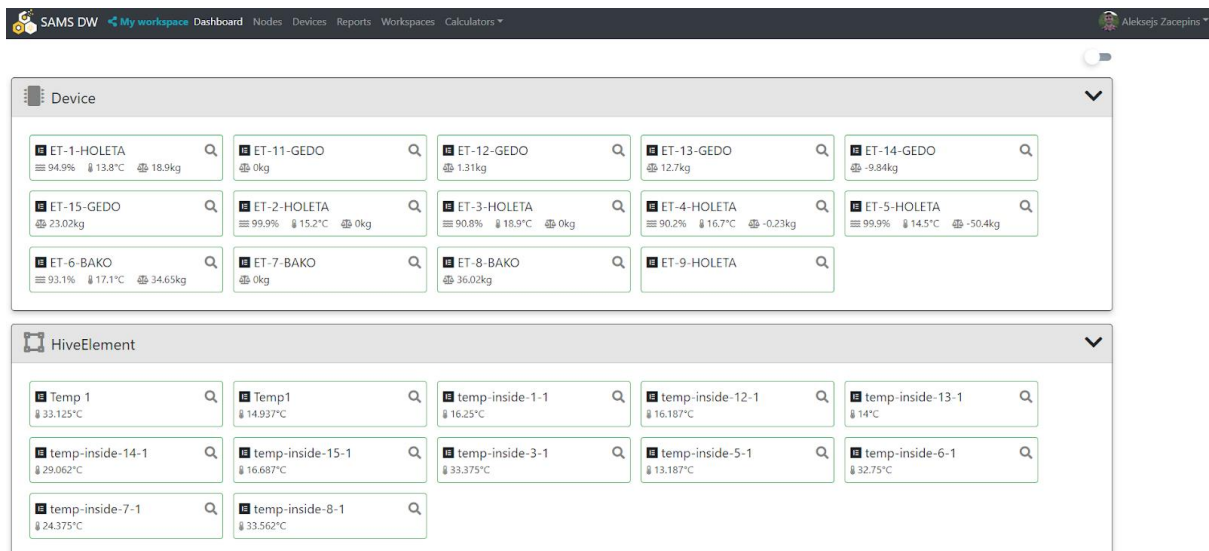


Figure 3. Example of WEB system dashboard

## 6. WebApi

WebApi component provides an access to DW for external systems and components via REST API. Its primary function is to transform incoming data-in requests from HIVE devices to corresponding requests for Core component. It provides flexible and extendable platform for data-in flow customization and configuration.

### 6.1 Nodes

Concept of Node in the context of SAMS DW stands for any logically distinct unit relevant for user in terms of parameter measurements and storage. Main purpose of nodes is to provide flexible abstraction layer between beekeeping objects (like hives and apiaries) and monitoring infrastructure (sensors and devices). Currently DW supports (but not limited to) following types of nodes: Group, Apiary, Hive, Hive element, Device and Other.

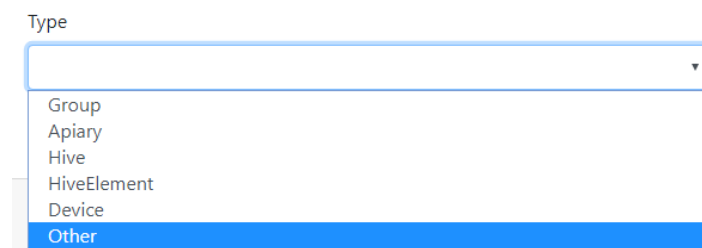


Figure 4. Different types of DW nodes

Nodes allow users to define the structure of his beekeeping objects (from a business perspective) and configure relevant data sources for these objects. System supports building of arbitrary hierarchy of nodes, which opens wide possibilities for structuring and organizing beekeepers objects of interest. Example of user nodes in the system is shown below.

