Adaptive Gateways for dIverse muLtiple Environments

D5.1

First Prototype of the AGILE Identity Management System

Project Acronym	AGILE	
Project Title	Adaptive Gateways for Diverse Multiple Environments	
Project Number	688088	
Work Package	WP5 Gateway Security, Data Provenance & Access Control	
Lead Beneficiary	UNI PASSAU	
Editor	Juan David Parra	UNI PASSAU
Reviewer	Andreas Menychtas	BioAssist
Reviewer	Carlo Curinga	Resin.io
Dissemination Level	Public	
Contractual Delivery Date	31/12/2016	
Actual Delivery Date	28/12/2016	

Version	V1.0

Abstract

Before defining security policies of any kind, entities in a system need to be authenticated. To achieve this, identity management components provide authentication mechanisms and administer attributes and credentials for the system's entities. In this deliverable we introduce AGILE IDM (Identity Management) and its relation to the gateway's architecture, the model chosen to handle entities attributes, and functionalities provided by AGILE IDM to gateway components and applications. Last but not least, some examples of use, tutorials (also available in github) and additional resources for piloting activities and developers are explained.

Document History

Version	Date	Comments
V0.1	29/11/2016	Creation of Table of Contents
V0.3	05/12/2016	Add initial content architecture and some tutorials
V0.4	13/12/2016	Inclusion of full description of examples
V0.5	15/12/2016	Complete chapters 1,2,4 and 5.
V0.7	16/12/2016	Update Cloud Integration after WP4 telco
V0.9	21/12/2016	Finish the first complete version
V1.0	28/12/2016	Integrate comments from Andreas and Roman Sosa

Table of Contents

List of Figures

Acronyms

Acronym	Meaning
AGILE	Adaptive Gateways for dIverse muLtiple Environments
IdP	Identity Provider
Oauth2	Open Authorization version 2

1 Introduction

The AGILE gateway collects and handles data coming from different sensors; moreover, this data can be sensitive for the user, e.g. quantified-self scenario. As a result, the security framework developed in AGILE's Work Package 5 is focused on ensuring that users can define policies on who can access their data, and under which conditions.

To achieve this, setting up an identity management is paramount, in order to have some kind of access (or usage) control model for the data. As a result, AGILE IDM not only supports user authentication (as the term identity management is commonly used), but it also supports the definition of entities and attributes in order to allow the definition of access policies for said entities.

In a nutshell, the main responsibilities of AGILE IDM are:

- Support components developed by the consortium by providing users' authentication.
- Support developer's application and workflows to rely on AGILE IDM to authenticate AGILE users and obtain AGILE tokens to execute operations through AGILE APIs integrated with the security framework.
- Management of entities registered in AGILE through:
    - Entity groups
    - Attribute modification
- Handling of potentially private information related to entities in AGILE, e.g. tokens or credentials, required to authenticate with external services.

In AGILE we foresee to have at least two kinds of users (admin users and lower privileged users). Although AGILE IDM can do more than role-based access control (Bishop 2010), since it is a generic attribute-based identity management, it has been configured to support roles and enforced policies based on users who have the role "admin" or not. Notwithstanding, AGILE IDM could either support more roles by changing configuration files, or also use attribute-based rules, based on attribute-based access control (Hu, et al. 2013), which are more flexible, and potentially more manageable in more complex environments, than role-based access control.

This document is organized as follows: first we describe the relation between IDM and the AGILE software architecture, as well as components interacting with IDM and its internal modules. Afterwards, we explain the two core concepts handled by identity management as follows. Initially, we describe the characteristics offered by the **user authentication** mechanisms supported by IDM. Then, we cover the second key building block if IDM, which is the **generic attribute-based identity model** and its realisation. Subsequently, a high level list of functionalities offered by this component and their relation to the project requirements are covered. Next, we briefly explain examples of use and tutorials on how to extend AGILE IDM. And last but not least, we reference additional resources available for developers.

Architecture

This chapter introduces both, the relation to the AGILE software architecture, as well as the internal components and their interaction within AGILE IDM. We start by relating AGILE IDM to the whole AGILE architecture.

1.1 Big Picture

The AGILE software architecture shown in Figure 1 depicts the micro-services approach taken by AGILE, and includes separate components with particular responsibilities. As already mentioned in the requirements specification and software architecture (Charalampos and Erdeniz 2016), Work Package 5 components are represented by one module, i.e. **Access Control and Authentication** component highlighted in Figure 1. Also, the Security API Implementation shown in the architecture is comprised of several APIs. At this stage in the project, the only API available so far to external components through a REST interface is the AGILE IDM API as well as the Oauth2 endpoints for authentication. In the future the API will be extended with services to provide policy management, encryption, etc.

