Ref. Ares(2023)1659695 - 07/03/2023

H2020 – LC – SC3 – EE – 2019 – GA 894240

Operating System for Smart Services in Buildings



D2.5 Upgraded Arrowhead Platform

WP2 IoT for Smart Buildings

	Name	Date
Prepared by	Nicola Cibin (AAU), Junior Dongo (AAU), Søren Enevoldsen (AAU), Brian Nielsen (AAU), Thomas Pedersen (NEOGRID)	15.01.2023
Peer reviewed by	Frédéric Revaz (HES-SO)	31.01.2023
Reviewed and approved by	Dominique Gabioud (HES-SO)	01.03.2023



This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 894240.

Distribution list

External		Internal	
European Commission	x	Consortium partners	x

Change log

Version	Date	Remark / changes
1.1	15.01.2023	First version by AAU & NEOGRID
2.1	22.02.2023	Second version by AAU & NEOGRID
3.1	01.03.2023	Final version by HES-SO

To be cited as

"D2.5 Upgraded Arrowhead Platform" of the HORIZON 2020 project domOS, EC Grant Agreement No 894240.

Disclaimer

This document's contents are not intended to replace the consultation of any applicable legal sources or the necessary advice of a legal expert, where appropriate. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user, therefore, uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Table of Contents

Executive Summary **6**

1. Introduction **7**
1.1. Aim 7
1.2. The Process 7
1.3. Definitions 8
1.4. Summary of the domOS Arrowhead Implementation 10
1.4.1. Overview of the domOS Arrowhead Platform 10
1.4.2. Description of the domOS Arrowhead Implementation 11
1.4.3. Description of domOS Arrowhead Security and Privacy Mechanisms 12
1.4.4. Description of domOS Arrowhead Platform Management 13

2. Background Technologies **13**
2.1. The domOS Architecture 13
2.1.1. Platform Concept 14
2.1.2. A Layered Architecture Model 14
2.2. Introduction to Arrowhead 16
2.2.1. Arrowhead Local Clouds 16
2.2.2. Core Components 17
2.2.3. Inter-Cloud Communication 19
2.2.4. Arrowhead Security 19
2.3. Semantic Web Technologies 20

3. The domOS Arrowhead Platform **21**
3.1. Tailoring of the domOS Platform 21
3.2. Key Features 21
3.3. Overall Architecture and Design 22
3.4. User Interface 24
3.5. Deployment Scenarios 24
3.5.1. Single Cloud, Simple Applications 24
3.5.2. Single Cloud, Complex Application 25
3.5.3. Local Clouds, Complex Applications 26

4. DomOS Components Design **27**
4.1. TDD 27
4.1.1. Functionalities 27
4.1.2. Data Structure 28
4.1.3. Public and Private APIs 28
4.2. BDD 29
4.3. Intermediator 29
4.3.1. Intermediation Functions 29
4.3.2. Intermediation Strategy 30
4.3.3. Performance and Scaling 31

4.4. Management 32
4.5. Administration GUI 32
4.6. Permissions 34
5. Identity and Access Management 36
5.1. Keycloack General Description 36
5.2. Keycloak Entities 36
5.3. Authentication and Authorization Flow 39
6. DomOS Application Specification 42
6.1. Arrowhead Application Specification 42
6.2. External Application Specification 42
7. End-to-end Interaction Example 43
7.1. Interaction Between Internal Application and Platform 43
7.2. Interaction Between External Application and Platform 44
7.3. End-to-end Interaction: From Application via Platform to Thing 45
8. NEOGRID Platform Things Provisioning 47
8.1. Generation of Things Descriptions 47
8.1.1. Conversion to domOS Ontology 49
8.2. Neogrid WoT Interface 50
8.3. Integrating Into domOS-Arrowhead 50
9. Conclusion 50
10. References 51
10.1. WoT Standards 51
10.2. Bibliography 51

List of figures

Figure 1: domOS Eco-System Development Phases 8
Figure 2: Overview of Implemented domOS Arrowhead Components 11
Figure 3: Layered Model of the domOS IoT Eco-System 14
Figure 4: Example of Arrowhead Local Cloud 17
Figure 5: Overview of the Software Components Deployed in the domOS Arrowhead Platform 23
Figure 6: Simple domOS Platform with Simple Internal and External Applications 25
Figure 7: Complex Application Running in domOS Cloud 26
Figure 8: Deployment with Local Application Clouds 27
Figure 9: A Consumer and a Thing Connected via an Intermediary. [WoT Arch, Figure 31] 29
Figure 10: Screenshot of the Platform Status Page Accessible Through the Graphical User Interface 33
Figure 11: Screenshot of the Plot Available in the Component Status Page 34
Figure 12: Example of a Permissions Document 35

Figure 13: Example of Thing Credentials Document. 35
Figure 14: Steps Required to Authenticate a User in the Platform Through an External Application Using the PKCE-Enhanced Authorization Code Flow 39
Figure 15: Steps Required to Authenticate an External Application in the Platform 40
Figure 16: Example of a JWT Released to the Intermediator Component 41
Figure 17: Internal Application Interaction 44
Figure 18: External App Interaction 45
Figure 19: End-To-End Interaction Flow 46
Figure 20: domOS Single Family Home in NEOGRID's Configuration Tool 48
Figure 21: Properties on the Central Heat Control (left) and Kitchen Thermostat (right) 48
Figure 22: Kitchen Indoor Climate Sensor in domOS Ontology 49

List of tables

Table 1: Definitions 8
Table 2: domOS Arrowhead Platform Features 22
Table 3: Keycloak Entities Configuration for the domOS Ecosystem 38

Terms, definitions, and abbreviated terms

DMP	Data Management Plan
DPIA	Data Protection Impact Assessment
DPO	Data Protection Officer
GDPR	General Data Protection Regulation
PC	Project Coordinator
TRL	Technology Readiness Level
RDF	Resource Description Framework
WoT	Web of Things
DOPL	domOS Platform
OAUTH	Open Authorization Protocol

Executive Summary

This report is the documentation that follows the “software” deliverable domOS “Deliverable 2.5 Upgraded domOS-Arrowhead platform”. It is one in a series of deliverables where three existing IoT platforms (S-IOT, cloud.iO, Arrowhead) are extended and upgraded to comply with the domOS IoT ecosystem functional specification. The purpose of the present report is to document the detailed technical developments that have been performed to upgrade the ***domOS Arrowhead platform.***

The present document reports in detail about the major design choices that was made when upgrading the platform. As an open platform it supports forced intermediation and fine-grained access control to the WoT devices that are registered in the platform. It supports things- and building-description directories. As an extension of the Arrowhead platform, it can be deployed in several different ways, ranging from a single automation cloud to multiple clouds. We have also connected the platform to NEOGRID’s backend to enable existing buildings to be integrated.

We also document implementation techniques that could be of broader interest, including the URL rewriting that is needed for the intermediation, and our security mechanisms.

In conclusion, the platform is operational, and the core features are implemented. Device monitoring and lifecycle management are to be added, and several management user interfaces need to be developed.

1. Introduction

1.1. Aim

One important objective of domOS is to elaborate and prototype a standard-based architecture named the “domOS IoT ecosystem”, that allows a decoupling between smart systems and applications, enable owners to manage their privacy, and facilitate a secure operation of buildings. It should enable future smart energy applications to seamlessly connect to sensing and actuating devices in new or retrofitted buildings.

As a key decision to enhance interoperability in the domOS eco-system, the smart systems and devices should adhere to the Web-of-Things standards specifications [WoT Arch]. Furthermore, these things are semantically enhanced and related in a building description using semantic web-technologies to further support the ease of porting applications and smart services. The binding between smart services and things are handled by a domOS IoT platform that controls and facilitates the communication, access-control policies, and privacy management. The specification of the overall functionality and architecture of the domOS IoT eco-system is described in deliverable [domOS D2.3].

Three existing IoT platforms (S-IOT, cloud.iO, Arrowhead) participate in the domOS project. These are to be extended and upgraded to comply with the domOS IoT ecosystem functional specification. The purpose of the present report is to document the detailed technical developments that have been performed to upgrade the domOS Arrowhead platform.

As NEOGRID has their own legacy proprietary platform that delivers smart energy control services to buildings. For demonstration purposes we have therefore created a link between NEOGRID’s platform and domOS Arrowhead, by exposing their buildings and devices as web-of-things objects in an automated way. In principle, this would allow further investigations of how NEOGRID applications potentially could be ported or re-implemented as external domOS arrowhead applications.

The Arrowhead platform is an existing framework and software for industrial IoT automation based on a service-oriented architecture and core-services. Application service-providers and consumers execute along core services in a local automation cloud. It thus provides an industrially accepted foundation to base developments on thereby increasing the possibilities for wider adoption. The present development enhances Arrowhead with functions for automation of energy services in buildings.

Further, as mature Arrowhead software implementations exist, it provides a stable execution vehicle for complex distributed software like an IoT platform that may be conveniently developed as a set of collaborating independent services providing flexible deployment options and potentially easing scalability.

1.2. The Process

The development of the eco-system progress according to the overall plan illustrated in Figure 1**Error! Reference source not found.**. Prior to the functional specification, the technology foundation has been

reviewed, a set of high-level requirements formulated, the first ontology version proposed (see [domOS D3.2]), and proof-of-concept of WoT intermediaries and things directory prototypes have been implemented to serve as additional input.

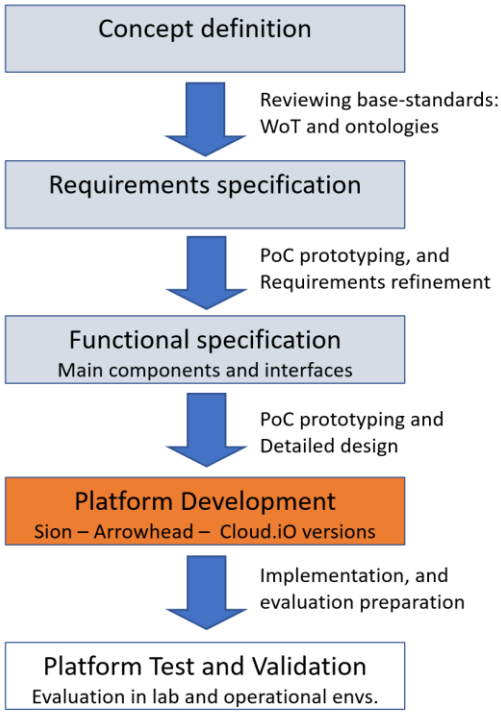


FIGURE 1: DOMOS ECO-SYSTEM DEVELOPMENT PHASES

Based on the functional specification, the next implementation step is continued here by documenting the detailed technical design and implementation of domOS Arrowhead platform. As a next step it will be demonstrated by the use-cases on flexible energy services.

1.3. Definitions

TABLE 1: DEFINITIONS

Term	*Description*	*Example*
Connected system	Digital system in a **Container** providing a data interface for an external information system	A Wi-Fi heat pump, a SunSpec compatible solar inverter, a Zaptec charging station for electrical vehicle, a Z-Wave controller with its wireless peripherals
Smart system	**Connected system** made compliant with the **domOS IoT ecosystem**	A connected system together with a cloud hosted domOS adapter component.
domOS WoT thing	**domOS compliant WoT thing**	Thermostat as defined in the domOS ontology (dco:thermostat)

Customer	Natural (or legal) person in charge of the management of a **Container** and using a **Service**.	An adult inhabitant in a household, a facility operation in a service building or in a multi-family residential building
Container	In this context, premises where the collection of **Smart systems** managed by a **Customer** are located: an apartment in a multi-family residential building, a single-family house, the communal area in a building…	The single-family house where the four above mentioned smart systems are located.
Application	Software component interacting with one or more **Smart systems** located in a **Building.**	A component aiming at maximising the self-consumption of solar power by the heat pump and the electrical vehicle charging station
Internal Arrowhead Application	Software application that offers a smart energy service through the domOS Arrowhead platform that is provisioned as an *Arrowhead compliant service consumer*.	
External Application	Software application that offers a smart energy service through the domOS Arrowhead platform, but that is provisioned outside the domOS *Arrowhead platform local cloud.*	
Service provider	Service Provider delivers a business service through an application	NEOGRID
Business-Service	Interplay of an **Application** and of one or several **Smart systems**, generating an added value for a **Customer**.	Self-consumption optimisation
SoA-Service	A software component that can function independently and offer a certain functionality via an API to other software client components. A component may be a **service-provider** or **service consumer**.	Energy consumption forecast service. Energy price forecast service Public building information registry (e.g., Danish “BBR”)
Platform	Software environment acting as a mediation entity between on one side **Containers** and their **Smart systems** and on the side **Applications.**	A cloud hosted utility enabling self-consumption optimisation application to interact with the building hosting a solar inverter, a heat pump and an electrical vehicle charging station.
domOS IoT ecosystem	ICT architecture specifying the interaction of **Applications**, **Platforms** and **Smart systems**.	
SoA	Service Oriented Architecture. An IT-Architecture where an application system is delivered by exposing, composing, coordinating multiple smaller functional service components.	Web-services

WoT	IoT Architecture defined by WoT Consortium **WoT Thing,** **WoT Consumer,** **WoT Intermediary,** **WoT Things Description (TD),** **WoT Things Directory (TDD)**	

RDF	Resource Description Framework. A W3C standard for describing the meta-data needed for data exchange from heterogeneous sources	Example in Turtle syntax:
		`ex:Heatpump01`
		`  rdf:type dco:Heatpump ;`
		`  dco:wattUsage 5430 ;`
		`  dco:locatedIn ex:Room2 .`

SPARQL	SPARQL is an RDF query language—that is, a semantic query language for databases—able to retrieve and manipulate data stored in Resource Description Framework (RDF) format.	SPARQL select query example:
		`SELECT ?device ?type ?power`
		`WHERE {`
		`  ?device rdf:type ?type .`
		`  ?device dco:locatedIn ex:Room2 .`
		`  ?device dco:wattUsage ?power .`
		`}`

JSON-LD	JavaScript Object Notation for Linked Data. A human and machine-readable data format for data that links to other data.	Building description fragment in JSON-LD:
		`{`
		`  "@id": "ex:Room2",`
		`  "dco:hasDevice": ["ex:Heatpump01"],`
		`  "dco:hasTotalSpace": {`
		`  "@type": "dco:SquareMeter",`
		`  "@value:": 42`
		`  }`
		`}`

1.4. Summary of the domOS Arrowhead Implementation

1.4.1. Overview of the domOS Arrowhead Platform

The domOS Arrowhead platform uses the Arrowhead SOA industrial automation framework as foundation to implement the domOS ECO System specification as foundation. This allows platform components as well as applications to take advantage of the Arrowhead automation clouds and its core services. Arrowhead is introduced in Section 2.2.

The domOS Arrowhead platform targets an ***open platform*** (as described in [domOS D2.3]) and thus address the particular security concerns that arise from this. This is elaborated in Section 3.1. It implements the following key features:

- AIM using OATH2.1 implemented by Keycloak, a standard and well-known access and identity manager.
- domOS dCO compliant things- and building descriptions.
- Semantic web-based knowledge bases, validation, and semantic queries. Building descriptions are queried using semantic web SPARQL queries. Support for import and export via JSON-LD is planned. These technologies are introduced in Section 2.3.
- Forced intermediation that handles all communications between external applications and things performing access control upon each affordance access.