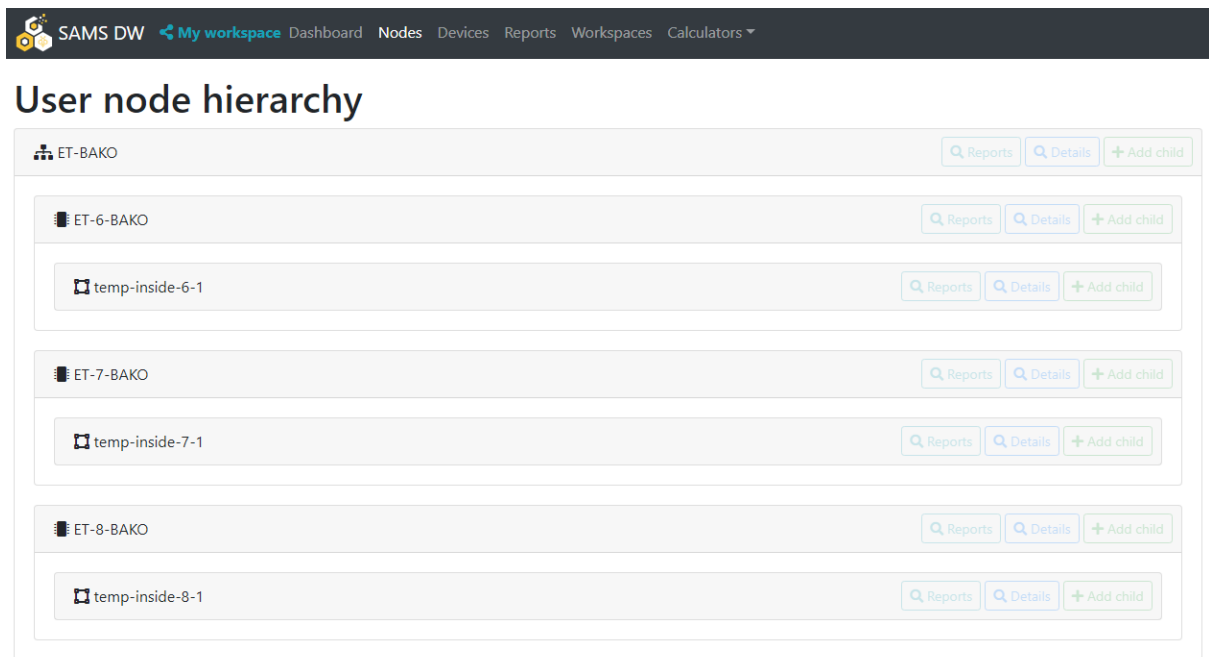


Figure 5. Example of user nodes in the system

Nodes support basic metadata, like name, type and free text location, and can be extended with more attributes:

Figure 6. Metadata of the user node

Device type nodes have additional attribute Client ID used for coupling with *HIVE* hardware. This ID is used as a device identifier and *WebApi* accepts measurements only from registered devices.

Information about all nodes is stored in dedicated *collection*. In addition to metadata each node record contains also references to its parent nodes (if any), owner username and workspace. For debugging and monitoring purposes nodes are supplemented with last 10 measurements for each parameter received through data-in interface for particular node. An example of node record is as follows:

```
{
  "_id" : ObjectId("5c923b605d71560009a76674"),
  "name" : "Hive A1",
  "type" : "HIVE",
  "parentId" : "5c922e555d71560009a76381",
  "location" : "LV Lab",
  "ancestors" : [
    "5c922e1d5d71560009a7637f",
    "5c922e3f5d71560009a76380",
    "5c922e555d71560009a76381"
  ],
  "lastValues" : {
    "temperature" : [
      ...
    ]
  },
  "createdBy" : "auth0|5c922df82e67ef322347721e",
  "workspaceId" : "5d09d674c41479000914d065"
}
```

Latest 10 measurements are maintained using MongoDB features allowing complicated conditional updates. Update query for pushing latest values looks similar to following:

```
db.nodes.update(
  {
    "_id": "hive-123"
  },
  {
    "$push": {
      "lastValues.temperature": {
        "$each": [{
          "ts": "2019-08-19T10:05:19Z",
          "value": 22.14
        },
        {
          "ts": "2019-08-19T10:15:25Z",
          "value": 23.56
        }
      ],
      "$sort": {
        "ts": -1
      },
      "$slice": 10
    }
  }
)
```

In essence this operation takes raw *timestamp-value* pairs, pushes them to measurement parameter array (temperature in the example), then sorts the array by timestamp in descending order and slices 10 values. All of these manipulations are performed in a single atomic operation.

## 6.2 Sensor mapping

By design DW data storage is organized by nodes and their parameters – it means that data about each node is stored separately and consistently regardless of the source of this data.

Such design is intentional and is based on UNILV experience running hive monitoring systems over the years – infrastructure of measurement system tends to change over time. There are reasonable use cases when sensors are replaced and reallocated between hives, devices are moved between apiaries, composition of *device-sensor-hive* chains tend to change in time.



Reconfiguring hardware on every infrastructure change is not feasible because it needs beekeeping site visiting for hardware specialist. Therefore a layer of configuration is added to *WebApi* for addressing such use cases.

*WebApi* uses sensor mapping concept. It implies that hardware solution (e.g. *HIVE* system) provides measurements identified by distinct sources (usually sensors), and *WebApi* uses its configuration to map these measurements to particular beekeepers object like hives or their elements.

This solution covers not only the simplest cases, when single device corresponds to single hive, but also more diverse configurations like shown on the picture below:

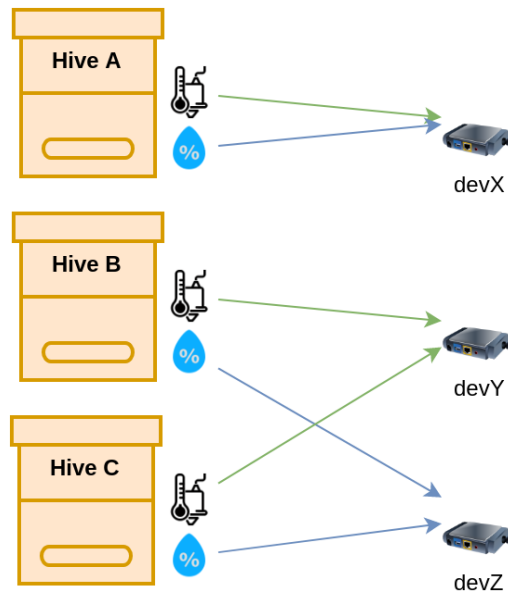


Figure 7. Example of sensor mapping

Corresponding source mapping would be as follows:

Source	Node	Key
temp@devX	Hive A	temperature
hum@devX	Hive A	humidity
temp1@devY	Hive B	temperature
temp2@devY	Hive C	temperature
hum1@devZ	Hive B	humidity
hum2@devZ	Hive C	humidity

Source mappings are stored in separate collection as documents like the following example:

```
{
  "_id" : "hum-LV-307",
  "valueKey" : "humidity",
  "nodeId" : "5c6bb6885d71560009a76372"
}
```

Collection has number of unique indexes ensuring data consistency:

- `_id` – default unique index ensures that the sensor is assigned to single node;
- `nodeId`, `valueKey` – composite unique index eliminates contradictions by ensuring that each node has only one source of particular parameter.