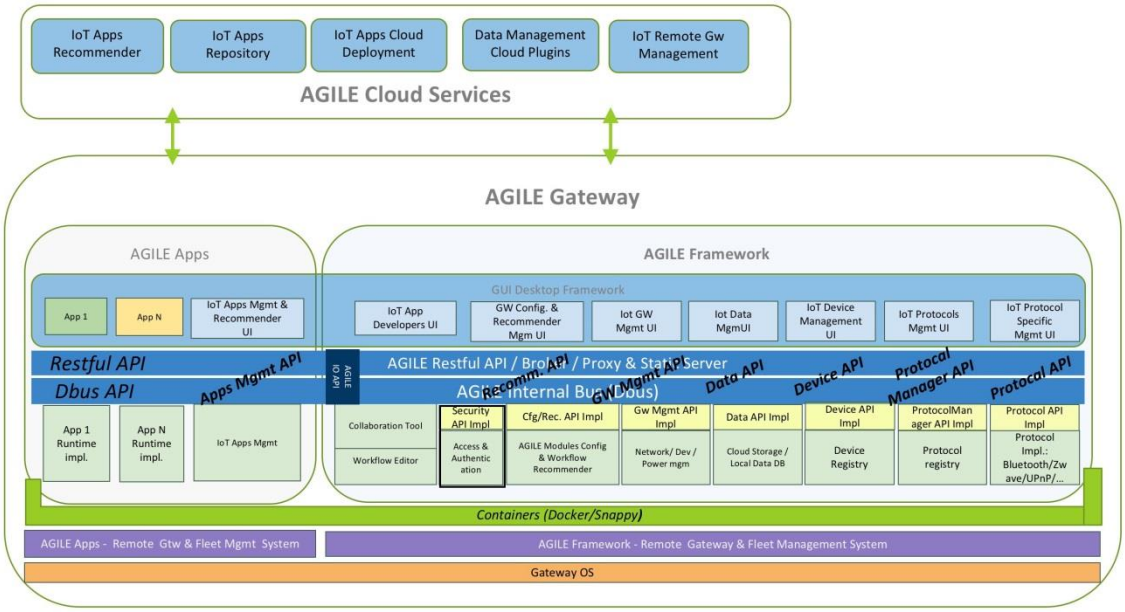

Figure 1: AGILE Software Architecture Development View

1.2 Components interacting with IDM

AGILE IDM should be integrated with at least the following components from the AGILE Framework:

- GUI Desktop framework: this allows users to login to the AGILE web interface. In turn, this opens the possibility to integrate the sub components accessible through the web

interface to use the same access token provided by AGILE IDM during the log in process, e.g. IoT Data Management UI.

- Device Registry: this component needs to register sensors as entities in AGILE IDM whenever they are registered.
- Cloud Storage: this module developed in the scope of WP4 will rely on AGILE IDM to manage credential information to push or retrieve information from the cloud. Authentication credentials could either be an external Oatuh2 token, i.e. Google, retrieved by AGILE IDM during the login process, or it could also be an array of pre-configured credentials for sensors, users, or other entities.
- Apps and workflows: As per AGILE deliverable 3.1 (Charalampos and Erdeniz 2016) there are two different types of apps hosted on the AGILE gateway: simple IoT workflows, or full-fledged IoT applications. In both cases, they must interact with AGILE IDM to obtain a valid access token in order to access data from AGILE.

Integration for components accessible through the user interface will rely on a token obtained upon initial login by the user on the gateway. Figure 2 shows an overview of the following interactions required for authentication within the AGILE framework:

1. User visits the AGILE UI
2. The AGILE UI redirects the user to IDM
3. IDM provides a token to the UI (this includes several HTTP requests)
4. From this moment on (all steps 4' can be executed several times and without any particular order) the Desktop UI will communicate the token to other AGILE components requiring interaction with IDM, external clouds, or security protected AGILE APIs.