- Fine grained things access control on affordance level; each access to a thing's affordance is checked against specified rwx affordance permissions as specified by the building owner.
- Uses an intermediation strategy that enforce a distinction between registered things and their credentials from things descriptions that are revealed to applications. Hence the internal things descriptions are re-written before they are sent to applications.
- Internal and external applications. An application may be written as a normal Arrowhead application and execute in the local cloud, or it may be completely external.
- Efficient implementation of the intermediator using Reactive Java. The platform has the ability to add multiple intermediators when required for scaling.
- It is developed as an Arrowhead compliant SOA application, thus building on a well-established software infrastructure. It supports a variety of deployment options. Some are discussed in Section 3.5.

1.4.2. Description of the domOS Arrowhead Implementation

The domOS Arrowhead platform adds to the Arrowhead components to serve as things description- and building description directories, forced intermediation with access control, platform monitoring, and user and management interfaces, see Figure 2. The design and components are elaborated in Section 4.

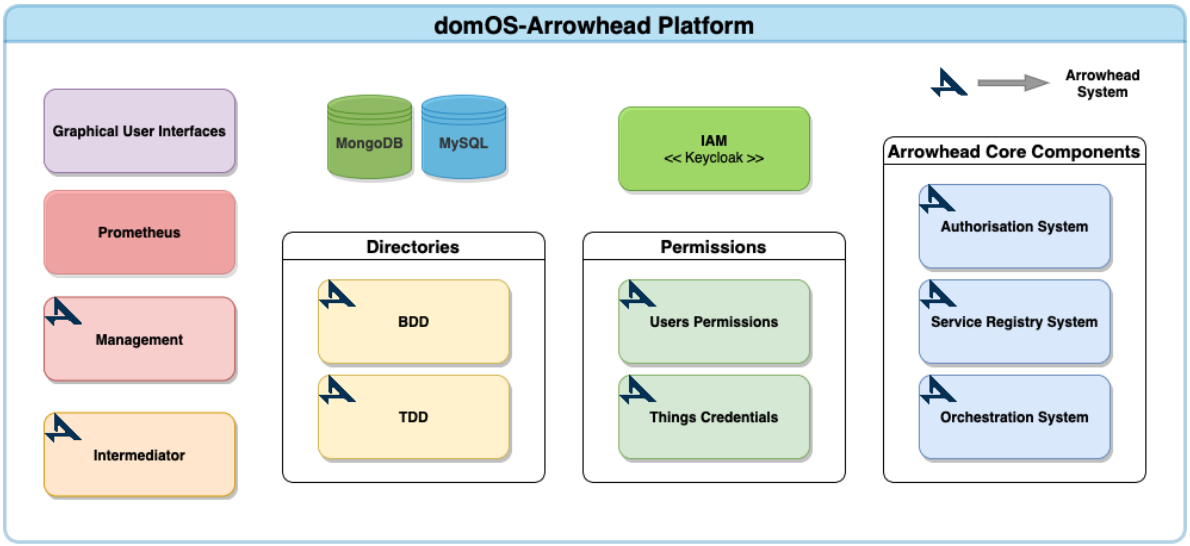


FIGURE 2: OVERVIEW OF IMPLEMENTED DOMOS ARROWHEAD COMPONENTS

1.4.2.1. Reachability of the Building Descriptions

The platform offers a REST API for semantic querying of the building description using sparql queries, see Section 4.2. These are handled by the BDD component. Import/export of building descriptions in JSON-LD is planned, but not yet implemented.

1.4.2.2. Reachability of the Things Descriptions

Things descriptions are offered through a WoT standard compliant things description directory via a REST API, as described in Section 4.1. We remark 1) there are two interfaces, one for the platform components that outputs the actual things descriptions and one for applications that return a modified description where the access URL and credentials point to the platform to enable the forced intermediation. Also 2) the BDD is the normal gateway to query the building.

1.4.2.3. Generation of the TDs and BDs

At present no user-interfaces exists to edit or import things- or building descriptions, so they are loaded manually. For the Aalborg demonstrator, Neogrid has developed automated generation of things descriptions from their platform, as elaborated in Section 8. Export of building descriptions is ongoing.

1.4.3. Description of domOS Arrowhead Security and Privacy Mechanisms

1.4.3.1. Type of Platform

The domOS Arrowhead platform targets an ***open platform*** and thus address the particular security concerns that arise from this. This is elaborated in Section 3.1. It supports two kinds of applications. Internal applications run within the Arrowhead automation cloud, and it authenticated via the core services, see Section 2.2.4. External applications run outside the Arrowhead cloud (e.g., in the NEOGRID platform) these are authenticated against a standard OAUTH service. These components and the authentication flows are elaborated in Section 5. The platform enforces intermediation, see Section 1.4.3.4.

1.4.3.2. User Authentication

The platform supports 3 user roles: platform operator, building owner, and service provider. At present there are no user-interfaces to manage these, so profiles and credentials are created manually by the platform operator. Access and Identity Management is executed using OAUTH 2.1 as implemented by the Keycloack software. This is elaborated in Section 5.

1.4.3.3. Application Authentication

Internal applications run within the Arrowhead automation cloud and are authenticated by the core services. External applications run outside the Arrowhead cloud (e.g., in the NEOGRID platform) these are authenticated against a standard Access and Identity Manager (OAUTH 2.1 service). These components and the authentication flows are elaborated in Section 5.

1.4.3.4. Security

All interaction between applications and WoT things are intermediated by the platform. The TDD never exposes WoT credentials to applications. It rewrites the things URL to the platform's intermediation component, rather than the thing. When the Intermediation component thus receives an access to an affordance, it first authorizes access against the permissions defined by the building owner. The permissions manager supports fine-grained access control with read- write- and execute-permissions at

affordance level per application per building. If access is granted the request is forwarded to the targeted thing, else an error code is returned to the application. The intermediation strategy is elaborated in Section 4.3. Permissions are described in Section 4.6.

1.4.3.5. Privacy

At present, the privacy is controlled using the low-level access control permissions. We wish to offer more high-level mechanisms to describe privacy rules, but these are to be developed. Also, we intend to filter query results from the building description based on the access control permissions, to let an application only see the relevant explicitly granted parts. But this remains to be explored and developed.

1.4.4. Description of domOS Arrowhead Platform Management

As we have prioritised key functionality like Intermediation, AIM, permissions, BDD and TDD, semantic querying, few management interfaces have been implemented. Health and performance monitoring of the platform components are handled by the 3rd party Prometheus component. We have implemented a dash-board web-application to visualize the platform status towards the operator. These are presented in Sections 4.4 and 4.5.

1.4.4.1. Device Management

Currently, there is no thing health monitoring and no interfaces for thing lifecycle management (commission, replace, decommission).

1.4.4.2. Application Management and Enrolling

Currently new applications need to be onboarded manually in the AIM. It remains future work to develop an “app-store” for the platform where building owners can purchase, deploy, and manage applications that hosts smart energy services, and enable management thereof. Many applications will need a subscription and a contract negotiated outside the scope of the platform.

2. Background Technologies

2.1. The domOS Architecture

The domOS report [domOS D2.3] gives a detailed specification of the domOS eco-system functionality and architecture. It provides:

- A reference-model that shows the conceptual layering of the eco-system.
- Definition of the major components and roles.
- A functional specification of the major components.
- An overall design of the domOS architecture that can later be designed in detail and turned into a proof-of-concept implementation and demonstration.
- Concise specification of the major interfaces and APIs.

2.1.1. Platform Concept

An ***IoT platform*** is the required hardware and system software components needed to facilitate the delivery a business service to a set of costumers. The platform is thus a central part of an eco-system that bridges applications and IoT devices. By developing the platforms from common concepts and specifications, the scope of IoT devices and applications are greatly improved and makes porting between platform instances easier.

A ***platform instance*** is a specific hardware and software implementation of the common concepts developed and delivered by a given vendor or a given platform operator.
The domOS project will prototype three different platform instances (SION, Cloud.IO, Arrowhead) having different application areas, deployment scale, and demonstrator goals. The functional specification must consequently be tailorable and allow sufficient flexibility to cater for the needs of the different platforms whilst providing sufficient similarity to facilitate industrialization of smart buildings and services.

2.1.2. A Layered Architecture Model

To make this document self-contained we here briefly summarize the main functionalities and components of the domOS IoT eco-system but refer to [domOS D2.3] for details.

Figure 3 **Error! Reference source not found.**shows a conceptual layered model of the domOS eco-system and its major functional components.

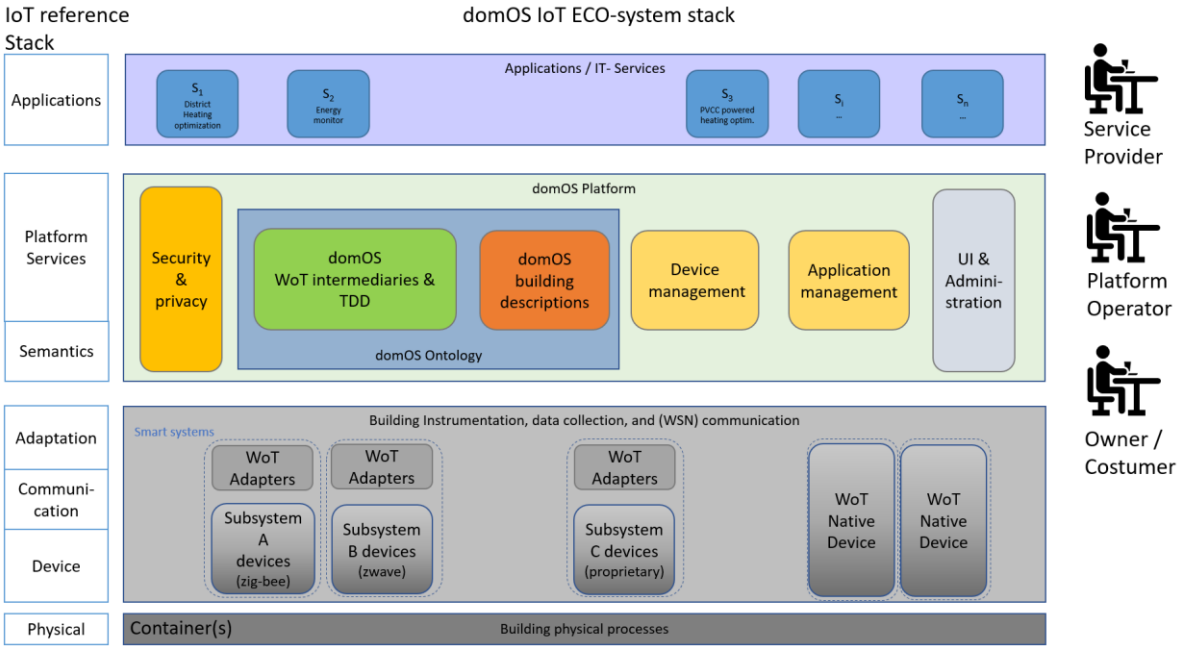


FIGURE 3: LAYERED MODEL OF THE DOMOS IOT ECO-SYSTEM

The lowest layer is the physical structure and processes (e.g., thermodynamics) in the buildings. On top of that is instrumentation by adding sensing and actuating capabilities typically hosted by wireless sensor network devices and provision of WoT access.

The next conceptual layer is the platform layer that bridges the gap between smart service applications and raw devices by providing a common, secure, and privacy aware semantic interoperability layer. The upper layer hosts the potential applications using the platform. The diagram shows how a platform instance may control several building containers and enable multiple services to be enabled for each building container.

The service provider role delivers an energy optimization service. The role is conceptually different from the platform operator role who hosts and manages the platform itself. The container owner role decides what services to run on its container, controls access to its devices and its privacy policy.

domOS use **WoT** as common device abstractions. Machine readable **Thing Descriptions (TDs)** [WoT TD] define the capabilities of the thing. These TDs are often supplied by the smart system manufacturer (or other external party) and become registered in the **Things Description Directory (TDD)** by the building owner using suitable user interfaces when deploying the smart system device.

A **Building Description** is a description that defines an abstract description of a building and its properties as needed by the smart services. It consists of a description of building specific metadata (e.g., total space, thermal capacity, envelope type), a description of the structure of the building (e.g., floors, apartments, rooms), and links to the things and services installed in the building.

The **domOS Common Ontology (dCO)** defines the semantics of the concepts and their relationships needed to enhance interoperability. In particular, the dCO is used in the building description and to semantically enhance things descriptions. The dCO will be described by semantic web schemas that can be used for validation of building and things descriptions.

Device (Things) Management concerns management of the life cycle of the Things: commissioning, decommissioning, pausing, replacing devices, and health-monitoring.

An important concern in domOS is **privacy**. Therefore, building owners must be empowered by the eco-system to control what services run, what sensor data a given service can access, and reveal to the services as little information as possible about the building and the installed devices and capabilities. Therefore, the platform acts as an **intermediary** that performs access control and filters unneeded information.

The end-user facing user interfaces are primarily web-based (“web-apps”) graphical user interfaces allowing “ordinary” people to manage their container, devices, and subscriptions. The technical interfaces are REST style web-APIs.

The architecture is defined functionally, that is, it does not specify where its components are deployed (i.e., on an existing or dedicated gateway/edge, cloud, or combination thereof), similarly no extra-functional requirements are specified.

2.2. Introduction to Arrowhead

The domOS platform concepts and requirements are being implemented and extended taking advantage of the Arrowhead Framework, which provides an architecture for building IoT based Automations Systems. In Arrowhead, IoT devices are organized into local clouds, and they are abstracted into services to enable interoperability in-between devices which have been fabricated by different manufacturers. For the implementation of these local clouds, the architecture consists of several core systems and services - which will be presented in the following sections - that support design, engineering, deployment, and operation of the developed platform. The Service Oriented Architecture (SOA) enforced in Arrowhead allows many different systems to seamlessly interact with each other without the need of any further adaptation layer; other fundamental properties exhibited by the SOA can be found in [Arrowhead]. Furthermore, given that many applications may require multiple local clouds to build a total automation system, inter-cloud communication is supported without sacrificing cloud internal security. In fact, data encryption across different clouds can still be maintained with, for instance, IPSec or TLS.

Arrowhead framework potentials and applications have been investigated in many fields spanning from Smart Cities, Smart Agriculture [MSBVD 2020], to Supply Chain management and logistic [KVH 2019]. In particular, it has proved to be a viable solution to create multi-vendor system of systems applicable in the smart city domain. In [JLL 2016], two pilots targeted to smart city and infrastructure domain have been deployed; the first one is for street lightning system and the other is for car heating system. In [Arrowhead], a demo of control loop over the level of a flotation tank is discussed.

2.2.1. Arrowhead Local Clouds

The concept of an Arrowhead local cloud is depicted in Figure 4 where several application services consumers and provider collaborate to provide a common functionality using the Arrowhead SOA core services.