An example of node with its source mapping as visible in UI:

All nodes / ET-BAKO / ET-7-BAKO / temp-inside-7-1

temp-inside-7-1 /hive\_element/

ID: 5d3d4f6bc414790009e74ce1

Source mapping:

dsb18b20-0-TTdYF5c1OJQwElsdSMkgPQDK62	temperature	
---------------------------------------	-------------	--

+ Add mapping

Name: temp-inside-7-1

Location: Bako

Type: HiveElement

Save Reset

Figure 8. An example of node with its source mapping

Format of source ID is not fixed and *WebApi* accepts any arbitrary string provided by data provider implying the string is globally unique (like GUID). For demonstration purposes and easier debugging *HIVE* hardware uses short sensor name suffixed with device ID as a source ID.

As a side effect source mapping concept addresses several security cases:

- single source can be assigned only to one node eliminating data leaks (if user mapped his source to his node, nobody else will be able to map the same source to another node);
- data-in requests from unrecognized sources are rejected preventing data pollution and database bloating with dummy information (only registered data sources are accepted for processing, everything else is discarded).

Combination of node, device and source registration from one hand creates additional configuration burden for the DW user, but from other hand ensures strong security. For better DW end-user experience common registration scenarios can be automated (e.g. registration of *HIVE* hardware with typical sensor configuration).

During data-in request handling, *WebApi* reads source ID from incoming data package, tries to find matching node in source mapping configuration, and if successful, composes Core data-in package with appropriate node ID and parameter (value key). This approach significantly simplifies infrastructure reconfiguration – in case any element of *device-sensor-hive* chain has changed, user has to reassign sources to proper nodes without the need to reconfigure hardware.

## 6.3 Authentication and authorization

On a high level *WebApi* acts as a proxy between DW core and external Internet. In addition to described node, device and source mapping configurations *WebApi* also performs user and device authentication and authorization. As *WebApi* is designed to be used by external modules (like Web based user interface or HIVE hardware solution) it relies on OAuth 2.0 access delegation standard, where users delegate their access to DW for these external modules.

SAMS DW solution uses authentication and authorization services provided by *Auth0* universal platform (<https://auth0.com/>). *Auth0* is ready to use platform with wide range of built-in authorization related functionality and integration options. In particular SAMS DW uses specific authorization flows for Web and non-interactive applications, completely delegates user credential handling and access administration functionality to the platform. From development perspective usage of *Auth0* platform removes the need to create custom solution for secure user credential storage, user experience for Sign-in and Log-in flows, simplifies administration tasks.

Technically OAuth 2.0 authorization and authentication relies on access tokens which are provided on every request to REST API. Access tokens are issued to external clients by an authorization server and usually contain basic information about user and means for token validation. In his case *Auth0* authenticates user or device and issues JWT access token which contains user ID (subject) and expiration time details. Cryptographic signature of the token ensures that it is issued by authorized party. Such access token is sent to *WebApi* on every request and is validated by checking expiration time and signature.

### Non-interactive clients

*HIVE* device authentication and authorization is performed using non-interactive flow. This flow is designed specially for machine-to-machine interaction (*HIVE* device to *WebApi* in this case) and imply that client side is considered as “trusted”. Technically it means that each device is configured with individual credentials (client ID and secret). Then according to the schedule device exchanges these credentials for the access token in non-interactive mode. Obtained token is used for sending data-in packages until it is expired (by default 24 hours) and has to be reviewed again. Following sequence diagram shows this scenario:

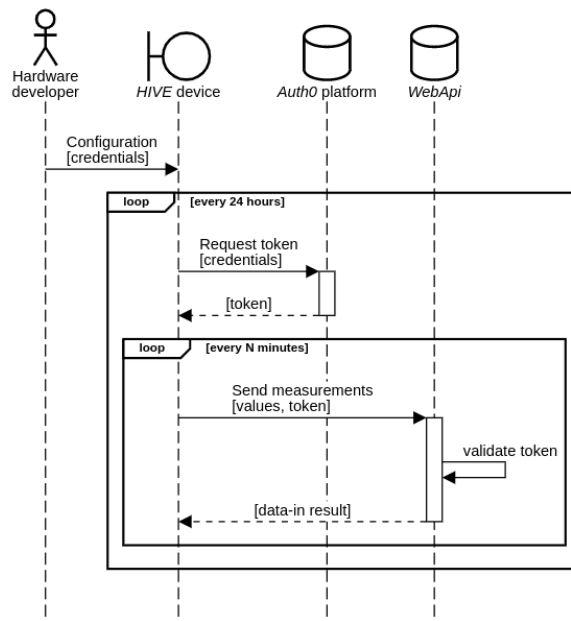


Figure 9. Example of non-interactive flow for device authentication

For additional security *WebApi* restricts requests from non-interactive clients (*HIVE* devices) only to data-in flow, all other endpoints available only for interactive clients described below.