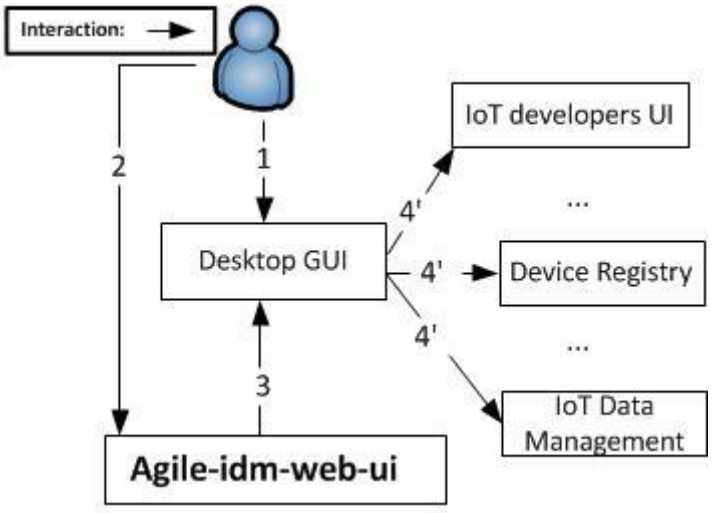

Figure 2 Integration with AGILE Framework

Technically speaking, the integration with the Desktop UI will be done through one of the Oauth2 authorization flows supported by AGILE IDM, for which several examples are available and are described in section 0. In addition, other components that require enforcement on certain API calls, such as access to data available through the Data management APIs will interact with an enforcement mechanism, yet to be defined and implemented in Task 5.2 that will require access to AGILE IDM.

1.3 Internal IDM Modules

AGILE IDM is comprised of several modules depicted in Figure 3. For visual clarity, the components drawn with a dashed line are available as a separate node js modules, and could be used separately from the agile-idm-web-ui for specific purposes such as entity storage. Although this is not required in AGILE, taking this approach helps to keep the code maintainable.

In reference to section 1.1, the AGILE IDM API mentioned is now split in two APIs. On the one hand, there is a REST Entity API for entities, attributes and and group management. On the other side there is an Oauth2 server API which follows the standard required to allow external applications to rely on AGILE IDM for user authentication.

Further, components that interact with an AGILE Oauth2 client through a REST API or with the user through a web browser are coloured with grey background; moreover, the arrows denote data flows in the direction of the arrow.

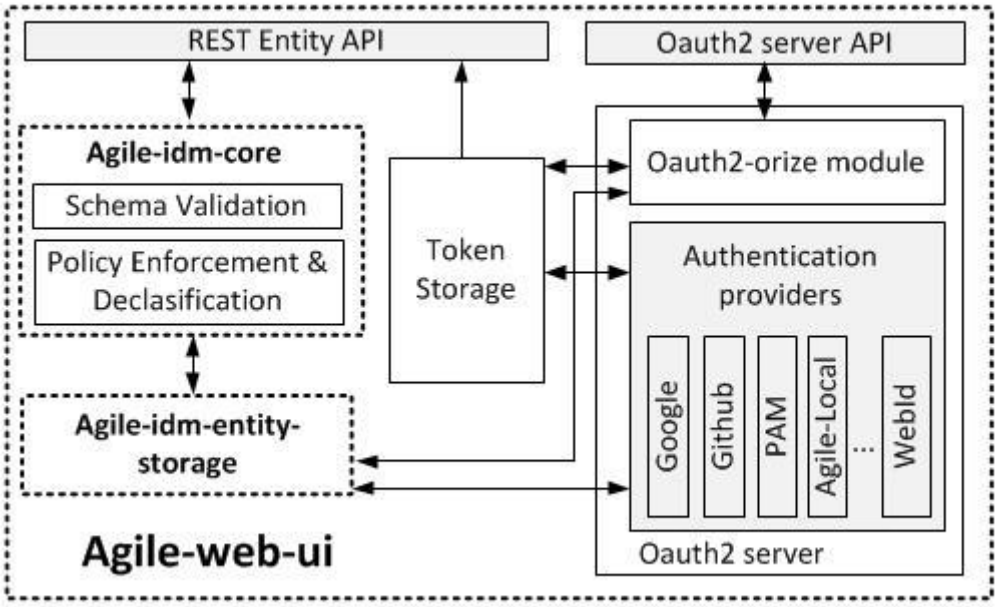

Figure 3 AGILE IDM Architecture and Components

For the description of basic responsibilities of each sub-module please refer to:

- **Agile-entity-storage**: provide an interface to a lightweight database to store entities and groups, while ensuring isolation and serialization of transactions. For more information, please see the github readme page: https://github.com/Agile-IoT/agile-idm-entity-storage

- **Token storage**: store access tokens received from external providers and generated by IDM.

- **Agile-idm-core**: provide an API to store and read entities, attributes, groups while ensuring proper policy enforcement to write and read for entity's attributes and preservation of entity formats. Agile-idm-core requires agile-idm-entity-storage to handle the persistence of entities. For more information, please see the github readme pace: https://github.com/Agile-IoT/agile-idm-core

- **Oauth2Server**: this component is taking care of OAuth2 server-side logic. In other words, this component allows applications to be oauth2 clients and rely on AGILE IDM to behave as an identity provider for them.

- **Oauth2-orize module**: this module generates access codes and exchanges them for tokens after the oauth2 client has been authenticated and has provided a proper authorization code. Also, the oauth2-orize module requires direct access to the agile-idm-entity storage because it needs to find clients by id and verify their secret before exchanging an authorization code for an access token.