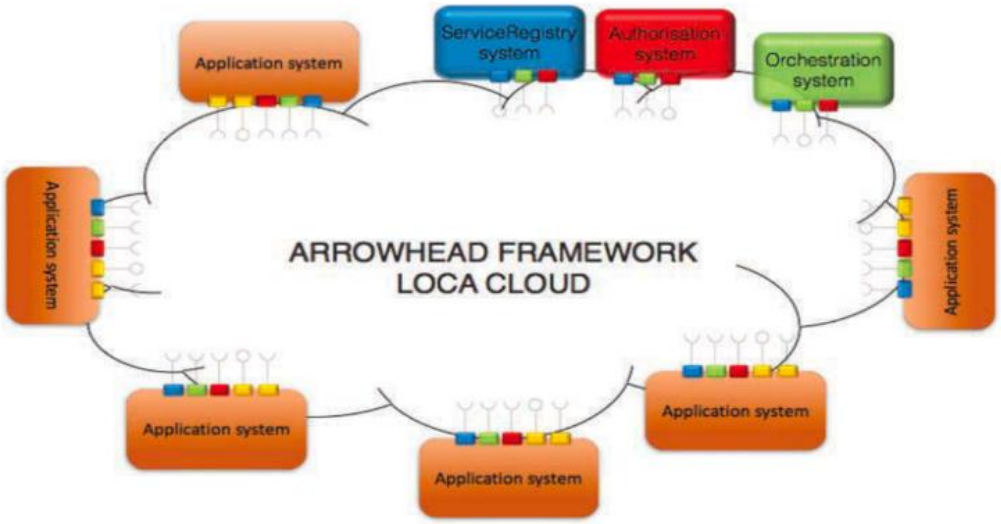


FIGURE 4: EXAMPLE OF ARROWHEAD LOCAL CLOUD

The adoption of Arrowhead Local Clouds provides to the developed platforms the following improvements and guarantees [Arrowhead wiki]:

- Real time data handling.
- Data and system security.
- Automation system engineering.
- Scalability of automation systems.

In the context of the Arrowhead Framework, a *Service* is what is used to exchange information from a *providing* system to a *consuming* system, and a *System*, which can also be called *Component*, is a piece of software executed on a device offering/consuming these services. Each service in turn provides *Service's Endpoints*, which are communication points identified by a unique URL and used to access the service functionalities. In the next section the three core mandatory components are presented and described. These are Authorization, Service Registry, and Orchestration. These three mandatory systems are needed to enable and allow services exchanges between a service provider and service consumer with the desired level of security and autonomy. Further of these mandatory components, the following Automation Support Core Systems are available to ease the deployment, operation, and maintenance of the cloud:

- Plant Description System.
- Configuration System.
- System Registry System.
- Event Handler System.
- QoS Manager System.
- Historian System.
- Gatekeeper System.
- Translation System.

Given that not all of them are used in our platform, it's out of the scope of this document to describe their characteristics, but their accurate description can be found in [Arrowhead]. In Section 2.2.3 we only briefly dwell on the Gatekeeper Systems, which can be used to provide inter-cloud services.

All the functionalities required by a DomOS compliant platform are implemented through specific Arrowhead Application Systems deployed in the local cloud. An Application System is an Arrowhead component which is at minimum designed to communicate with the mandatory components (e.g., register and un-register provided services in the Service Registry System), and to consume and provide at least one service. The DomOS related Application Systems deployed in our platform are discussed in Section 4.

2.2.2. Core Components

In the following, the three Arrowhead core components are presented. To operate an Arrowhead local cloud, it is required that at least these components are deployed and running. This is because, as it will be discussed in the following, they provide fundamental functionalities to allow and secure all the interactions that occur in-between the various Arrowhead systems deployed in the local cloud.

2.2.2.1. Authorization

The Authorization System deals with the authentication and authorization of the various other systems providing/consuming services in the local cloud. Authentication is based on the release and verification of X.509 certificates, which are generated for each component starting from the Arrowhead Local Cloud specific root certificate (which is in turn generated starting from the Master root certificate). More details about the certificates are provided in Section 2.2.4.1. For what concerns the authorization process, this core component relies on two different databases: the first one describes which Application System can consume which Services from which other Application System (Intra-Cloud access rules), and the second describes which other Local Clouds are allowed to consume what Services from this Cloud (Inter-Cloud authorization rules) [Arrowhead Github]. Moreover, the Authorization System provides a session control functionality through the core components with the usage of Json Web Tokens (JWTs), but in order to ease the interaction with applications external to the Arrowhead Local Cloud, this functionality has been disabled in our deployed platform (without affecting the overall security of the platform), and the generation and management of the tokens has been moved to an external AIM (i.e., Keycloak) which is described in Section 5.

2.2.2.2. Service Registry

The Service Registry System offers functionalities to keep track of and provide information about which services are offered by the systems deployed in the Arrowhead Local Cloud. Indeed, this system allows other application to publish, un-publish, and update services that they're offering, and to look up for services offered by other components. This component is of fundamental importance in particular for the *query* endpoint, which is meant to be used to translate a symbolic service name into a physical endpoint (e.g., IP address and port), which then will be used to consume the required service. Before allowing any Application System to register/un-register its offered services in the registry, the Service Registry first of all verifies that the Application System possess a valid X.509 certificate, then checks that the Application System belongs to this Local Cloud, and finally if it is properly bootstrapped into the Local Cloud. Only at that point the request is processed. Obviously, an Application System can only manipulate entries in the Service Registry databases which are relative to services which it is providing, and not the ones provided by other components.

2.2.2.3. Orchestration

The Orchestration System is used to store orchestration rules and resulting orchestration patterns. This information is used to redirect a service consumer to the most appropriate service provider according to QoS, security, or other specific requirements defined by the user, and to provide the requesting application with all the information needed to establish the connection. In particular, the orchestration component is of fundamental importance if in the deployment scenario one or more of the following features is needed:

- Accountability for all Systems in the Local Cloud.
- QoS and resource management reasons.
- Inter-Cloud orchestration and communications.

In these cases, the Orchestration is the sole entry point to establishing new connections within the local cloud and with the external clouds, and Applications are obliged to first interact with it.

2.2.3. Inter-Cloud Communication

A complex application can often require access to services offered in different local clouds; to support these uses cases, Arrowhead framework supports inter-cloud communications. This functionality is provided through the Gatekeeper System, and the objectives of this component are to enable the discovery, orchestration, authentication, and authorization of services residing in different local clouds. In order to offer these functionalities, this component needs to interact with the three core components deployed in the local cloud where itself is deployed, and with the Gatekeeper of the remote cloud with which the data exchange needs to be initialized and then performed. The decision to which remote Gatekeeper to connect to can be hard written on the local Gatekeeper configuration, or it can be the Gatekeepers itself who automatically detects and polls their neighbourhood for cloud discovery. The choosing of which approach to implement depends on the volatility of the local clouds and of the components deployed in them.

2.2.4. Arrowhead Security

In Arrowhead clouds, authentication and authorization of the various systems can be enforced using a combination of two different approaches, which consist of the usage of X.509 Certificates and Json Web Tokens (JWTs).

2.2.4.1. Certificates

Arrowhead Framework's security relies on the SSL Certificate Trust Chain. This chain consists of three levels: Master, Cloud, and Client Certificate. Each Arrowhead System needs to have its own X.509 certificate signed with the Local Cloud certificate's private key. In turn, each Arrowhead Local Cloud has its own certificated signed by the master certificate's private key, which makes the whole chain trustworthy. In the current domOS platform implementation, self-signed certificates are used, but in a public deployment a master certificate released by a Certification Authority should be used for trustiness reasons. Each communication inside the Local Cloud and with external applications relies on the HTTPS protocol, with both client and server certificate authentication. Thus, if one of the two certificates presented by the parties involved in the message exchange is not properly signed, the interaction is not even started, and the process is aborted. During the authentication process, the Subject Common Name stated in the presented certificate is compared with the component name which started the communication, and, if the two don't match, the request is refused, and the connection is dropped. A special certificate named "System Operator Certificate" (SysOp) can be issued starting from the Cloud Certificate if an external client needs to use the management endpoints of the Arrowhead Core Systems. This can be useful during the development process to interact with the platform for instance through browser or other testing software (e.g., Postman), or to use the Arrowhead's Management Tool in secure mode.

2.2.4.2. Tokens

To control the authorization to access the various systems services, a token system based on the generation of Json Web Tokens (JWTs) is implemented in Arrowhead. These tokens are released during the orchestration process by the Authorization component to the system which is trying to consume a service. Then, the token released to access the required service is presented by the service consumer to the service provider to be authorized and to start the data exchange.

2.3. Semantic Web Technologies

There are various web technologies which may be of use in domOS. The building description and thing descriptions are JSON-LD documents. This implies that in addition to being readable as plain JSON documents, they are also linked data. JSON-LD documents can be provided with a context that maps the document to the RDF (Resource Description Framework) data model. The RDF data model consists of triple statements which links *subjects* to *objects* with an arc describing the relationship *predicate*. An example, here written in a syntax known as Turtle: `bd:Building01 dco:hasApartment bd:Apartment4`, which encodes a triple describing that `bd:Building01` has an apartment identified by `bd:Apartment4`. This provides the terms in the documents with unambiguous identification and provides semantic meaning to the text such that it can be machine processed and the knowledge integrated with other data. For domOS, which uses the domOS Common Ontology, unambiguous meaning can be given to the elements of building descriptions. For a temperature measurement, the ontology can be used to provide unambiguous information about the unit of measurement or what the purpose of the measurement is (e.g., if it is the supply temperature or return temperature in a heating context).

The implied RDF data model can be stored in a triplestore, a database specifically tailored for this kind of structured information. Using a query language, such as SPARQL, the data can then be queried in more expressive ways than just raw RDF triples. Some queries that may be interesting for consumers of the building descriptions are:

- Retrieve the IDs of all Things that have a property of type `dco:returnTemperature` or `dco:supplyTemperature`, where dco is a prefix for the domOS Common Ontology.
- For an apartment block they may want to query for all rooms that are an apartment: `SELECT ?apartment WHERE {?space dco:hasApartment ?apartment. }`.

For the platform itself, RDF can aid the building owner in making the building description compliant. For instance, by combining the data for the tentative building description with the thing descriptions associated with the owner, the platform can query, on behalf of the user, to inform about:

- Thing descriptions that are not referenced by the building description, and thing descriptions that are referenced but not found.
- Missing, but required, information that must be in the building description.
- Semantic inconsistencies like conflicting unit of measurement, or circular references in the spatial partitioning.

3. The domOS Arrowhead Platform

3.1. Tailoring of the domOS Platform

The domOS functional IoT eco-system specification [domOS D2.3] is designed to accommodate a variety of different needs, ranging from simple gateway deployments for a single building offering a few controlled services to fully fledged open platforms with multiple service suppliers offered for multiple buildings. The specification is therefore tailored towards its intended usage of a given platform.

The domOS Arrowhead platform aims at being an **open platform** where the building owner is in control of the dynamic set of devices attached to his building, and the dynamic set of applications he subscribes to. Applications may be developed by 3rd party developers, may potentially be externally hosted, and potentially be advertised in an application marketplace and activated by the building owner on demand. In such a platform strong security, access control, and privacy management features must be implemented. The platform is deployed in a multi-tenant cloud and operated by an independent platform operator.

3.2. Key Features

The domOS Arrowhead platform implements the following key features:

- AIM using OATH2.1 implemented by Keycloak, a standard and well-known access and identity manager.
- domOS dCO compliant things- and building descriptions.
- Semantic web-based knowledge bases, validation, and semantic queries.
- Forced intermediation that handles all communications between external applications and things performing access control upon each affordance access.
- Fine grained things access control on affordance level; each access to a thing's affordance is checked against specified rwx affordance permissions as specified by the building owner.
- Uses an intermediation strategy that enforce a distinction between registered things and their credentials from things descriptions that are revealed to applications. Hence the internal things descriptions are re-written before they are sent to applications.
- Internal and external applications. An application may be written as a normal Arrowhead application and execute in the local cloud, or it may be completely external.
- Efficient implementation of the intermediator using Reactive Java. The platform has the ability to add multiple intermediators when required for scaling.
- It is developed as an Arrowhead compliant SOA application, thus building on a well-established software infrastructure.

TABLE 2: DOMOS ARROWHEAD PLATFORM FEATURES

Feature	Level (M) Mandatory (R) Recommended (O) Optional	domOS Arrowhead implementation status (Implementation level)
Devices as DCO annotated WoT Things	M	100%
Applications as WoT Consumers	M	100%
Building Descriptions using DCO	M	50%
WoT TDD	R	100%
WoT Intermediaries	R	100%
Forced WoT intermediation	R for closed platform M for open platform	100%
Privacy Rules, Information Hiding	M	80%
Transparent Security and Privacy status view	R	50%
Access-control	R M for open platform	100%
Application Authentication and Authorization	R	100%
Platform multi-tenancy	R	90%
Device management and monitoring	O	0%
Application subscription and lifecycle management	O	0%
Application compatibility check and feature interaction check	O	0%
User- and- Administrative Interfaces	O	10%

An overview of the domOS Arrowhead platform features and comparison with the functional specification is provided by Table 2.

3.3. Overall Architecture and Design

In this section, an overview of the components deployed in the platform and their design are presented.

As can be seen in Figure 5, a number of different components have been developed and deployed in order to offer all the functionalities promised by the domOS ecosystem. On the right side of the figure, we can observe the three Arrowhead core components, namely, Authorisation, Service Registry, and Orchestration systems. These components are of fundamental importance to operate the Arrowhead local cloud, and they rely on a MySQL database to store their operational information and data. Then, we

have the two components used to manage users' permissions and devices' credentials, namely, User Permissions and Things Credentials. To offer SSO functionalities, an external IAM is deployed, that is Keycloak. In order to store Building Descriptions (BDs) and Thing Descriptions (TDs), BD Directory (BDD) and TD Directory (TDD) respectively have been designed and deployed. They both rely on MongoDB to store and retrieve these documents in JSON format. As for domOS specifications, all interactions between external applications and devices have to be intermediated through the platform, for this scope the Intermediator component has been conceived. As it will be discussed in detail in Section 4.3, it is meant to properly scale to serve a huge number of concurrent requests. Given the number of different systems which needs to be monitored, Prometheus is used to retrieve insight abouts the components status and the number of processed requests. Finally, the management component is used in combination with the Graphical User Interface to offer users functionalities to interact and manage the platform as a whole (e.g., check components status), and the various provided services (e.g., modify users, devices, and buildings information).