Access token is included into header of data-in request as follows:

```

POST /api/data HTTP/1.1
Host: sams.science.itf.llu.lv
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IklVSTRNRGc....
Content-Type: application/json
  
```

```

[
  {
    "sourceId": "temp-id-123",
    "values": [
      {
        "ts": "2018-10-10T23:06:00Z",
        "value": 33.2
      }
    ]
  },
  {
    "sourceId": "hum-id-234",
    "values": [
      {
        "ts": "2018-10-10T23:06:00Z",
        "value": 42.8
      }
    ]
  }
]
  
```

## Single-page Web applications

Interactive single-page Web applications (the demonstration UI) are considered as “untrusted” clients and credentials can’t be safely stored in them. Therefore upon Log-in users are forwarded to Auth0 platform hosted Web page, where they have options to Sign-in or Log-in using their credentials or social media accounts. After successful authentication user is forwarded back to SAMS DW user interface, which acquires access token for given user session (by default 3 hours). Below the Auth0 hosted login page is demonstrated:

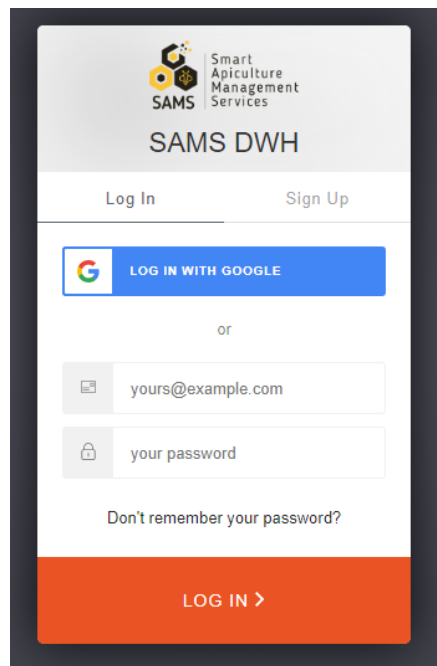


Figure 10. Auth0 login page

The access token is not persistently stored anywhere and is read from internal application variable until it is closed. Access token is included into header of any request to *WebApi* as follows:

```
GET /api/nodes HTTP/1.1
Host: sams.science.itf.llu.lv
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IklVSTRNRGc....
Content-Type: application/json
```

Following sequence diagram shows interactive authentication and authorization flow:

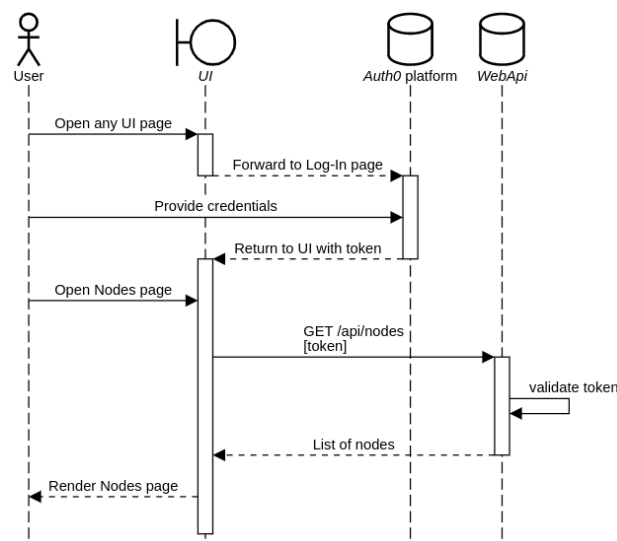


Figure 11. Example of interactive flow for authentication

Access token itself is base64 encoded JSON document with payload similar to following:

```
{
  "iss": "https://sams-project.eu.auth0.com/",
  "sub": "google-oauth2|10950374465050712345",
  "aud": [
```

```

    "sams-dwh-web-api",
    "https://sams-project.eu.auth0.com/userinfo"
  ],
  "iat": 1566382753,
  "exp": 1566389953,
  "azp": "3Zv55BlAANYq7fLkNh1FGWMS5RA4sVEF",
  "scope": "openid profile email admin",
  "permissions": [
    "admin"
  ]
}

```

Token issuing (iat) and expiration (exp) timestamps are used for token validation, while subject (sub) and scope represents user name and his permissions.

From user privacy perspective point of view SAMS DW does not store or process any user sensitive and private information – only his username is used as a reference to distinguish between private workspaces of each user.

## 7. DW Core

What it is? Why needed?

DW *Core* is considered as the main data storage and processing unit. It is important to note, that direct access from external Internet to *Core* is not possible, but provided via an intermediate WebApi module that handles user authentication and authorization, applies defined configuration and converts incoming data to a predefined format. The *Core* itself consists of several services that ensure incoming (data-in) and outgoing (data-out) data flows and provide infrastructure for internal data processing functionalities.

Core is extracted into separate module from *WebApi* in order to create abstract data processing platform suitable for building complex data processing pipelines (data-in  $\Rightarrow$  transformation A  $\Rightarrow$  transformation B  $\Rightarrow$  data-out) without focusing peculiarities of particular domain (e.g. beekeepers hive organization).

Internally *Core* services are loosely coupled with each other via messaging service. First of all, this design enables asynchronous data processing. And secondly, it has potential for sharing *Core* services over multiple hosts.

The idea behind messaging between *Core* services is to ensure data flow through them as fast as possible when data is ready for processing without specified schedule. When any component is ready to provide new data, it notifies other components about it, and if there is any consumer for new data, it immediately can start processing it.

In case of increased data processing demand (e.g. large batch of measurements was posted to the *Core*) messaging service queues processing commands and executes them as processing power is available. Also messaging service implements one-to-many (also called topic-subscriber) notification model useful for pipelines when there are multiple consumers for a single type of data. Infrastructure for messaging service is provided by ActiveMQ broker software.

## 7.1 Swamp

In the event of new data is posted to the *Core*, it is inserted into a temporary data storage, called *Swamp*. The main purpose of the *Swamp* is to handle data-in request as fast as possible without losing received data.