- **Authentication providers**: this module loads a set of authentication providers that interact with users to authenticate them with different passport strategies. This component uses the token storage to save tokens obtained or generated by authentication providers. Also authentication providers use agile-idm-entity-storage to ensure that only users that are stored as entities of type user in the AGILE IDM database can log in. For more information check https://github.com/Agile-IoT/agile-idm-web-ui/blob/master/docs/authentication.md

- **REST Entity API**: this module is an express router encapsulating the entity management api. It requires an access token when it is called through HTTP and it relies on agile-idm-core to enforce proper policies to read and write attributes. It requires access to the token storage to verify the owner of a token provided during the REST call.

- **Oauth2 server API**: this is an express router taking care of exposing the proper HTTP calls to make AGILE IDM an Oauth2 provider.

1.4 Key Open Source Frameworks Used

Although AGILE IDM has more dependencies, the key open source components used are the following:

- **Express**: This framework is used as the basis for building the AGILE IDM web server. Along with additional express middleware were used to cope with body parsing, session handling, etc.

- **Passport**: Passport is an authentication framework with a live ecosystem of plugins contributed by the open source community. It has been used to implement the authentication mechanisms used by AGILE IDM.

- **Oauth2-orize**: This framework uses passport to facilitate the construction of an Oauth2 server.

- **LevelDB**: LevelDB provides a lightweight key value database used by AGILE IDM and its submodules to store tokens, entities and groups.

- **UPFROnt**: the ULock Policy FRamewOrk (UPFRont) is used to support usage locks to apply enforcement to write and read actions on attributes (available here: https://github.com/SEDARI/UPFROnt).

2 User Authentication

AGILE IDM behaves as an Identity Provider for applications (Oauth2 clients). In its current state AGILE IDM supports the authorization code, the client credentials and the implicit token grant flows from Oauth2, and it can use the following authentication mechanisms:

- **agile-local**: Local username and password
- **pam**: PAM (Pluggable authentication modules) UNIX authentication
- **webid**: WebID protocol
- **google**: Google Oauth2
- **github**: Github Oauth2

Every user has an id composed of the username and authentication type. During the login process, every strategy verifies that a user with the particular username and authentication type has been registered in order to allow the user to log in. This ensures that only allowed users log into the gateway; otherwise, anyone with a Github or Google account could log in.

Every strategy used by AGILE IDM takes care supports a particular identity provider. Each strategy is represented by an authentication type name. The authentication type name is comprised of the lowercase letters corresponding to the strategy as shown in bold characters in the aforementioned list (e.g., google, webid, github, pam, agile-local). Also, a file with the passport strategy calls name needs to be placed in the folder lib/auth/providers. Likewise a file with the same name needs to be located in the routes/providers folder. In the latter, the proper routes required for the authentication type are mounted in the authentication express router. For a detailed tutorial on how AGILE IDM can be extended to support additional providers, please see section 5.2

2.1 Authentication Types

The authentication type used to register the user should match (capital-case sensitive) the file name. For example, to allow the user alice in github to login into the gateway, it is necessary to create a user with the attributes user_name alice and auth_type github (given that there is a file called github.js in the provider folders). Now, we present a table with examples for each of the authentication types supported by AGILE IDM.

Description	**Strategy and Routes filename**	**auth_type property**	**user_name property**	**password**
User *alice* relying on *github*	github.js	github	alice	
Username bruce using gmail, google drive etc	google.js	google	bruce	

with email bruce@gmail.com.				
User *bob* relying on *local authentication* and using password "secret"	agile-local.js	agile-local	bob	secret
User root relying on UNIX authentication module (PAM)	pam.js	pam	root	
User eddy, who owns a WebId certificate generated by databox.me	webid.js	webid	https://eddy.databox.me/profile/card#me	

In the case of users for which the authentication strategy does not check the password, users *stored in the Agile IDM* do not require to have a password. Such examples include the Oauth2 providers, webid, or even the PAM mechanism. Or to put it differently, the only user registration that expects the user to have a password is the agile-local authentication provider. This is the case because this strategy verifies the username and password provided using the information stored as part of the user entity, i.e. user.

2.2 User Bootstrapping

Given that AGILE IDM works as an Oauth2 identity provider, an Oauth client and a valid user must be registered before any action can be performed with identity management. To this end, the scripts createClient and createUser available in the scripts folder of agile-idm-web-ui should be used; however, it must be noted that since these scripts are used for bootstrapping the system before there is any user, they do not check security policies. Therefore, they should only be used to create the initial clients and users, and afterwards they should generate new clients or users (when they are permitted by the identity model). More information on the steps to bootstrap the system is show as part of the examples setup in Chapter 5 or in github (https://github.com/Agile-IoT/agile-idm-oauth2-client-example)

2.3 Configuration

For a step by step guide to configure different Oauth2 providers please see:
https://github.com/Agile-IoT/agile-idm-web-ui/blob/master/docs/idps-configuration.md

3 Identity Model

This section covers available options to implement access control rules in systems, and a short discussion on why certain mechanisms are a better fit for the AGILE gateway. In addition, the entity entities, and by extension the attribute management, as well as groups supported by AGILE IDM are described.