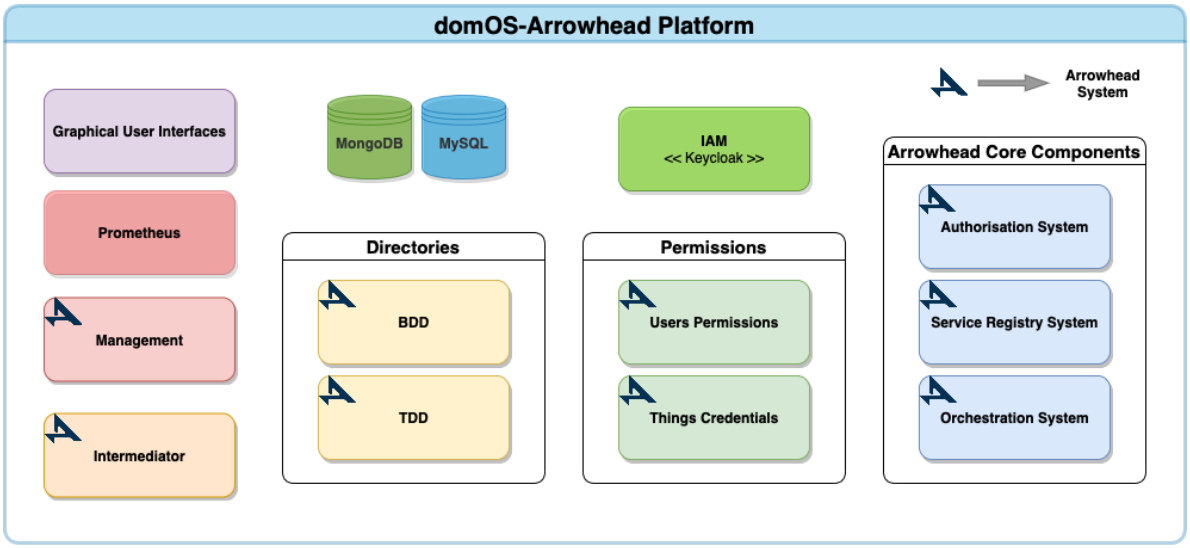


FIGURE 5: OVERVIEW OF THE SOFTWARE COMPONENTS DEPLOYED IN THE DOMOS ARROWHEAD PLATFORM.

Few more words need to be spent about Figure 5. It can be noted that most of the components are marked with the Arrowhead logo; this is to emphasize that those systems are deployed as Arrowhead applications in the arrowhead local cloud, and this means that they are distinguished by the other general-purpose components by the following characteristics:

- Orchestration and Authorization systems are used to check whether the interactions with another component are authorised and meant to be performed.
- Service Registry is used to publish provided services, to look for services provided by other components, and to resolve services name into physical endpoints.
- X.502 Certificates are installed to provide client-server authentication, and thus to add a further level of trust and security.

Implementing these components following Arrowhead directives allows as to take advantage of the security and automation features provided by the Arrowhead Framework.

3.4. User Interface

The platform provides a graphical user interface (GUI) which allows users to get access and manage the system in a user-friendly way. In order to make the GUI as much portable and easily accessible as possible, it has been developed as a web application which can be accessed from any device (e.g., pc or smartphone) on which a browser is installed, without requiring any additional set up nor configuration. Once the user has accessed the platform web interface, he only needs to provide his credentials to start enjoying the various offered functionalities, which will be discussed in detail in Section 4.5. Thanks to the SSO experience implemented in the platform, the just mentioned user's credentials are the same username and password used to get access to all the other services available in the domOS ecosystem. The same web portal is used by all the three different types of users, namely, building owners, platform operators, and service providers, and according to the type of user who has logged in, different information can be accessed, and operation can be performed.

3.5. Deployment Scenarios

The domOS Arrowhead platform enables a variety of options for deploying the platform and hosting the applications. In the following, three different deployment scenarios are delineated.

3.5.1. Single Cloud, Simple Applications

In a simple setup, the domOS platform components (management, BDD, TDD, permission checker, credentials manager, etc.) run in an Arrowhead local cloud that supports the binding, authorization, and orchestration of these. We refer to an application that is implemented as an Arrowhead service as an ***internal application***. This configuration is suitable where an application provider and platform operator coincides. An application may also be hosted by an application provider outside the platform. Such ***external applications*** may be granted access via the platform authentication mechanism for external applications. This second configuration is suitable when the application provider and platform provider are different (legal or administrative) entities.

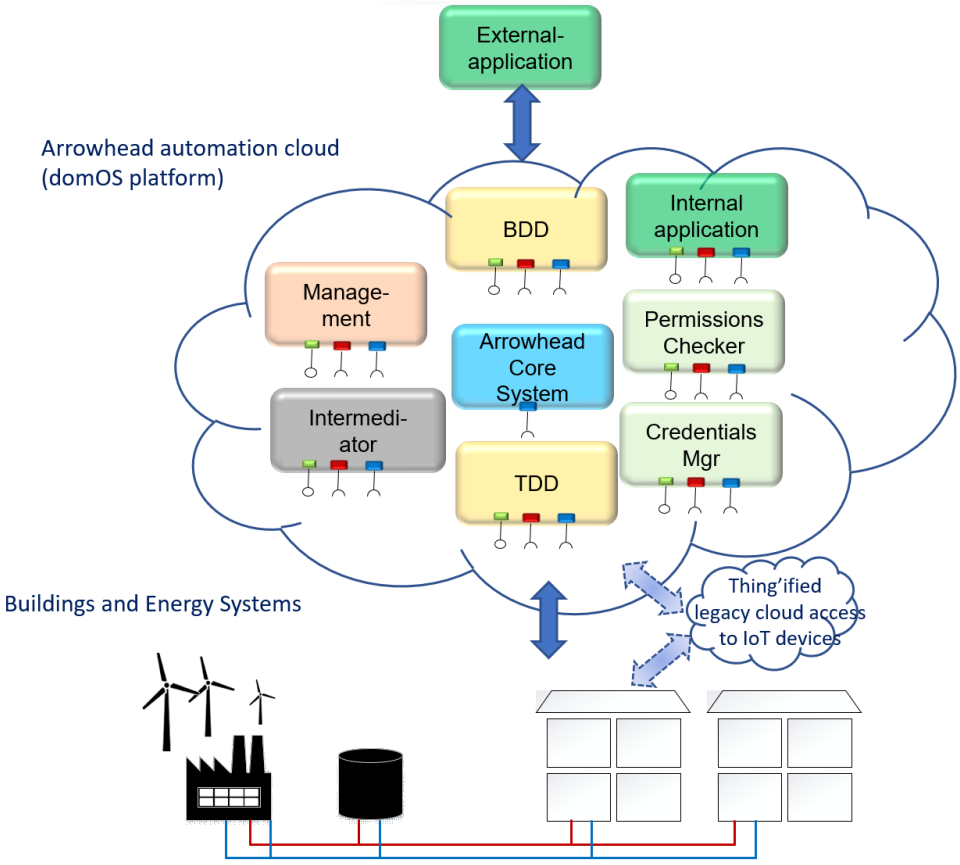


FIGURE 6: SIMPLE DOMOS PLATFORM WITH SIMPLE INTERNAL AND EXTERNAL APPLICATIONS

A final remark is that from the domOS perspective, access to external IoT devices or services takes place via the WoT Things Description abstraction. This may offer direct access to WoT native IoT devices, or access to devices where the manufacturer (only) offers a web-api interface that is (manually) abstracted as a WoT thing. This is seen frequently in the current silo-ed IoT world where each manufacturer offers his own cloud access API, or where a larger installation already exists (such as the NeoGrid/Aalborg demonstrator). Further, application components may access any external service or data-source (not illustrated) that is technically possible that does not conflict with the deployed security mechanisms.

3.5.2. Single Cloud, Complex Application

More complex application may be build using several separate, independently developed and running components. For instance, a (hypothetical) application that reduce the carbon emission in district heating, may perform optimized control using services that computes consumption- and emission-forecasts. Such an application may also benefit from the SoA mechanisms that already exists and are supported by the Arrowhead cloud.

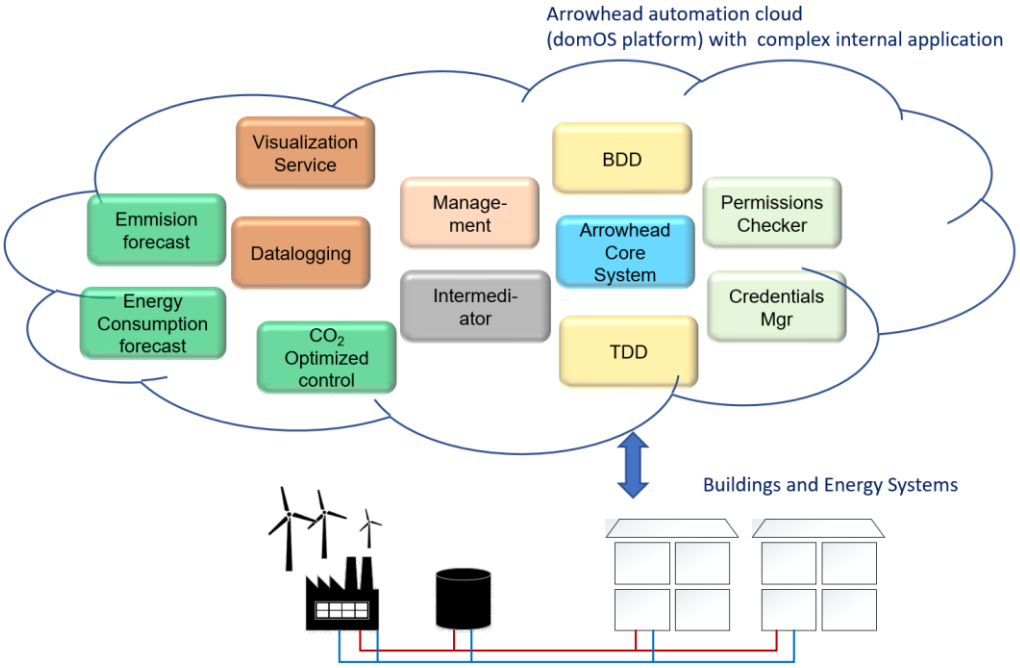


FIGURE 7: COMPLEX APPLICATION RUNNING IN DOMOS CLOUD

3.5.3. Local Clouds, Complex Applications

A final option discussed here is the use of a local automation cloud per application. This is useful where it is desired to keep the application components isolated from the platform components, e.g., where there are several complex applications, and where the application provider and platform operator are different (administrative or legal) entities. Arrowhead supports the required cloud-to-cloud intercommunication and authorization.

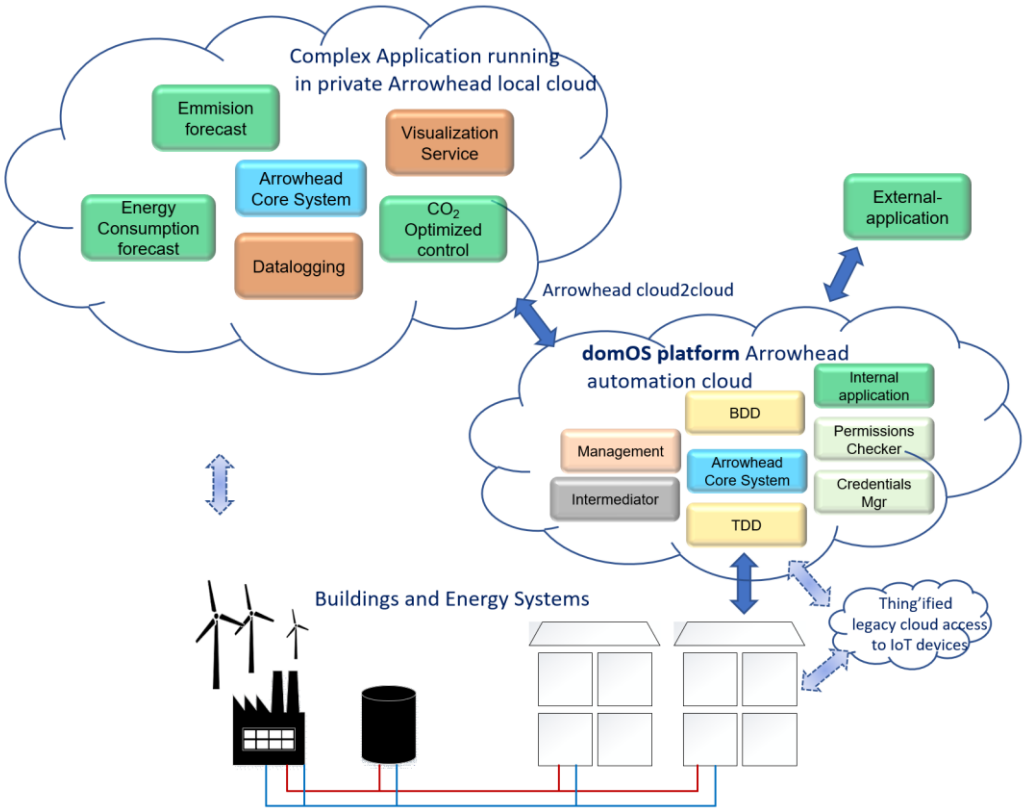


FIGURE 8: DEPLOYMENT WITH LOCAL APPLICATION CLOUDS

4. DomOS Components Design

4.1. TDD

Defined by the WoT architecture, the Thing Description directory is a directory service for Thing Descriptions. It provides a Web API to register them and make metadata of the things available to applications. A client application can then find the meta-data needed to communicate with a thing by querying the Thing Directory.

4.1.1. Functionalities

The main functionality of this component is to provide storage capability for the Things description.

We consider for our implementation a full HTTP directory, meaning that we provide both read and write over HTTP. All the CRUDL (create, read, update, delete, and list) interfaces have been implemented.

Create

This functionality refers to the registration of a new TD inside the TDD. A container Id must be provided with the TD content.

The TD is validated following the JSON Schema validation; a thing description Id is generated and returned after registration.

Read

This functionality refers to the retrieval of an existing TD from the TDD. The thing description id and the container id must be provided.

Update

This functionality is used to replace or partially modify an existing TD. The thing description id, the container id and the new TD must be provided. The TD is then validated in accordance with the JSON Schema validation before storage.

Delete

The delete functionality is used to remove a TD from the directory. The thing Description id and the container id must be provided.

List

This functionality provides a way to query the collection of full TD from the directory belonging to the same container. The container Id must be provided.

4.1.2. Data Structure

Thing descriptions are JSON-LD documents. They contain the interaction model with Properties, Actions and Events of a thing. The TD provides different metadata such as Protocol Bindings (protocols supported by a thing). They are identified by a unique id and contain elements named title and description to provide human-readable metadata, which can be used as comments for developers or to display texts in the user interface.

A Thing Description instance complies with this domOS specification if it is valid. We provide a JSON Schema [https://dco.domos-project.eu/files/domos-td.json] for this validation.

Thing Descriptions are organized by containers with a container-id as an identifier to support multi-tenancy. For their storage, the TDD relies on using a MongoDB database.

4.1.3. Public and Private APIs

Two APIs are implemented for the TDD. The internal API is used internally by the components part of the platform. It is the case for the intermediator component, which uses the internal API to get the original TD for the Thing so that direct communication can be established between the intermediary and the Thing.

As mentioned in Section 4.3, the system does not allow direct interactions between a thing and an application. The intermediary should handle all communications. The public API is then used to expose

only retrieval operations. It allows to retrieve modified version (as described in Section 7) of the TD, which points to the intermediary.