*Swamp* stores data-in package and few metadata fields used for debugging and monitoring purposes in dedicated collection in database:

```
{
  "_id": ObjectId("5ca89be2da6ad500097bd478"),
  "objectId": "5ca4115c5d7156000834c319",
  "type": "weight",
  "values": [{
    "ts": ISODate("2019-04-05T21:52:42Z"),
    "value": 45.23
  }],
  "createdTs": ISODate("2019-04-06T12:30:26.283Z"),
  "status": "Initial"
}
```

Upon saving the package data-in response is sent out and message that new data is available is handled to messaging service. If there are no consumers for given type of data, then saved package remains in *Swamp* until data consumer service is available. This design is deliberately selected for cases when no implementation is available yet for given data type, so *DW* does not lose received data even if particular data type is not recognized. Another potential scenario covered by this design is temporary downtime of *Core* services in distributed environment – data will stay in *Swamp* until needed service is online.

## 7.2 Vaults

For internal organization of data processing *DW Core* uses concept of *Vault*. *Vaults* can be considered as processing units with persistent storage which perform particular transformations on incoming data. After processing data *Vaults* store results in dedicated data storage and notify potential data consumers via messaging service. Data consumers in turn are other *Vaults*, and in this way pipelines of *Vaults* are built in order to implement desired data processing and/or modeling.

There can be distinguished the first order *Vaults*, which are responsible for processing raw incoming data from *Swamp*. Technically such *Vaults* store raw data “as-is” with all details, however it is more implementation feature rather than limitation. For data consistency reasons only one first order *Vault* per data type is allowed (in other words only one *Vault* can take data out of *Swamp*).

The second order *Vaults* are not limited in number and are designed to consume data previously processed by the first order *Vaults*. Multiple consumers of the processed data are allowed, as well as multiple data sources are allowed for single second order *Vault*. These *Vaults* are main components implementing desired data processing pipelines. In addition to general data aggregation functionality *Vaults* can be used to implement complex models and even handle processing to external services.

Data storage for *Vaults* were designed taking into consideration data volume estimations in time perspective. First of all taking into account *DW* application domain (primary beekeeping,

potentially other agricultural fields) timestamps are stored with precision up to a minute. More fine grained timestamps are not considered within SAMS project.

Numbers of potential records are analysed for different time periods taking into account various recording frequencies (see table below):

Table 1. Summary of potential records for different time periods

		Recording frequency					
		1 / year	1 / month	1 / week	1 / day	1 / hour	1 / minute
Total records	per minute						1
	per hour					1	60
	per day				1	24	1 440
	per week			1	7	168	10 080
	per month		1	4	28	672	40 320
	per year	1	12	48	336	8 064	483 840

As it can be seen from the table, if each minutely measurement is saved as separate record, then in course of a year significant number of records is accumulated. And these numbers are are per **single** object and **single** parameter. Taking into account that saving new record is computationally expensive operation (technically, insert operation in database involves table/collection extension, index update, etc), different data models can be more effective.

Generic *Core* usage scenario is to handle frequent data-in events (store incoming data) and occasionally provide processing results (which are usually aggregated in one way or another). Thus data storage design is biased towards effective data write operations rather than data read operations.

To address aforementioned cases data storage of *Vaults* is designed to store data in pre-aggregated bundles while still ensuring access to raw values. MongoDB specific features are actively used for this implementation.

First of all, *Vaults* store several values (usually, measurements or their derivatives) in a single record. Currently *Core* uses hourly and daily data bundles. Each record contains references to object ID, timestamp of the bundle and values of particular bundle.

The unique ID of the data bundle is a string concatenated from object ID and relevant part of timestamp. For example, ID "hive-123:2019082214" corresponds to object *hive-123* and hour from 14:00 to 14:59, day 22nd of August, 2019. Such string based ID provides very effective indexing covering main use cases. Data queries are effectively handled by RegExp engine of MongoDB like follows:

- finding records for given object
  - `db.temperature_hourly.find({ _id: /^hive-123/})`
- filtering records by timestamp
  - `db.temperature_hourly.find({ _id: /^hive-123:201904/})`
  - `db.temperature_hourly.find({ _id: { $gte: "hive-123:20190410", $lte: "hive-123:20190415" }`



```
} ))
```

- ordering records.

Storing object ID and timestamp as separate fields would require additional indexing on these fields, which in turn would lead to increased load for write operations and consumption of additional disk space.

Detailed values of the data bundle are stored as embedded document, which consists of key-value pairs:

- key is reference to timestamp element (minute for hourly bundles, hour for daily bundles);
- value is scalar or array type numerical information.

New values are added to this embedded document as they are provided to Core. MongoDB effectively handles such manipulations on records and treats them as update operations rather than insert operations. In case of relational database similar solution would imply a table with ID and number of columns corresponding to each timestamp element.

In addition to storing detailed values data bundles contain pre-aggregated values, which are computed on-the-fly when new data is added to the bundle. An example of hourly data bundle for object with ID "hive-123", day 10th of April, 2019 and three detailed values for 16:04, 16:24 and 16:45 is provided below. Fields `count` and `sum` can be used for quick average value calculation for the whole data bundle.

```
{
  "_id" : "hive-123:2019041016",
  "count" : 3,
  "max" : 21.700000762939453,
  "min" : 21.600000381469727,
  "sum" : 64.90000015258789,
  "values" : {
    "04" : 21.700000762939453,
    "24" : 21.600000381469727,
    "45" : 21.600000381469727
  }
}
```

Corresponding command for storing new value is as follows:

```
db.temperature_hourly.update(
  {
    _id: "hive-123:2019082212",
    "values.56": { $exists: false }
  },
  {
    $inc: { count: 1, sum: 23.456 },
    $max: { max: 23.456 },
    $min: { min: 23.456 },
    $set: { "values.56": 23.456 }
  },
  { upsert: true }
)
```

This command issues update operation for the database with `upsert` flag, which means it will use existing bundle if it exists, otherwise it will create new bundle. In update portion of the command it performs pre-aggregation and sets detailed values in embedded document. Whole command is performed as atomic operation and is highly effective in terms of performance.