3.1 Overview of Access Control Models

Given that the gateway could have several users administering different sensors, relying on simple access control mechanisms applied to every user of the gateway, such as Mandatory Access Control (MAC) rules, to enforce access to data would drastically reduce the possibility to handle data with proper granularity for different users interacting with the gateway.

A more flexible mechanism termed Discretionary Access Control (DAC) allows uses to override default policies and give or deny access to resources to other users based on a their identities. For instance, to implement DAC, it is common to see security policies defined in terms of resource ownership, user roles, or user groups, etc. This yields a role or group based access control mechanism respectively. Role based access control require a centralized enforcement role management to ensure that policies cannot be circumvented by adding entities changing entities' roles.

Although role based access control are useful to handle system-wide policies in which certain groups or roles are allowed to perform certain operations, it cannot efficiently cover use cases in which users need allow other users to access their data without requiring a particular role to be defined by a system administrator. The latter is imperative for AGILE, given that use cases will require users, i.e. data owner in the quantified-self scenario, to give access to another user, i.e. family doctor, without necessarily requiring the administrator of the gateway to create new user roles.

On the other hand, policies defined in terms of groups may be enough in cases when access policies are fairly simple; further, by allowing users to create their own groups of entities to define policies based on such groups solves the issue of having a centralized role management. However, these mechanisms tend to become more and more complex to configure and understand when the number of users and groups increases.

Luckily, attribute based access control (ABAC) is a more generic approach allowing policies to be specified in terms of attributes that could belong to a subject, an object, an action performed by the subject in an object, or the environment (Hu, et al. 2013). Fortunately, mechanisms such as mandatory access control (MAC) or role based access control (RBAC) can be easily implemented in ABAC.

In AGILE we have defined an attribute based identity management. This attribute management system is comprised of a persistent entity database complemented with components which allow the enforcement of a set of configurable policies on such entity's attributes. These two elements together can be used to enforce access to attributes to ensure that only users allowed changing or

seeing attributes can update them or see them. Furthermore, we have also included the possibility of users to create groups of their own. This allows them (once proper enforcement mechanisms developed in Task 5.2 are implemented) to define policies that reference groups they handle. Alternatively, for more security savvy users, or more complex setups data access policies can be based on entities' attributes.

3.2 Entities

In AGILE IDM, there are two kinds of objects in the database: entities and groups. On the one hand, entities are designed to represent principals in the system. For AGILE, we have considered that users, sensors, and oauth2 clients will be required to represent the different principals so far. Notwithstanding, it is likely that additional entities such as workflows or applications also need to be represented in AGILE IDM as the use cases evolve. In any case, the definition of additional entities can be performed very quickly by adding entries in the configuration file and restarting AGILE IDM.

The definition of entities in the system can be performed by defining entities schemas and policies on particular attributes in a configuration file. The reason to support this is twofold. On the one hand, we want to avoid introducing too many software changes as identity management and other components evolve in Task 5.4 (where there is a loopback feedback to improve aspects of security components, such as identity management) . On the other hand, we want to give the possibility to developers to leverage AGILE IDM for different projects in which they can define their own entities and policies on how attributes can be read or written.

The schema specifications do not require code modifications as long as entities are defined by the jsonschema package for Node. Js (this schema is mostly compatible with the IETF json schema draft v4 according to their documentation and supports nested entity definitions, i.e. objects inside objects). An entity can be specified using JSON schema in the validation-schema property of the AGILE IDM configuration as follows:

```
{
    "id": "/user",
     "type": "object",
     "properties": {
     "user_name": {"type": "string"},
     "auth_type": {"type": "string"},
     "password": { "type": "string"},
     "role":{"type":"string"}
   },
  "required": ["user_name",  "auth_type","role"]
 }
```

In this specification, a user must have the properties “user_name”, “auth_type” and “role”. The username references the username or id according to the identity provider, while the “auth_type” reflects the identity provider. For example if alice@gmail.com would be using agile her “user_name” attribute would be alice and ther “auth_type” would be google, according to the identity provider introduced in Chapter 2. Also, the “role” attribute is going to be used to determine whether the user created can perform particular operations on the gateway. This will be also a configurable parameter explained in section 3.4

3.3 Groups

To allow users to specify groups of entities freely, we have implemented a mechanism to allow users to create their own groups, and add any entity to groups *they own*. We expect that for scenarios that are simple enough and users that are not so experienced with policy definitions, this would prove to be a useful tool to control access and usage of their data. However, we also expect that users that have more complex set-ups leverage the attribute based approach to make the management of entities and policies.