4.2. BDD

The building description directory is for building descriptions what the thing description directory is for thing descriptions. The component stores the building descriptions and permits, given proper authorization, the manipulation of a given building description. In addition to storage, retrieval, and modification of the building descriptions, it also supports building owners in designing their building description by providing syntactical and semantic feedback. This can include informing about the required field that are missing, or elements with invalid or conflicting data with respect to the domOS Common Ontology.

4.3. Intermediator

One of the goals of domOS is to leverage the WoT architecture [WoT Arch] for interoperability with IoT devices. Based on a Thing Description, a consumer knows how to interact (communicate) with the device (the WoT Thing). There are various technologies that can be used to perform the interaction depending on the device. A Thing may expose itself as capable of understanding HTTP requests. Sending an HTTP request with the verb 'GET' to a particular URL may result in the temperature reading of a thermostat. Sending a POST request may permit changing the setpoint of the thermostat. HTTP is not the only understood protocol: there are also bindings for CoAP and MQTT and other bindings may be standardized in the future.

Since these devices can be exposed to the internet it is often necessary to have the proper credentials for the request to be successful. A major security requirement of domOS is that application services do not possess the credentials for the resource owners' devices. This drives the need for domOS (which is trusted to care for the credentials) to contact the WoT devices on behalf of the application service. Simultaneously we do not want application services to use some different patterns of communication since they already know how to interact with WoT devices. This implies domOS must insert itself in between the application service and the WoT device and mediate the communication. In the WoT architecture this is known as an intermediary, shown in **Error! Reference source not found.**.



FIGURE 9: A CONSUMER AND A THING CONNECTED VIA AN INTERMEDIARY. [WOT ARCH, FIGURE 31]

4.3.1. Intermediation Functions

The intermediary will receive requests from the application services intended for WoT Things, forwards them to the device, and then forwards the response back to the application services. Since application services do not have the credentials for the devices, the intermediary will instead need to be able to retrieve the credentials to send the request. The application will use another set of credentials to prove

to the domOS platform that they have indirectly been granted the rights (e.g., subscription process) to communicate with the devices.

Application services determine how to make the interaction based on the connection information embedded in the URIs found in the thing descriptions. We rewrite the connection information (in addition to security configuration) in the original thing descriptions presented by the resource owner to guide the application services to connect to our intermediary of which there may be multiple instances, instead of directly to the device. The receiving intermediation instance will then parse the request, and check permissions, and gather the credentials to communicate with the WoT device. At start-up time the platform will choose an *intermediation strategy* which determines how the connection info is rewritten to direct application service requests to the proper intermediation instance. The real thing description given by the building owner is referred to as the *internal thing description* and the modified version given to application services is referred to as the *external thing description*. For platforms with a low number of users there could just be a single intermediation instance, and platforms with a larger user count may balance the load among multiple instances. To prevent undue stress on the other components, the intermediation instances will cache rarely changing information, such as credentials and permissions.

4.3.2. Intermediation Strategy

There is only one intermediation strategy implemented at present, which we refer to as *single server intermediation*. Basically, all interactions are directed towards one intermediation server. Suppose we have an intermediator instance running on at `https://intermediation.domos-platform.eu/`. When the server receives a request to

```
https://intermediation.domos-platform.eu/1234-9876/4567-
2109/temperature/1?unit=celcius,
```

by an application service who previously read an external thing description containing such an URL, it decomposes the information as if it were a URI template:

```
https://intermediation.domos-platform.eu/{container-id}/{thing-id}/{affordance-
name}/{form-index}{?queryParams}
```

to determine that the application service wishes to:

- Interact with a thing residing in container 1234-9876.
- The id of the thing is 4567-2109.
- The name of the affordance for the interaction is ‘temperature’.
- It wishes to use the second interaction *form* (0-based indexing) from the thing description.
- It wishes to present the interaction with the query argument ‘celsius’ to parameter ‘unit’.

Then the intermediation instance will

1. Determine whether the application service has permission to interact with the thing.
2. Fetch necessary data to complete the interaction.
    a. Fetch the internal thing description to find the interaction based on the affordance name and the form.

   b. Fetch the appropriate credentials to be used for the interaction.
3. Forward the request including any additional data such as the query parameters.
4. Return the response back to the application service.

Another intermediation strategy may map different things to different servers, for instance, it may load balance to `https://server{N}.domos-platform.eu/`. where `0 <= X < N` for some `N` number of intermediation instances.

4.3.3. Performance and Scaling

A quick calculation on the number of active connections and bandwidth requirements with some assumptions guides the technical implementation. Assume some modest assumptions:

- An application service wishes to make 15 interactions (read or writes) to a device every 10 minutes for every container it interacts with.
- Each container is subscribed on average by 1.5 application services.
- There are two connections per interaction. One between the application service and intermediary, and one between the intermediary and the WoT device.
- Assume a latency of 200ms per interaction.

Then the number of active interactions at any given time can be calculated using Little's law [Little's Law] as:

$$\text{ActiveConnections} = \text{NumContainers}\frac{15\text{req} \cdot 1.5 \cdot 2 \cdot 0.2s/req}{600s} = NumContainers \cdot 0.015\, req$$

And this may be considered a conservative guess, since application may try to interact at specific times of the hour like every 15 minutes on the clock, the worst-case connection count may significantly peak one or two orders of magnitude higher. We can calculate the necessary bandwidth by as:

$$\text{Bandwidth} = \text{NumContainers}\frac{0.015\, req}{0.2s/req}(RequestSize + ResponseSize)$$

Including TLS resumption, but excluding TLS session creation, then assuming an average network request size of 1100 bytes and a response size of 1300 bytes the bandwidth needed given the above assumptions is: $NumContainers \cdot 180\, bytes/second$. To handle the peak load, which we assume can be 30 times higher, a server with a 1 Gbit connection will be able to serve roughly 22,370 containers with a peak of 10,050 active connections (half between the application and the intermediary the other half between the intermediary and the WoT device). A 10 Gbit server would handle 10 times the amount. Poor usage of TLS resumption on the side of application services or WoT devices could add an additional 6kB of overhead to the requests reducing the amount to 40% of above. The calculation for the number of active connections assume connections are closed when the requests are completed; however, WoT also supports subscribing to MQTT brokers or listening for server-sent events which may keep connections open indefinitely. If 20% of interactions will establish persistent connections, then the platform must maintain on average 9 (4.5 on each side of the intermediator) persistent connections per container, or around 200,000 persistent connections in the case of 22,370 containers.

To support many resource owners in the system we cannot use a single thread-per-connection style server for the intermediation; the memory overhead is large and will put much pressure on the OS scheduler. Instead, we will use an I/O asynchronous based server and with the option of multiple servers to handle the intermediation.

Our current implementation of intermediation uses the Spring Webflux [Spring WebFlux] framework which is a reactive web application framework. Since most of our components are not reactive, we do not get the full benefits of such an architecture, but Spring Webflux still offer a non-blocking web stack that, unlike some other non-blocking architectures which only runs single-threadedly (e.g., Node.js [Node.js]), can make full use of the multiple hardware threads present. Future Java development in Project Loom may result in “virtual threads” that could be another implementation strategy. While the development investment using Spring Webflux is not free, the principle of separating the deployment of the intermediation using an intermediation strategy means that the technical implementation of the intermediation can change if necessary.

4.4. Management

The management component is responsible for storing information about the users on the platform and ownership of resources, including which containers are owned by a building owner, and what application are subscribed to the container. It also compiles information from the other components such as request metrics or health status. To retrieve these metrics from the various components, Prometheus is used. Prometheus is an open-source systems monitoring and alerting toolkit which collects, store, and expose components metrics as time series data. The component is also queried by the administration GUI to present a dashboard with health information and overview for the platform operators. When new users are added to the platform it also communicates with the IAM service to ensure the user has associated authorization.

4.5. Administration GUI

As briefly introduced in Section 3.4, the platform comes with a Graphical User Interface offering various functionalities to each type of user. Currently, only the platform operator web pages are supported, thus, in the following these pages are discussed.

Platform status. In the platform status page, information about the deployed components are presented to the user. The operator can check which components are running and which are not active. The component state can be assessed by checking the dot’s colour in the relative component’s box; if it is green coloured means that the component is running correctly, otherwise it turns into red. By clicking on the component’s box, it is possible to access the component specific page, where more insights about the component status can be accessed. Furthermore, in this page a plot showing the incoming request rate for part of the component is available. Unfortunately, only the incoming traffic of the components of which the source code can be modified is accessible, thus this insight is not available for Arrowhead

core components, MySQL, and MongoDB. In Figure 10, a screenshot of the current development progress of the Platform Status page is depicted.



FIGURE 10: SCREENSHOT OF THE PLATFORM STATUS PAGE ACCESSIBLE THROUGH THE GRAPHICAL USER INTERFACE

Component status. In the component status page, more detailed information about the component status is provided. For each endpoint exposed by the component, it is possible to check how many requests have been received, the HTTP method which has been used, and also the resulting response HTTP status code. To further enhance the operator's monitor capabilities, historical data (number of processed request and request rate) are provided in the form of lines plot. An example of this last-mentioned plot is provided in Figure 11.

Users. In the users page, a table with all the users registered in the platform is presented to the operator. Through the GUI, users can be created or deleted, and their information can be view and modified.

Devices. In the device page, the things deployed in the various building part of the domOS ecosystem can be managed. Available devices are listed, and the table can be ordered and filtered to look for a specific device. New devices can be created, and no more existing devices can be deleted. Furthermore, once a specific device has been selected, it is possible to update its information, and in particular its Thing Descriptor and Thing Credentials can be inspected and updated.

In the building page, edifices which are part of the domOS ecosystem can be managed. As for the device page, a table with the information of the various building is presented, and building can be created, deleted, and updated. In this case, once a building has been selected it is possible to modify its Building Descriptor.

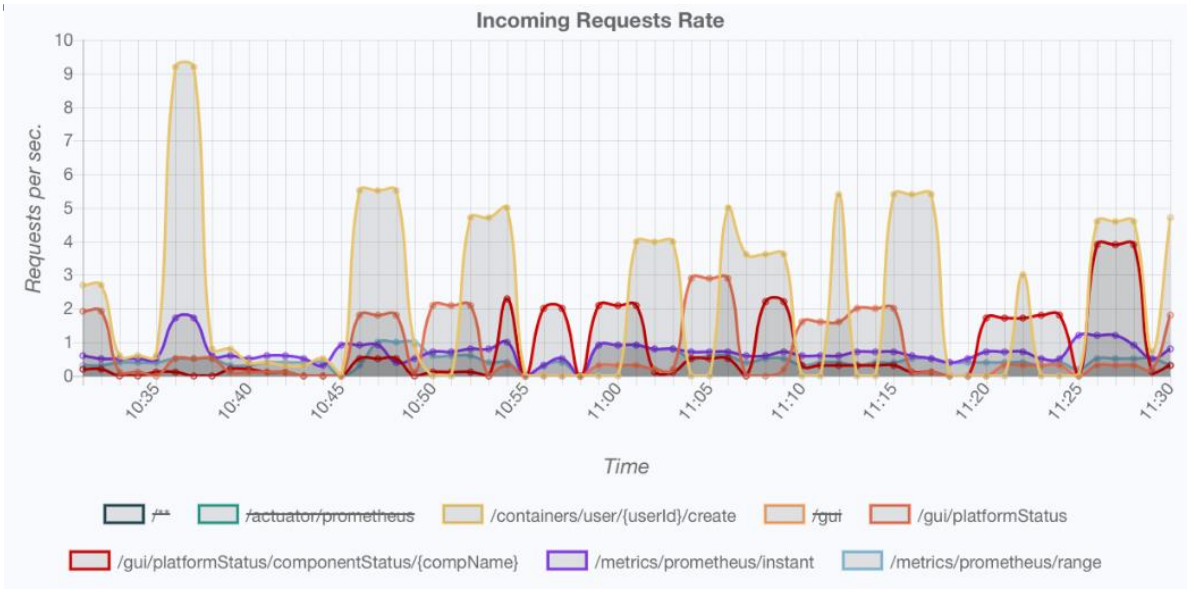


FIGURE 11: SCREENSHOT OF THE PLOT AVAILABLE IN THE COMPONENT STATUS PAGE

4.6. Permissions

The permission component is used to store and provide information about:

- User and application permissions.
- Things credentials.

For what concern the first functionality, permission documents are used to represent what permissions have been granted to users and applications in terms of reading and writing BDs and accessing to Things affordances.

In **Error! Reference source not found.**, an example of permission document is presented. A permission d ocument is identified by the pair of values "**principal-id**":"**container-id**", which expresses that the document is about the permissions granted to user/application with identifier equal to "principal-id" on the container "container-id". Then, in the document the following information are provided:

- **createdAt**: date and time of when the document has been created.
- **updatedAt**: date and time of when the document has been updated the last time.
- **buildingDescription**: defines if the user/application has read and/or write permissions on the BD of this container.
- **things**: list of permissions that has been granted to the principal. For each Thing, it is stated what properties can be read or written, which actions can be performed on the device, and to which events the principal can subscribe. This follows the "Interaction Affordances" schema defined in WoT Thing Descriptor specification.

The permission component exposes endpoints to retrieve, create, modify, and delete permissions documents.

Furthermore, the permissions component deals with the management of Thing Credentials documents. These documents are used to store the credentials needed to access the actual devices deployed in the buildings. In fact, these credentials are the ones that the thing itself will check to authorize the interaction initiated by the Intermediator. Each document is identified by the pair of value "**container-id**":"**thing-id**"; then, for each security schema defined in the thing's TD, the associated credentials are stored. An example is provided in **Error! Reference source not found.**. Even though in the provided example the d ocument is shown in clear text (i.e., not encrypted), in reality the Things Credentials documents are store encrypted using a KEK (Key Encryption Key) + DEK (Data Encryption Key) approach.

As for the permission documents, this component allows to retrieve, create, modify, and delete Thing Credential documents.


```
{
  "principalId": "7171842c-70a7-4726-b0eb-5206a70ae82c",
  "containerId": "548f251b-0ad5-4833-94a2-3f181f087f91",
  "createdAt": "Thu Jun 02 10:18:08 CEST 2022",
  "updatedAt": "Thu Jun 02 10:18:08 CEST 2022",
  "buildingDescription": {
    "permissions": [
      "read"
    ]
  },
  "things": {
    "urn:uuid:79c912fa-27e7-49c9-b3a9-762c120c1f25": {
      "read": [
        "status",
        "toggle"
      ],
      "write": [
        "setpoint"
      ],
      "action": [
        "toggle"
      ]
    },
    "urn:uuid:66a222da-56e1-89b9-b5g9-123a199c1k38": {
      "read": [
        "status"
      ],
      "write": [
        "setpoint"
      ]
    }
  }
}
```


FIGURE 12: EXAMPLE OF A PERMISSIONS DOCUMENT


```
{
  "_id":"14e2ac2d-e9d7-4d70-a8b5-170b801ec037:51c39e40-fa16-4861-be4c-360b3463f212",
  "containerId": "14e2ac2d-e9d7-4d70-a8b5-170b801ec037",
  "thingId": "51c39e40-fa16-4861-be4c-360b3463f212",
  "apikey_sc": {
    "apikey": "Apikey [redacted]"
  }
}
```


FIGURE 13: EXAMPLE OF THING CREDENTIALS DOCUMENT.