Also taking into account filtering conditions, this command will fail on attempt to rewrite already existing value. This behaviour is expected and selected by design to increase data consistency.

There are cases for the second order Vaults when data rewrite is legitimate operation, for example, daily average temperature should be recalculated when new hourly information is added. This situation might be addressed in two ways:

- wait until data about all hours is loaded and only then initiate daily calculations;
- recalculate daily data every time hourly information is changed.

Each of these approaches have drawbacks. Expecting that all hours will be loaded is not feasible as measurement hardware might fail and some hours will be missing. From other hand recalculations after every change in detailed data might be computationally expensive.

Core tries to combine these two approaches by postponing recalculations until detailed data is stable. In other words, when new data is received, all detailed operations are processes first (e.g. hourly bundles), and then information about changed bundles is messaged to consuming Vaults for processing and, potentially, recalculations.

Vaults that expect data recalculations use slightly different queries for storing the data. First of all detailed values are stored (or replaced if already exist) and updated or inserted record is queried:

```
db.temperature_daily.findAndModify({
  query: { _id: "hive-123:20190822" },
  update: { $set: { "values.14": 21.5 } },
  new: true,
  upsert: true
})
```

Then pre-aggregated fields are calculated and stored:

```
db.temperature_daily.update(
  { _id: "hive-123:20190822" },
  { $set: { count: 9, sum: 198.53, min: 20.153, max: 23.67 } }
)
```

As described before when data bundle is created or updated, Vault sends notification message to potential consumers of new data (other Vaults). Depending on peculiarities of consumer Vault it can use already pre-aggregated values, or it also can use detailed values for custom data transformation. For this purpose Core relies on flexible query language available in MongoDB. For example, to query only aggregated values following query is used (note how whole values embedded document is excluded from query result):

```
db.temperature_hourly.find({ _id: "hive-123:2019082214"}, { values: 0})
```

Currently SAMS DW Core supports all parameters provided by HIVE devices, which are:

- temperature measurements from the inside (above bee nest) and outside of the hives, detailed values are stored in hourly data bundles and aggregated into daily bundles;
- humidity measurements from outside the hives, similarly, stored in hourly and daily data bundles;
- weight readings from scales under the hive, stored in hourly data bundles only;

- audio spectrum from sound recordings inside the hive (fast Fourier transform), provided as array of 2048 values and stored and pre-aggregated as hourly bundles.

Core is functioning from January 2019, for this moment (August 2019) database occupies 1.2 GB on disks, including journaling information. Additional statistics are as follows:

- Number of documents (records): 78 301
- Average objects size: 7.4 KB
- Data size (uncompressed): 567 MB
- Index size: 1.27 MB

As it is seen from statistics, data vs index ratio is very effective. Also significant part of data volume (553 MB) is occupied by unprocessed audio data.

### 7.3 Reports

Report in the context of *DW Core* can be considered as a component capable to extract data from one or several *Vaults* and transform it to particular format. Transformation in this context means different data presentation (e.g. plotting) rather than data processing (e.g. modelling) which is done by *Vaults*.

At this stage *Core* supports several reports to overlook the stored data. It is possible to get raw temperature, humidity and weight measurements. Audio data is still out of scope because raw audio recording spectrum is useless if presented to user as-is and should be processed before in order to extract frequency features (planned for DSS related tasks).

Core has dedicated REST endpoint for requesting report data with number of parameters:

- code – report code;
- objectId – the ID of object which data is requested;
- from / to – timestamps defining requested data period (defaults recent 7 days);
- limit – maximum number of records to be returned by the report (defaults to 5000).

The report request URL looks similar to following example:

```
https://dwh-core/reports/temperature/hive-123
?from=2019-08-18T21:00:00Z
&to=2019-08-25T21:00:00Z
&limit=2000
```

*WebApi* performs user authentication and authorization and proxies these requests to Core without modification. DW User interface has basic reporting functionality. Several screenshots show list of available reports:

## Available reports

### Name

[Raw temperature measurements](#)

[Raw humidity measurements](#)

[Raw weight measurements](#)

Figure 12. List of available reports

After selecting one type of the report, user can select time period, search for a needed nodes and chose them for report creation.

SAMS DW My workspace Dashboard Nodes Devices Reports Workspaces Calculators ▾

### Temperature /raw temperature measurements/

From   Nodes

To

Figure 13. Report creation

The chart below shows example of the raw temperature measurements of one bee colony.