3.4 Attribute Management

Given that attributes are a key component to define security policies, it is imperative to control which users can write and read to them. This section covers how attribute authorities are handled, as well as how it is ensured that only allowed entities can read particular attribute values.

3.4.1 Attribute Authorities

The main paradigm shift between previous models and the attribute based model is that certain users are (selected by an attribute value, role, group, id etc.) are allowed to define attributes for the entities. In particular, de user allowed to set an attribute value is the *attribute’s authority* (Hu, et al. 2013).

A typical example to clarify the meaning of an attribute authority is to consider a company in which a new employee has been registered, and values such as his home address and security clearance level are part of his profile. In such scenario, the human resources department should be allowed to set the employee’s home address or his job description, but should not be entitled to assign the employee’s clearance level regarding data access. The latter attribute should commonly be set by someone in the IT-security department. In this case, the attribute authority for the user’s address is the human resources department, and the attribute authority for the IT access is the IT-security department. Furthermore, the attribute based mechanism can be configured to rely on a user’s attribute, e.g. department, to be used when deciding whether or not a particular user can change an attribute, such as address or clearance level.

Attribute authorities are flexibly defined in AGILE IDM. To achieve this configuration file states which policies must be applied when an entity tries to change an attribute. This is translated to enforcing a write policy on the attribute, as specified in section 3.4.3.

3.4.2 Attribute Visibility

AGILE IDM can also be configured to handle particular attributes which can only be read by a set of entities. Common cases of such attributes are credentials, passwords or any other privacy sensitive information from a user.

Attribute visibility is also configurable in AGILE IDM, and entities are automatically declassified, i.e. parts of the entity which are not allowed to be read are removed, before the entity is returned. Thanks to this property, AGILE IDM can manage credentials for sensors or other entities as attributes. This proves particularly useful to push data to clouds that require particular credentials for users or sensors, or perform additional operations in the gateway. More on this is covered in section 4.1.

Ensuring proper declassification can be done by removing attributes which are not allowed by a read policy as described in section 3.4.3.

3.4.3 Attribute Policy Enforcement

We leverage an on-going research, in the scope of the FORSEC project (Pernul, Schryen and Schillinger 2016), effort on policy definition and enforcement initiated originally under the European funded COMPOSE project (Schreckling, Parra and Gottschlich 2015). Specifically, we implement the attribute visibility and attribute authorities based Usage Locks which are currently under development as part of UPFROnt (https://github.com/SEDARI/UPFROnt) .

A policy can be expressed as an array of objects. Each object must have either a source or a target property. On the one hand, sources reflect conditions relevant for entities which are sources of data that flows to the referenced attribute, i.e. write. On the other hand, target policies reflect that they are applied to entities that will receive information from the attribute, i.e. read.

Each policy should have at least one source and one target, although the lack of one of the types just results in denied access. However, when there is more than one source, policy evaluation will allow the action if the one of the sources is allowed, i.e. disjunction operation. The same applies for targets.

Additionally, each source or target can contain a set of usage and parametrized locks. In order for the policy to allow an action, every lock must be opened, i.e. the conjunction of all the locks inside a source or a target must evaluate to true. By default, every lock is closed at the beginning of the execution, and during the policy evaluation locks are opened when the entities fulfill the given conditions; also, when no locks are defined it is considered that all locks are open, i.e. action is allowed. Current lock implementations available from UPFROnt are:

- **isOwner**: this lock does not need any parameters and verifies that the user executing the action is the owner of the entity he is acting upon. In particular, to simplify the ownership relationship, we have defined that users own themselves.

- **attrEq**: this lock evaluates that a particular argument has a given value; therefore, it requires two arguments, namely the attribute name and the attribute value.

Identity management takes a default policy (not shown for simplicity, although it is can be seen in the top_level_policy field of the configuration file for AGILE IDM) which allows administrators to create users, and any user to see the user's attributes by default. However, this is overridden by more (or less) restrictive policies. Such policies are placed in the policy administration point by identity management, and are configured by specifying the attribute's name under the "attribute_level_policies" property of the configuration file.