5. Identity and Access Management

The identity and Access Management (IAM) system deals with the authorization of users and applications who want to interact with the platform and with the things in the buildings. Using such a solution, Single-Sign On (SSO) is offered to the clients across the various services offered in domOS, improving their experience throughout the eco-system. Even though we choose to use Keycloak, which details are exposed in the following sections, our developed platform is compatible with every IAM system which supports OpenID. The authorization flow is based on OAuth2.1 industry-standard protocol, which enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner or on its own behalf. On top of OAuth2, OpenID is used to provide authentication functionalities. As it will be discussed in detail in Section 5.3 (Authentication and authorization Flow), external applications are authorized using the Client Credential flow, whereas Users (e.g., container owners) use the Authorization Code flow with added security through “Proof-Key for Code Exchange” (PKCE).

5.1. Keycloack General Description

Keycloak is an Open Source IAM solution for modern applications and services developed in Java by WildFly, a division of Red Hat. The first production version has been released in September 2014, and it has proved to be a reliable solution in the industry during the last years. It supports various Identity Providers (OpenID Connect, SAML 2.0, and Kerberos), Social Logins (Google, Github, and Facebook), and connection to existing external Users Providers (Active Directory, LDAP, or relational databases). Moreover, given that Keycloak complies with OAuth2.1 specification, external applications can seamlessly interact with it as they would do with every other OAuth2.1 compliant authorization server. The management of the deployed instance is possible through the Web GUI, the REST APIs and with the available Java adaptors, allowing quite straightforward and flexible instance deployment and management. In Keycloak, many different entities and concepts are available, and this provides a very fine-grained authorization process, which is of fundament importance in order to respect the domOS project requirements.

As briefly introduced in the previous section, Keycloak provides various entities and concepts to fine grain the access permissions to the various platform services. The highest and widest concept provided is the Realm one, which then is used to enclose the ones of clients, roles, and users. After that, the concepts of client’s roles and client’s scopes are assigned to the client entity to further describe its authorization properties. Meanwhile, users can be assigned to specific groups in order to represent their belonging to a specific subset of users, which can have by default different permission to act on the platform.

5.2. Keycloak Entities

In the following, the main Keycloak Entities as presented, discussed, and their connections with domOS concepts are explained. Moreover, implementation details relative to our platform are introduced, and the specific setup used is further described in Table 3.

Realm: it is an isolated group of applications, users, groups, and roles. Using multiple realms allows a separation of the underlying entities when, for example, the same Keycloak instance is used by multiple

companies, and each of them wants their users to be stored on a different database, to use different permission groups and scopes, or to adopt a different authorization mechanism. During the interaction between an application/user and the platform, when the provided Json Web Token (JWT) has to be verified, the realm public key is the one which has to be used. Given that we want to provide external applications the possibility to potentially access all the buildings managed by the domOS project (after that permission has been granted by the user), we opt to use a single Realm to manage all users and all applications registered in domOS. In this way, once that an application has been registered in the domOS platform, it is already accessible by a large audience of potential customers.

Clients: Clients are applications and services that interact with Keycloak to authenticate users and obtain tokens. Once that a valid token has been retrieved, clients can act on behalf of a user or on their own behalf, and access protected resources. In our case, each external application which wants to interact with domOS platform and offer services to the users, has to be registered in Keycloak in order to be able to be authenticated and authorized. Furthermore, a client is created for each component deployed in the platform to improve the authorization process performance across internal services (further details are provided in the Authentication and Authorization Flow Section).

Roles: A role identify the type or a category of user or client. They can be seen as a first level of authorization; indeed, assigning a specific set of roles allows to manage which services can be access and actions can be performed with the given token. If instead a more fine-grained level of authorization is needed, client scopes should be used. The roles which are needed to manage realm, clients, and users are assigned to the management component to allow the execution of CRUD operations on these entities.

Client Roles: Client roles are namespaces dedicated to clients. In these namespaces it is possible to define clients' specific roles. These roles are used to authorize interaction between platform's components. For instance, in the TDD component client we defined two client roles which allow direct read or write functionality to the TDs, respectively called "domos-td-read" and "domos-td-write". Then we assigned the "domos-td-read" client role to the intermediator component, in this way allowing it to be authorized for reading TDs without the need to interact with the permissions component.

Client Scopes: Client scopes are used to provide a more fine-grained authorization system to the platform. According to OAuth2.1 specification, a scope is a way to limit the amount of access that is granted to an access token. Consequently, clients' scopes are nothing more than a string containing a space-separated list of scopes associated with the token. In the platform, in order to decline unauthorized request as soon as possible - that is, without the need to interact with the permission component - scopes are used to check if the requesting entity has access to the container it is trying to interact with. Thus, the scope claim contains the list of the containers ids which the application has been granted access to. This also means that every time a new container is registered into the platform, the scope associated to the just inserted container needs to be created in Keycloak.

Users: they are entities that are able to log into the system. They can have attributes associated with themselves like email, username, address, phone number, etc, and they can be assigned group membership and have specific roles assigned to them. Note that, in Keycloak, every time that a new client



is create in Keycloak, a “service user” is automatically created, and it is the user that will be specified in the token presented to the platform by the external application. This is to be consistent with Keycloak logic that users are always the entities being authorized to access the platform.

Groups: this entity defines a common set of attributes and roles for a set of users. It eases the identification of what kind of operation the user is allowed to perform on the platform. The three different user types defined in domOS are mapped in Keycloak to the following groups: customers, platformOperators, and serviceProvider. Moreover, we defined a fourth group, called componentsServiceUsers, which includes the service users automatically created by Keycloak when the domos component client is instantiated.

TABLE 3: KEYCLOAK ENTITIES CONFIGURATION FOR THE DOMOS ECOSYSTEM

Entity Name	*Description*	*domOS-Arrowhead platform Defined Entities*
Realm	Name of the realm defined for the domOS ecosystem.	domOS
Clients	List of the defined clients. Each domos component deployed in the platform has its own associated client instantiated in Keycloak. Also, every internal/external application has a relative client defined.	• domos-component-bdd • domos-component-tdd • domos-component-intermediator • domos-component-management • domos-component-permissions • nodejs-app-demo (External Application) • arrowhead-app-demo (Internal Application)
Roles	*Not Used.*	*Not Used.*

Client Roles	Each domos component has specific roles used to authorize interactions inside the arrowhead cloud. For each component, the list of available Client Roles is provided.	• domos-component-bdd
		  ○ domos-db-read
		  ○ domos-bd-write
		• domos-component-tdd
		  ○ domos-td-read
		  ○ domos-td-write
		• domos-component-intermediator
		  ○ *Not used*
		• domos-component-management
		  ○ domos-containers-read
		  ○ domos-container-write
		  ○ domos-users-read
		  ○ domos-users-write
		• domos-component-permissions
		  ○ domos-permissions-read
		  ○ domos-permissions-write
		  ○ domos-thingCredentials-read
		  ○ domos-thingCredentials-write

Client Scopes	List of scopes used to identify to which containers the token gives access to.	*List of the id of the containers managed by the domOS ecosystem.* *e.g., “548f251b-0ad5-4833-94a2-3f181f087f91”*

Users	Entities that can authenticate and request an access token.	*List of the users registered into the platform, including customers, platform operators, service providers, and component service users.*
Groups	List of the groups defined in the domOS ecosystem.	• Customers • PlatformOperators • ServiceProviders • ComponentsServiceUsers

5.3. Authentication and Authorization Flow

Two different types of authentication and authorization flows can occur on the platform:

- User Authentication.
- Application Authorization.

The User Authentication case takes place when the user wants to login into an external application or into the platform using the user's personal credentials. For this purpose, the OAuth2 Authorization Code with PKCE code flow is used; this flow ensures that user's credentials are never disclosed to third parties, thus increasing the overall platform and users' security. Moreover, it provides to the user the beforehand mentioned SSO functionality. The various steps of the grant flow are illustrated in Figure 14. Here we consider that the user is trying to be authenticated through an external application; if instead the user is being authenticated directly on the domOS platform, the application entity has to be substituted with the platform instance.

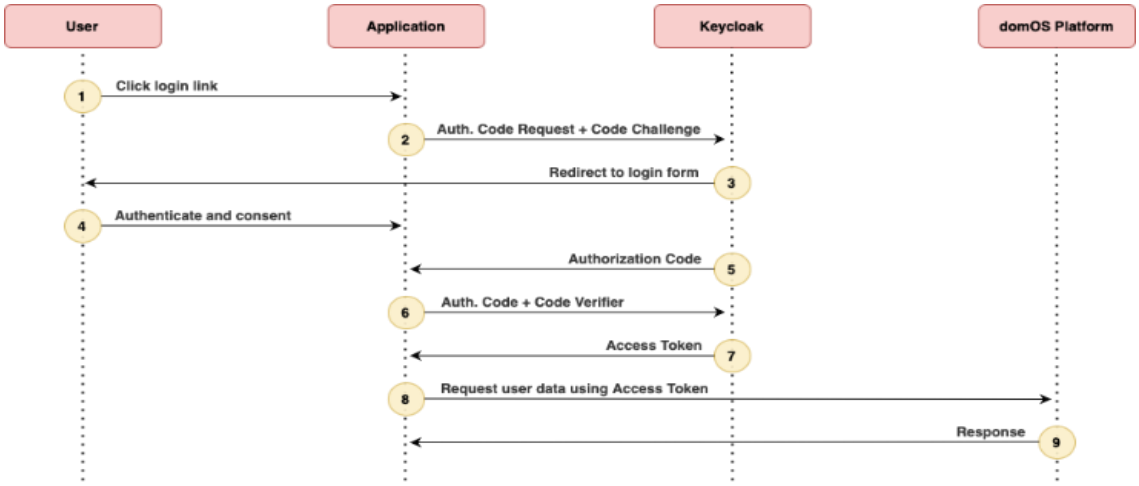


FIGURE 14: STEPS REQUIRED TO AUTHENTICATE A USER IN THE PLATFORM THROUGH AN EXTERNAL APPLICATION USING THE PKCE-ENHANCED AUTHORIZATION CODE FLOW

In the Application Authorization case, applications are authorized to access the platform using the Client Credentials flow. In this flow, which is illustrated in Figure 15, the pre-shared client-id and client-secret have to be provided by the application to the AIM in order to get the authorization token. These two authentication factors are supplied from the platform operator to the external application during the registration of the application as "accredited" in the domOS eco-system.

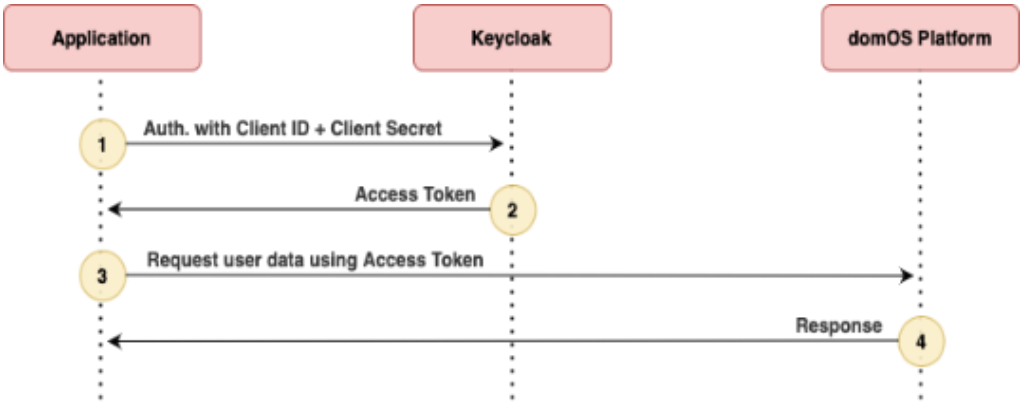


FIGURE 15: STEPS REQUIRED TO AUTHENTICATE AN EXTERNAL APPLICATION IN THE PLATFORM

In Figure 16, an example of JWT released by Keycloak for the Intermediator component is depicted. To improve document presentation, JWT's signature has been omitted. We now briefly describe which claims can be found in the token and which their meaning and usage are.

For what concern the token's header, three claims can be found:

- **alg:** The algorithm used to sign the JWT. RSA Signature with SHA-256 is used.
- **typ:** The type of token. It is used to indicate that the content of the token is a Json Web Token.
- **kid:** The identifiers. It identifies which key has been used to sign the token.

In the token's payload, a number of claims are inserted to provide fine-grained authentication and authorization functionalities:

- **exp:** token expiration timestamp.
- **iat:** token creation timestamp.
- **jti:** token unique identifier.
- **iss:** issuing entity.
- **aud:** token audience. List of "who" or "what" the token is intended for.
- **sub:** technical identifier of the user.
- **typ:** authentication schema.
- **azp:** authorized party. It states the party to which the token was issued.
- **resource_access:** Keycloak optional claim. It in turn contains the list of client roles associated to the client requesting the token.
- **scopes:** list of scopes the token gives access to.
- **clientId:** identifier of the client who requested the token.
- **clientAddress:** IP address of the client who requested the token.

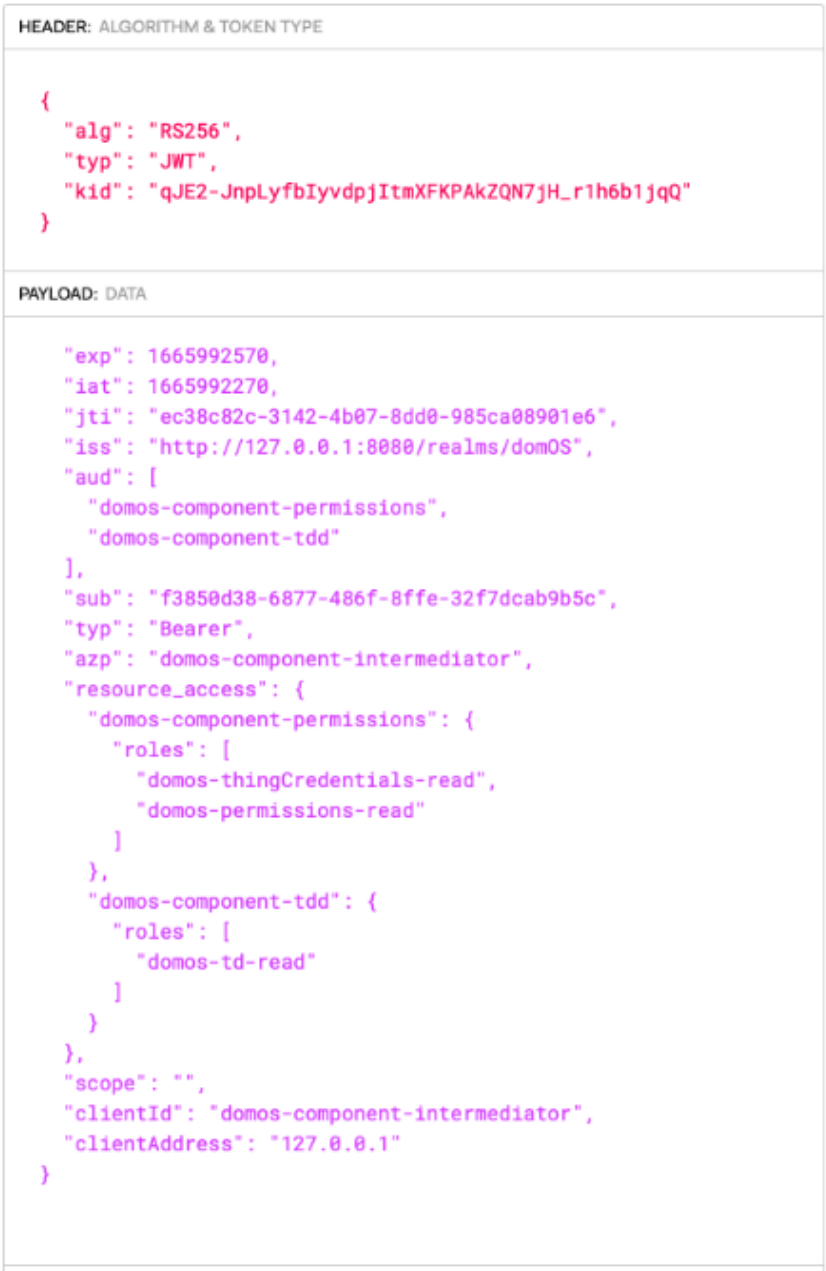


FIGURE 16: EXAMPLE OF A JWT RELEASED TO THE INTERMEDIATOR COMPONENT

6. DomOS Application Specification

The DomOS-arrowhead platform is implemented using the Arrowhead framework as the basic building block specified in Section 3.