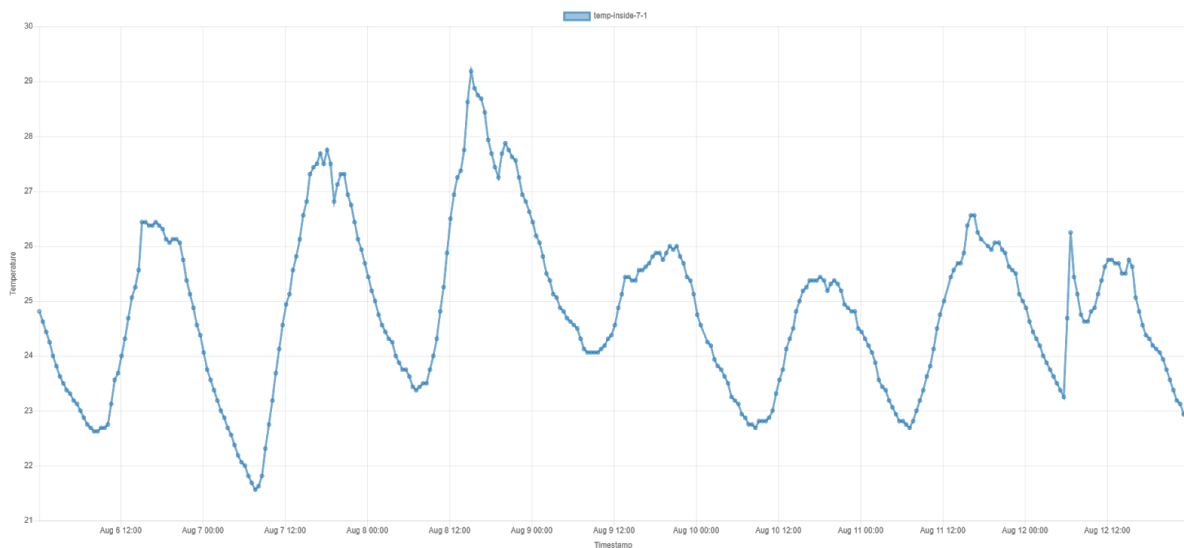


Figure 14. Example of the raw temperature measurements

Technically reports build queries for the database, transform results to column-wise representation and return results as JSON documents. Reports effectively utilize data aggregation pipeline available in MongoDB. It transforms detailed records into aggregated results taking into account report parameters. Reports use such aggregation operations as `$match` a) to filter by object ID and partial timestamp, and b) to fine-filter by full timestamp, `$project` to transform record fields, `$unwind` to work with detailed array elements, `$sample`

to limit number of records, `$sort` for record ordering. Full example of aggregation command looks like follows:

```
[{
  $match: {
    _id: { $gte: "hive-123:20190815", $lte: "hive-123:20190824" }
  }, {
    $project: {
      "_id": { $arrayElemAt: [{ $split: ["$_id", ":"] }, 1 ] },
      "arr": { $objectToArray: "$values" }
    }
  }, {
    $unwind: "$arr"
  }, {
    $project: {
      _id: {
        $dateFromString: {
          dateString: { $concat: ["$_id", "$arr.k"] },
          format: "%Y%m%d%H%M"
        }
      },
      "values.temperature": "$arr.v"
    }
  }, {
    $match: {
      _id: {
        $gte: ISODate("2019-08-15T12:45:00.000Z"),
        $lte: ISODate("2019-08-24T17:23:00.000Z")
      }
    }
  }, {
    $sample: { size: 500 }
  }, {
    $sort: { _id: 1 }
  }]
}
```

Report time period is defined using parameters and helps to estimate number of expected records for given period. Based on this information most effective data source can be selected, for example, several days can be queried from hourly vault, while year-long period can be queried from aggregated daily vaults. This information helps to balance load on the system and should be based on DW usage scenarios.

## 7.4 Core internal components

Overall conceptual diagram of *Core* is as follows:

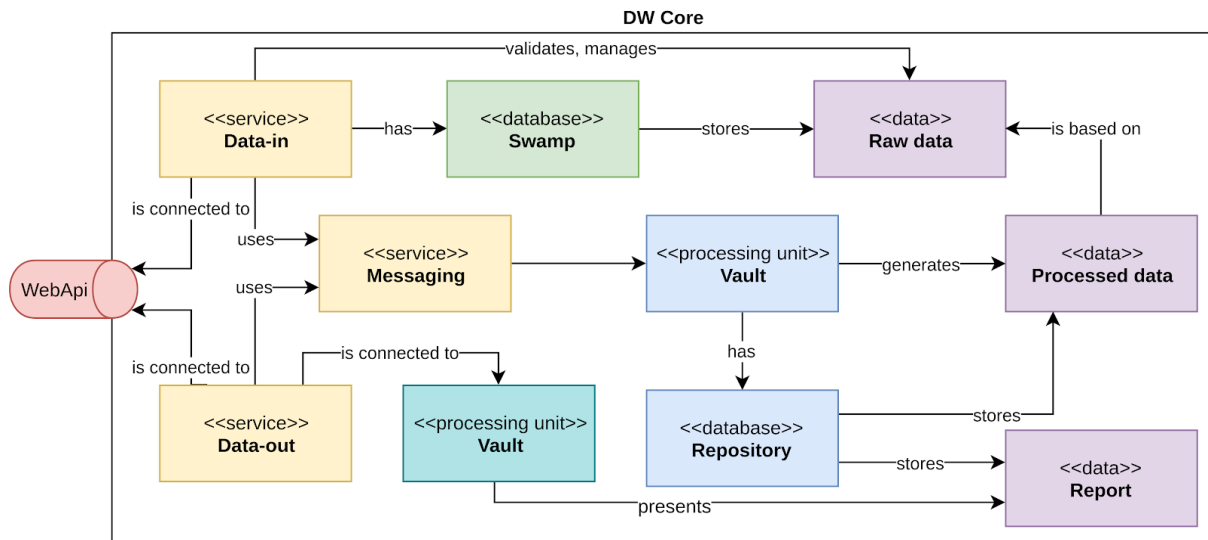


Figure 15. Conceptual diagram of the DW Core

## 8. Deployment components and infrastructure

UNILV suggest implementing *DW* as a cloud based data storage and processing unit with capabilities to combine different data sources like existing systems and available on-apiary generated data. Cloud-hosted solution has several benefits, such as simpler development and maintenance process, availability to wider audience, and others. From other hands unexpected cases may arise, such as Internet availability and network latency on client side.