Policy Example

For clarity, we explain now the definition of the user's attribute policies as an example:

```
"user": {
  "password": [
    {
      target: { type: "user" },
      locks: [{ lock: "isOwner" }]
    },
    {   source: { type: "user" },
        locks: [{
            lock: "isOwner"
        }]
    },
    {
     source: { type: "user" },
     locks: [{
                    lock: "attrEq", args: ["role", "admin"]
          }]
    }
  ],
  "role": [
    {
      target: { type: "any" }
    },
    {
```

```
      source: { type: "user" },
      locks: [{ lock: "attrEq",  args:
                          ["role", "admin"]
                        }
      ]
     }
  ]
}
```

Breakdown of Policy Example

First of all, the previously shown configuration shows that identity management overrides the default policy for the password and the role attributes with more a restrictive policy.

First of all the **password** policy has the following parts:

Policy Section	Semantics
target:{type:"user"} **locks**:[{**lock**:"isOwner" }]	The password can be read by the user himself. This is true only for the user for which this password is defined because every user owns himself, i.e. the user's id will match the owner id always.
source:{type:"user"} **locks**: [{**lock**:"isOwner"}]	The password can be updated by the user himself, applying the same reasoning as the previous lock.
source:{type: "user"} **locks**: [{**lock**:"attrEq", **args**:["role", "admin"] }]	The password can be updated by users whose attribute called "role" equals "admin". This allows administrators to reset passwords. Note however, that administrators are **not allowed** to read the password. If it would be desirable that administrators read the user's password, a similar rule must be added, but the source tag should be exchanged for a target.

Overall, the previous policy allows users to read their own passwords. On the other hand, there are two options to allow a user to update a password, i.e. two sources. Either the user updating the password is the user himself, or the user is an administrator.

Now, given that we are basing policy decisions on the role attribute, it is required to ensure that only certain users can set the attribute for users. This is covered by the **role** policy, which has the following parts:

Policy Section	**Semantics**
target:{type: "any"}	This section specifies that any entity can read the role of the user. This is an example in which the lack of lock definition implies that all locks are open, i.e. allowed action.
source:{ type: "user"} **locks**: [{**lock**:"attrEq", **args**:["role", "admin"] }]	This section specifies that only users of who have an attribute role "admin" can update the user's role. Please note, that this policy disallows even the user himself to change his role.

Overall, the role policy allows every entity to read a user's role. However, the only case in which a user can update a role on any user is when this user has the role attribute as "admin". The first user with attribute role as "admin" needs to be generated before identity management is started by using specific scripts to bootstrap the system.

3.5 Privacy Preserving Tokens

In addition to AGILE IDM, an initial effort towards an implementation for one time tokens for constrained devices as described in (Cuellar, Suppan and Poehls 2015) has been started. This token allows devices to generate one time tokens to prove their identity towards AGILE by executing hash functions and bit-shifts. We expect that computation requirements are kept low due to the simple token generation.

For the development of a small code base in C, which can be executed on resource-constrained RIOT devices for AGILE, an initial Java-based implementation developed for COMPOSE (Schreckling, Parra and Gottschlich 2015) has been used as a guideline to test the basic functionality of the library (by translating the existing unit tests from Java to C).

Currently, the token generation to prove the sensor's identity has been implemented, but additional effort is required to also support some encryption mechanism and to integrate this into AGILE. This extension will be performed under Task 5.3.

4 Functionality Overview

This section covers a general description of AGILE IDM's functionality; afterwards, functionalities are related to specific project requirements and upcoming integrations.

4.1 Functionality

The main functionalities of AGILE IDM are the following:

- **Oauth2 provider**: AGILE IDM implements the authorization code grant of Oauth2. In turn, AGILE IDM can still use external identity providers such as google, github, or even the local operating system to authenticate users, depending on its configuration. This allows the AGILE framework and potentially other applications to rely on it to handle the authentication of their users.

- **User registration**: AGILE IDM also handles the user registration and management. In this way only certain users are allowed to login in the gateway. This is of utmost importance when AGILE IDM relies on external identity providers, because otherwise any user with a valid account in the external identity provider would be able to log in.

- **Entity registration**: AGILE IDM is used to register entities, such as sensors, oauth2 clients, workflows, etc. This forms the basis for policy enforcement.

- **Entities' attributes management**: AGILE IDM allows users to manage their entity's attributes. It implements write and read policies on the data to provide attribute assurance, which then allows for the implementation of a wide range of access control mechanisms, such as role based access control among others.

- **Entity lookup**: AGILE IDM allows users to lookup entities. To this end, users can provide a set of constraints specifying attribute name and value.

- **Entity attribute declassification**: AGILE IDM also declassifies attributes that are not readable by users that query IDM. For example, there may be certain attributes that can only be read by the entity owner, but not by the rest of the users.

- **Credential Management**: Thanks to the entity declassification functionality of AGILE IDM, it could be configured to allow entities to store credentials (that are used to connect to external clouds or systems for example). Provided that the right policies are configured, IDM would only return this information to the entities allowed to read this information, e.g. the owner of the entity.