That opens the way for two possibilities when deciding to implement an application which will interact with the platform.

The first possibility is a complete Arrowhead application which complies with the arrowhead application specification. However, for the platform not to be “closed” by adding strong constraints on developers and enabling interoperability, we opened a way for external applications that are not “Arrowhead” based to be developed and interact with the platform. In this section, we define the specification for both approaches. The domOS application is WoT compatible.

6.1. Arrowhead Application Specification

The Arrowhead framework is currently used to support application development in many business domains such as industrial processes and manufacturing, Smart Buildings, and infrastructures. The Arrowhead framework applies the Service Oriented Architecture (SOA) to the industrial domain by having distributed systems designated as service providers making their services known to services from systems designated as consumers, which have a way to discover and consume them. This is the case for the domOS Arrowhead platform, with domOS core components (Thing Description Directory, Building Description Directory, Intermediator) implemented as service providers.

For an application to interact with the core components, this application needs to be implemented following the Arrowhead Application skeleton [https://github.com/arrowhead-f/client-skeleton-java-spring].

The goal is to have an application which can initiate an orchestration request and consume the service from the different providers (core domOS components).

Applications are built using the Arrowhead Application Library [https://github.com/arrowhead-f/application-library-java-spring], which should be imported as a Maven dependency. This application library provides the ArrowheadService.class, a singleton Spring-managed bean designed to interact with the Arrowhead Framework.

An overview can be seen in Section 7 with an end-to-end interaction between an arrowhead application and the platform.

6.2. External Application Specification

The domOS Arrowhead platform has been implemented so that an application written in another language than Java (the main language for Arrowhead), which is not Arrowhead compatible, can still interact with it. The domOS core components offer various APIs for this interaction purpose. The main requirement is that the application should be able to have an HTTP client for the different requests.

Section 7.1 gives an example of end-to-end interaction between a nodejs application and the platform.

7. End-to-end Interaction Example

Different interactions are possible. First, consider a case where there is no information about the thing for which the interaction is needed. In this case, the application first needs to query the BDD to get the building description so that it knows available things and obtain the id of the thing it needs to interact with. Using that id, it will interact with the TDD to get a thing description containing the information to interact with the thing. It is good to mention that the TD returned at this level is the modified version, where all available links point to the Intermediator.

For this part, the interaction process differs if the application is internal (Arrowhead application) or external (non-arrowhead application).

The next step is to interact with the Intermediator for interacting with the thing on behalf of the application. This interaction flow is the same for both types of applications.

7.1. Interaction Between Internal Application and Platform

This interaction (Figure 17) presents the case where an arrowhead application needs to interact with a thing.

We describe the part specific to arrowhead interaction and the common interaction described in Section **Error! Reference source not found.**.

1. Application sends an orchestration request to the Arrowhead Orchestrator for BDD's service providing building descriptions.
2. Orchestrator queries the Service Registry to get entries related to the requested service.
3. Service Registry provides entries available.
4. Orchestrator checks from the Arrowhead Authorization if the requesting application is authorized to interact with the service.
5. Application is authorized.
6. Orchestration response to the application.
7. Application can consume the service from the BDD.
8. BDD checks the permission component to ensure the application can read the container's BD.
9. Application has read access.
10. Application gets the building description.

Now that the building description has been retrieved, the application will try to get the Thing description for the thing it wants to interact with using the thing Id. The previous interaction is performed, but this time with the TDD involved.

Interaction with the thing is described in Section **Error! Reference source not found.**.

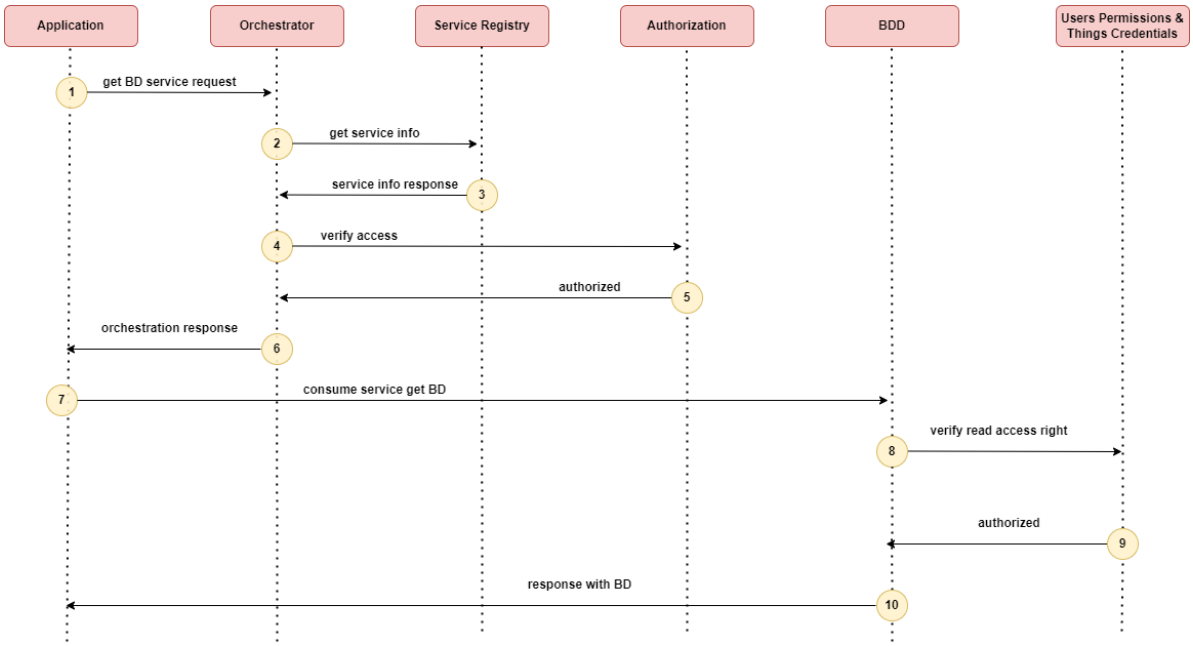


FIGURE 17: INTERNAL APPLICATION INTERACTION

7.2. Interaction Between External Application and Platform

This interaction (Figure 18**Error! Reference source not found.**) presents the case where a non-a rrowhead application needs to interact with a thing. We describe the part which is specific to this type of application.

1. Application query the BDD for building description.
2. BDD checks the permission component to ensure the application can read the container's BD.
3. Application can read that BDD.
4. Application gets the building description.

A similar interaction is performed with the TDD to get a thing description.

Interaction with the thing is described in Section **Error! Reference source not found.**.

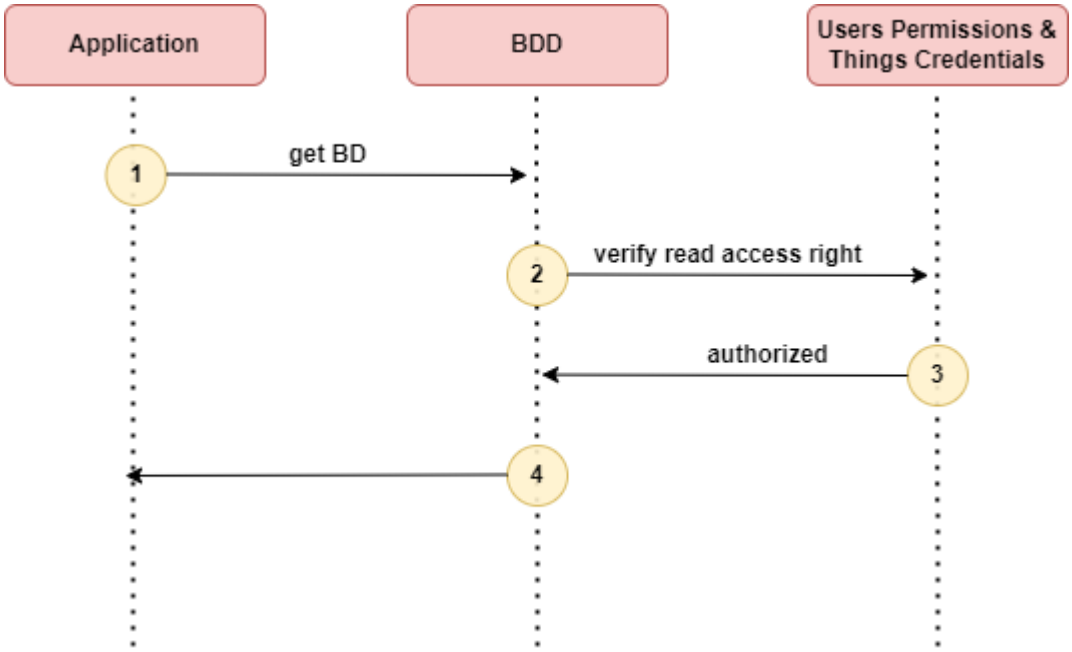


FIGURE 18: EXTERNAL APP INTERACTION

7.3. End-to-end Interaction: From Application via Platform to Thing

In this section, the whole end-to-end interaction to access a Thing from an Application is described, paying particular attention to the authorization process.

As it has been introduced in Section 5.2, client roles are used to improve the performance of the authorization process inside the platform. Without the adoption of this approach, each interaction in-between the internal components deployed inside the platform would require a cumbersome forwarding of tokens and multiple interactions with the permission component. Instead, by instantiating each domos component as a client in Keycloak, by defining its own set of client roles, and by assigning to it the required client roles of the components which it needs access to, it is possible to drastically reduce the number of requests sent to the permission component.

In Figure 19 the flow is depicted, and hereby it is described in detail:

1. The interaction flow start with the Application trying to read/write a Thing property through the Intermediator. In this step, Application's JWT is presented.
2. If the Application's JWT is valid, Intermediator checks on the Permissions component that the application is authorized to access that property of the Thing. When Intermediator interacts with the Permissions component, Intermediator's JWT is used; this token contains the client role "domos-permissions-read", which allows to directly access the permission component and read the users permissions documents.
3. Permissions component checks if the required client role is present in the token, and, if it is, replies to the Intermediator if the Application is allowed to access the Thing's affordance.

4. If the Application has the right to access the Thing, Intermediator request Thing's TD to the TDD. Again, Intermediator's JWT is used, and because it contains the "domos-tdd-read" client role security checks are bypassed.
5. TDD checks if the required client role is present in the token, and, if it is, forwards back to the Intermediator the requested TD.
6. The last step before the Intermediator is ready to interact with the Thing is to retrieve the Thing's credentials. These credentials are retrieved through another request to the Permission component.
7. Permission component check that the JWT used by the Intermediator contains the client role "domos-credentials-read". If the condition is satisfied, Thing Credential are forwarded to the Intermediator.
8. Intermediator has now all the information needed to interact with the Thing, thus forwards Application request to the device.
9. Thing check the provided credentials are correct, elaborate the request, and send beck the response to the Intermediator.
10. Finally, Intermediator forwards Thing's response to the Application, and the interaction flow is concluded.

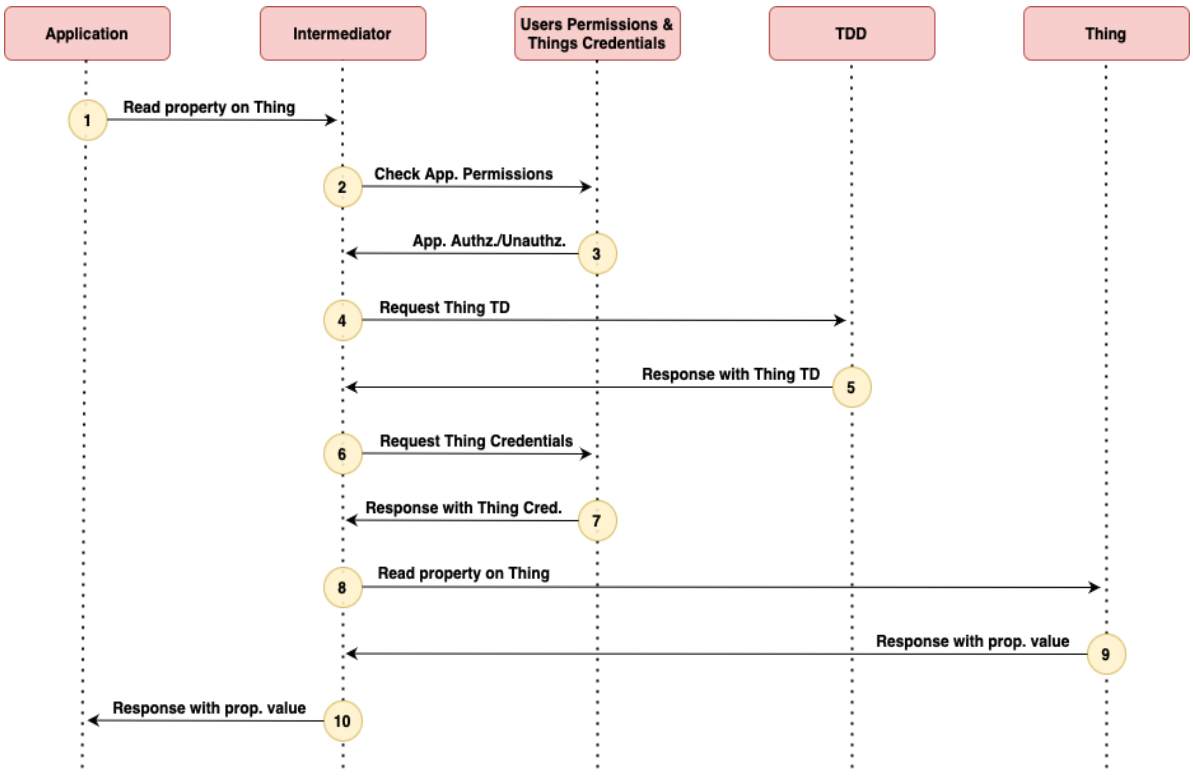


FIGURE 19: END-TO-END INTERACTION FLOW

8. NEOGRID Platform Things Provisioning

Like many companies, NEOGRID have their own custom designed data structures for describing buildings and things. These have been tailored over more than 4 years to fit the use cases within NEOGRID's commercial products and research activities, and all existing components; data collection, control systems, data visualisation, and many more, are built around these data structures. In order to integrate these existing platforms into the domOS-Arrowhead platform, it will often be more feasible do a platform-to-platform integration, where the internal data models are left intact and merely converted to/from the domOS-Arrowhead platform. This is the same approach if an existing IoT solution like Phillips Hue should connect to Google Home – cloud to cloud.