For prototyping purposes *SAMS DW* is hosted on UNILV server with is physically located in university campus, Jelgava city, Latvia. The infrastructure has broadband connection to the Internet and is available from anywhere in the world (with exception if local restrictions applied to user connections). Production-grade solution should be hosted on high-end cloud platform, such as Amazon, Google, Microsoft or similar, with deliberate selection of geographical location of data centers.

In order to facilitate faster development process, to support continuous integration and to simplify deployment processes whole *DW* solution is divided into several deployment components. Main *DW* components *WebApi*, *Core* and *database* are deployed as containers on Docker platform. These containers are enclosed into local network and only *WebApi* is accessible from external networks. Requests from public Internet are handled by Nginx server, which hosts static UI and routes requests to *WebApi* container. Overall deployment structure is as follows:

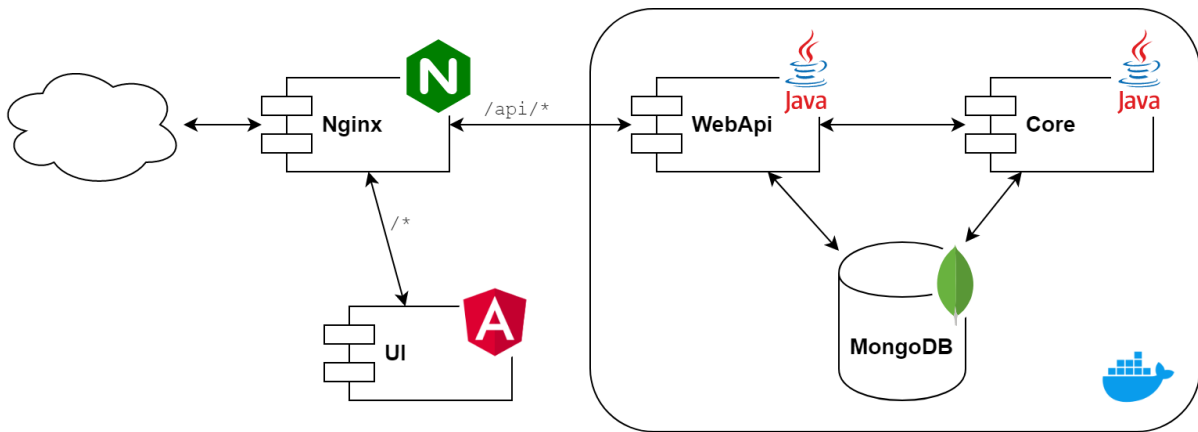


Figure 16. Deployment structure of the DW

Containers are controlled via Docker Compose utility. Configuration file (docker-compose.yml) is as follows:

```
version: '2'
volumes:
  data:
services:
  mongo:
    container_name: bees-dwh-mongo
    image: "mongo:4"
    restart: always
    volumes:
      - data:/data/db
  web-api:
    container_name: bees-dwh-web-api
    image: "registry/bees-dwh-web-api"
    restart: always
    ports:
      - "127.0.0.1:8088:8080"
    depends_on:
      - "mongo"
      - "dwh-core"
    environment:
      - JAVA_OPTIONS=-Xmx300m
      - AUTH0_CLIENT=
      - AUTH0_SECRET=
  dwh-core:
    container_name: bees-dwh-core
    image: "registry/bees-dwh-core"
    restart: always
    depends_on:
      - "mongo"
    environment:
      - JAVA_OPTIONS=-Xmx300m
```

Configuration file for building *WebApi* and *Core* images (Dockerfile) is similar to this:

```
FROM openjdk:8-jre
ENV TZ=Europe/Riga
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
ARG VER=*
ADD ./build/libs/bees-dwh-web-api-${VER}.jar /opt/app/bees-dwh-web-api.jar
WORKDIR /opt/app/
ENTRYPOINT java $JAVA_OPTIONS -Dspring.profiles.active=prod -jar bees-dwh-web-api.jar
EXPOSE 8080
```

Nginx server configuration is as follows:

```
server {
    listen 443 ssl;
    listen [::]:443 ssl;

    server_name sams.science.itf.llu.lv;
```

```

client_max_body_size 50m;

ssl_certificate /etc/.../fullchain.pem;
ssl_certificate_key /etc/.../privkey.pem;

location / {
    root /var/www/sams;
    index index.html;
    try_files $uri $uri/ $uri.html /index.html =404;
}

location /api {
    rewrite ^/api/(.*) /$1 break;
    proxy_pass http://127.0.0.1:8088;
}
}

```

## 9. Further development

In the further development of SAMS DW solution following changes shall be made:

- implement audio data processing (detecting representative frequencies for bee colony states) and reporting to user;
- develop decision support module according to user needs (advanced models);
- introduce capped collections to most voluminous raw measurements and gradually move historical data to aggregated representations;
- implement push notification from *Core* to *WebApi* and further to *UI*;
- improve UI experience for mobile platforms.

**Project website:** [www.sams-project.eu](http://www.sams-project.eu)

**Project Coordinator contact:**

Angela Zur  
 Deutsche Gesellschaft für Internationale Zusammenarbeit (GIZ) GmbH  
 An der Alster 62,  
 20999 Hamburg, Germany  
[Angela.Zur@giz.de](mailto:Angela.Zur@giz.de)



### DISCLAIMER

Neither GIZ nor any other consortium member nor the authors will accept any liability at any time for any kind of damage or loss that might occur to anybody from referring to this document. In addition, neither the European Commission nor the Agencies (or any person acting on their behalf) can be held responsible for the use made of the information provided in this document.