In the case of the demonstrators in Aalborg and Skive, this means we must 1) convert NEOGRID's internal building model into things (and building) descriptions and 2) make these documents, and the included URLs for interacting with the things, available for the applications running in the domOS-Arrowhead platform.

8.1. Generation of Things Descriptions

NEOGRID utilises a *building model*, an abstraction layer between devices and building context. The model not only describes the building in a uniform way for applications, but it also helps manage the building over time. For example, it segments the building into *zones* for more fine-grained control, and decouples the physical devices from virtual data points, such that the physical sensors can be replaced without losing historical measurements.

The devices in one of the domOS single family homes, part of the Skive Demonstrator, are depicted in Figure 20, as these are shown within Neogrid's configuration tool.

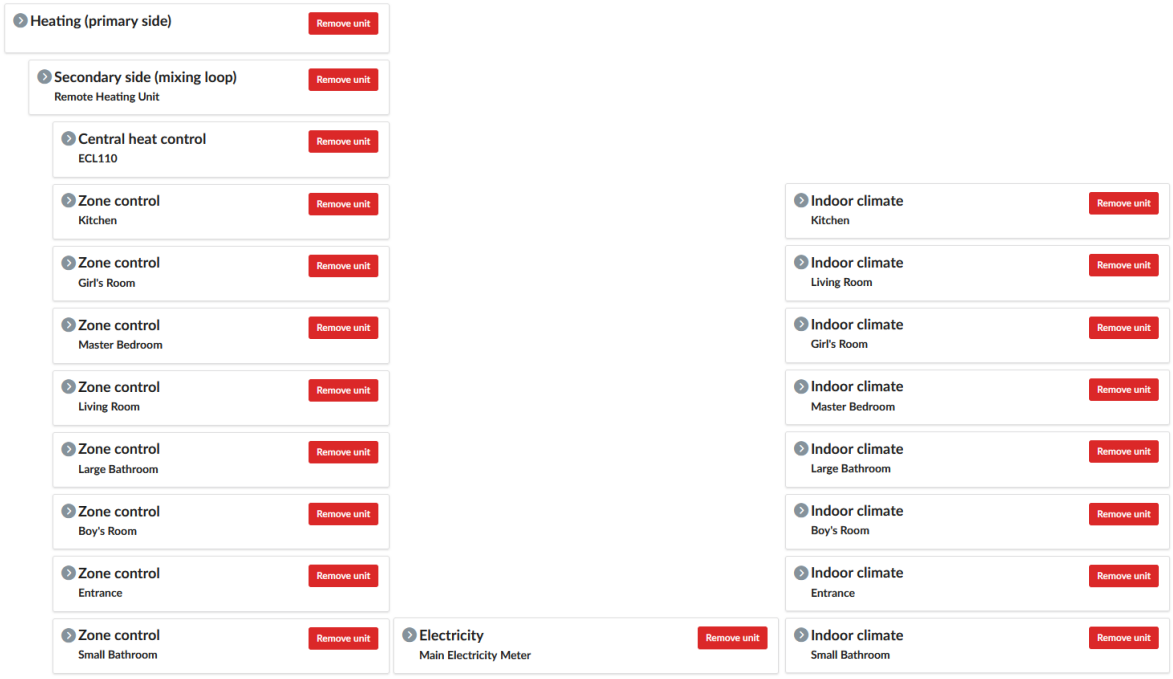


FIGURE 20: DOMOS SINGLE FAMILY HOME IN NEOGRID'S CONFIGURATION TOOL

The building has 8 zones, 3 bedrooms, 2 bathrooms, a kitchen, a living room, and an entrance. The *Heating (primary side)* is the main heating meter, measuring the forward and return temperatures of the remote heating supply to the house. Each device can be configured with a number of properties that can be read or set, in case only forward and return has been configured. The main meter supplies a mixing loop on the secondary side, where we again can read the supply and return temperature, but on the secondary side this time. The mixing loop has a control unit, *Central heat control*, which can be used to lower the forward temperature between the mixing loop and the rooms. Running the heating system with a lower temperature result in less loss of heat to the environment. The heating loop then in turn supplies each room, in this case controlled by Vicky thermostats with the properties shown in Figure 21.

Setpoint (readable)	ECL110 NEOGRID_HEAT_SETPOINT
Setpoint offset	
Controlled signal	ECL110 HEAT_FORWARD_TEMP (S3)
Control input (writable)	ECL110 NEOGRID_HEAT_SETPOINT
Motor position	
Pump on/off	

Setpoint (readable)	Kitchen Set_Kitchen
Controlled signal	Kitchen Temp_Kitchen
Control input (writable)	Kitchen Set_Kitchen
Motor position	Kitchen Pos_Kitchen
Pump on/off	Kitchen Motor range

FIGURE 21: PROPERTIES ON THE CENTRAL HEAT CONTROL (LEFT) AND KITCHEN THERMOSTAT (RIGHT)

Besides the heating devices, the house also consists of an electricity main meter, and an indoor climate sensor in each zone, measuring the ambient temperature and humidity.

8.1.1. Conversion to domOS Ontology

The models within Neogrid's existing platform first must be translated into things descriptions. Currently, this is an automated process of traversing the existing tree of devices, in the building model, and converting each device to a thing in the domOS ontology.

Only a subset of device types is recognised at the moment, but the implementation is flexible, making it easy to add more types moving onwards. Devices supported, at the time of writing, are primary meters for heating, hot water, electricity, and cold water, secondary meters for heating and hot water, indoor climate sensors and control devices for heating and hot water. As with device types, properties are easily added to the translation, as they are needed.

The conversion of the kitchen indoor climate sensor, from Figure 20, to a domOS "thing" is shown in Figure 22. This includes translating its two properties temperature (shown) and humidity (collapsed). All the devices in Figure 20 have similar translations within the domOS ontology, which are combined into a single thing description document.

```
{
  "security": [ "apikey_sc" ],
  "description": "Indoor climate sensor",
  "id": "neogrid-unit-15207",
  "title": "Kitchen",
  "@context": [ … ],
  "actions": {},
  "securityDefinitions": { "apikey_sc": { "scheme": "apikey", "in": "header", "name":
"Authorization", "description": "Apikey for the Neogrid Public Api" } },
  "events": {},
  "properties": {
    "temperature": {
      "@type": "dco:temperature",
      "readOnly": true,
      "type": "object",
      "properties": {
        "value": { "unit": "dco:celcius", "type": "number" },
        "timestamp": { "type": "string" }
      },
      "forms": [
        {
          "op": "readproperty",
          "href": "/public/api/wot/v1/property/32161",
          "htv:methodName": "GET",
          "contentType": "application/json"
        }
      ]
    },
    "humidity": { … },
  "base": "https://api.neogrid.dk"
}
```

FIGURE 22: KITCHEN INDOOR CLIMATE SENSOR IN DOMOS ONTOLOGY

At the time of writing, the building descriptions are still being designed, so currently these are created manually. However, when the design of the building descriptions stabilises, we will investigate the possibilities to automatically generate these as well.

8.2. Neogrid WoT Interface

Neogrid already provides a custom REST API for external users to fetch a building model for their location and interact with their data and devices[1]. The API is protected by API keys that the users must create within the web UI.

This existing API have been extended with a WoT endpoint `/public/api/wot` to support the domOS-Arrowhead platform. The features implemented in this endpoint are:

- `/v1/locations/{locationId}`: Constructs and returns a things description, as described above, for a location with the given location id.
- `/v1/property/{id}`: Read a thing property.
- `/v1/setpoint/{controlUnitId}`: Read a set point.
- `/v1/setpoint/{controlUnitId}`: Write a set point.

The above endpoints are available for all locations within Neogrid's system, not only those participating in the domOS demonstrators. However, as stated above, only a subset of the device types is currently available in the things descriptions and thereby in the WoT interface.

8.3. Integrating Into domOS-Arrowhead

At present we use the Neogrid API to manually query for all the TDs for a particular building and then upload them into the TDD with a client operating as a building owner. Our platform does not require any modifications to the TDs before insertion, and credentials are inserted to accompany into the credentials for each TD.

9. Conclusion

We have presented the design and implementation of the domOS Arrowhead platform for providing interoperable access of smart energy serves to WoT equipped buildings. It exploits the mature Arrowhead platform in two ways: first as a vehicle for the execution of the platform components, and second, as a support vehicle for complex energy service applications that are structured according to a (micro) services-oriented architecture. The platform is designed to be scalable and open, and to support fine-grained access control. It offers both secure access to external applications as well as internal applications.

We conclude that we have a stable and sophisticated platform that is ready for further demonstration and interoperability tests.

[1] Neogrid REST API documentation is available at https://neogrid-technologies.gitlab.io/neogrid-api/

Some implementation aspects remain to be implemented: Things-device health monitoring and recovery, device lifecycle management, as well as user interfaces for managing buildings and owners, building description editor, and access-control editor.

As future work, it could be interesting to develop a more high-level model for specifying privacy policies in addition to the low-level access control mechanism currently supported. We also consider developing generic services (logging, visualization, time-series storage, forecast-algorithms) that may be useful components for the development of smart energy services.

10. References

10.1. WoT Standards

[Wot UCR] Web of Things (WoT): Use Cases and Requirements, 27 January 2021, https://w3c.github.io/wot-usecases/

[WoT Arch] Web of Things (WoT) Architecture, 9 April 2020, https://www.w3.org/TR/wot-architecture/

[WoT TD] Web of Things (WoT) Thing Description, 9 April 2020 (corrected 23 June 2020), https://www.w3.org/TR/wot-thing-description/

[WoT Binding] Web of Things (WoT) Binding Templates, 30 January 2020, https://www.w3.org/TR/wot-binding-templates/

[WoT Scripting] Web of Things (WoT) Scripting API, 24 November 2020, https://www.w3.org/TR/wot-scripting-api/

[WoT S&P] Web of Things (WoT) Security and Privacy Guidelines, 6 November 2019, https://www.w3.org/TR/wot-security/

[WoT Discovery] Web of Things (WoT) Discovery (Working Draft), 2 June 2021, https://www.w3.org/TR/wot-discovery/

[WoT Arch1.1] Web of Things (WoT) Architecture 1.1 (draft), November 2020, https://www.w3.org/TR/2020/WD-wot-architecture11-20201124/

10.2. Bibliography

[Arrowhead] Jerker Delsing (Eds.) ***IoT Automation- Arrowhead Framework***. 2017.Taylor & Francis Inc. ISBN 9781498756754

[Arrowhead wiki] "Arrowhead Framework Wiki", url: https://forge.soa4d.org/plugins/mediawiki/wiki/arrowhead-f/index.php/Main_Page

[Arrowhead github] https://github.com/eclipse-arrowhead/core-java-spring

[domOS D3.2] Amir Laadhar, Christian Thomsen. DomOS deliverable D3.2 Domos Common Ontology – initial version https://www.domos-project.eu/filedelivery.php?docId=17

[domOS D2.1] Frédéric Revaz, Dominique Gabioud, Domos Deliverable 2.1: Requirement Analysis for IoT Ecosystem. https://www.domos-project.eu/filedelivery.php?docId=20

[domOS D2.2] D2.2 Report on In-Building Infrastructure for Smart Services

https://www.domos-project.eu/filedelivery.php?docId=19

[domOS D2.3] Brian Nielsen, Søren Enevoldsen, Junior Dongo, Amir Laadhar, Nicola Cibin. domOS deliverable D2.3 Functional Specification of the IoT-Ecosystem. https://www.domos-project.eu/filedelivery.php?docId=21

[JLL 2016] J. Jokinen, T. Latvala and J. L. Martinez Lastra, "***Integrating smart city services using Arrowhead framework***," IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, 2016, pp. 5568-5573, doi: 10.1109/IECON.2016.7793708.

[KVH 2019] D. Kozma, P. Varga and C. Hegedüs, "***Supply Chain Management and Logistics 4.0 - A Study on Arrowhead Framework Integration***," 2019 8th International Conference on Industrial Technology and Management (ICITM), 2019, pp. 12-16, doi: 10.1109/ICITM.2019.8710670.

[Le Guilly et al. 2016] Thibaut Le Guilly, Petur Olsen, Thomas Pedersen, Anders P. Ravn, Arne Skou. ***Model-checking Feature Interactions***. Journal of Communications in Computer and Information Science, Vol 586, pp 207-325, Springer, 2016.

[Le Guilly 2016] Thibaut Le Guilly. ***Model-based analysis of Embedded Software for smart-homes***. PhD Thesis, Aalborg University, 2016

[Little's Law] https://en.wikipedia.org/wiki/Little's_law

[MoSCoW] https://en.wikipedia.org/wiki/MoSCoW_method

[MSBVD 2020] Marcu I, Suciu G, Bălăceanu C, Vulpe A, Drăgulinescu A-M. ***Arrowhead Technology for Digitalization and Automation Solution: Smart Cities and Smart Agriculture***. Sensors. 2020; 20(5):1464. https://doi.org/10.3390/s20051464

[Node.js] https://nodejs.org/en/

[OAUTH2] OAuth working group: ***The OAuth 2.0 Authorization Framework*** https://oauth.net/specs/

[OOAD] Mathiassen, Munk-Madsen, Nielsen, Stage: ***"Object Oriented Analysis & Design"*** 2ed. 2018. Metodica. ISBN: 978-87-970693-0-1

[Pedersen 2018] Thomas Pedersen. ***Smart-home models – Analysis, Simulation, and Synthesis***. PhD Thesis, Aalborg University, 2018, ISBN 978-87-7210-189-7

[Reference Model] ISO/IEC 30141:2018 EN https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

[SGAM 2021] CEN-CENELEC-ETSI Smart Grid Coordination Group. ***Smart Grid Reference Architecture. 2012.*** https://ec.europa.eu/energy/sites/ener/files/documents/xpert_group1_reference_architecture.pdf seen 03-12-2021

[Spring WebFlux] https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html

[Swagger] https://swagger.io/resources/open-api/

[Wot-Arrowhead] Junior Dongo, Michele Albano, Arne Skou, and Brian Nielsen**. Web of Things Interoperability for the Arrowhead Framework**. In IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society