- **Group Management**: To simplify the policy definition for developers and other users, they can define groups and add entities to them. This can provide an easy way to handle entities and to define security policies for them by referencing the group directly.

4.2 Project Requirements

AGILE IDM supports a range of requirements already specified in previous deliverables already written within the scope of other technical Work Packages. When there has been a previous deliverable mentioning different sets of requirements relevant for IDM, we take sections of other deliverables (copied literally in italic font). Immediately after each requirement we highlight how the current implementation, and the previously described functionalities, addresses each one of them.

4.2.1 Device, Data Management & Developers Environment Development (WP3)

The following requirements (relevant for AGILE IDM) were present in Deliverable 3.1 (Charalampos and Erdeniz 2016):

UMI-UGM-01 *- AGILE Gateway Users: the AGILE gateway will support the creation and deletion of users having different roles in the access to Gateway capabilities. A default user called "agile" will be predefined at installation time, having the capability for administering the default set of gateway capabilities*

This is enabled by the **user registration** and the **group management** functionalities explained earlier. Creation and deletion is supported through the REST API, the default user can be created with the scripts used to bootstrap IDM referenced in section 2.2.

UMI-UGM-02 *– AGILE Gateway User Groups: users can be organized into "groups" that will allow to share access to a common set of capabilities offered by the Gateway*

This is implemented through the **group management**. Group management is implemented in such a way that any user can create groups, and they can always add or remove entities to groups that they own. Alternatively, this can be implemented by setting an attribute value and using it for policy decisions too.

UMI-UGM-03 *– AGILE Gateway User Roles: users and groups will define the configuration of capabilities offered by the gateway to which users/group can have access."*

This is implemented by using the **entity registration** and the **group management**. In the current setup, we have configured to have two roles for basic access control rules, but this can/and will be extended from now in the scope of tasks 5.2 and 5.3.

4.2.2 Public and Private Cloud Integration (WP4)

When users want to push data or applications to external clouds, different authentication mechanisms are used. For example, if a user wants to store data in Google Drive or Things Speak he needs an access token. However, depending on the cloud, either an Oauth2 token or a set of static credentials such as an API KEY may be needed. Now we cover the two possible scenarios and explain how IDM supports the cloud drivers in WP4.

Oauth2 Credentials

In general, the interactions to integrate IDM with cloud drivers and the GUI Desktop Framework are shown in Figure 1, and can have the following steps:

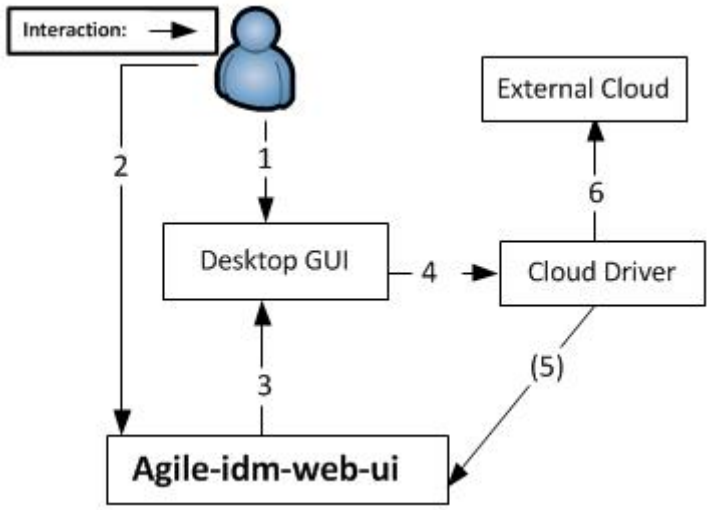

Figure 4 Integration with Cloud Drivers

1. User visits the AGILE UI
2. The AGILE UI redirects the user to IDM
3. IDM provides a token to the UI (this includes several HTTP requests)
4. The user executes a call the cloud driver through some component in the interface.
5. This step is optional: here, the cloud driver could obtain a token for a particular Oatuh2 provider in case the token provided in step 4 is for a different identity provider (this step requires several network requests).
6. The cloud driver pushes information to the cloud

Step 5 can be normally skipped provided that the user is currently logged in with the proper Oauth2 provider through AGILEs GUI Desktop framework. If this is not the case, the cloud driver would receive the access token used for AGILE generated by a different identity provider. Let us assume that the cloud driver receives a request from a user that used a local authentication user (no external Oauth2 provider), and wants to push data an external cloud, i.e. Google Drive. In this case, the driver can interact with IDM to get the Google token independently to perform this action. In this case, the cloud driver would register with AGILE as an Oauth2 client, in order to obtain an access token from the required Oauth2 identity provider, i.e. Google in this case,

Static Credentials

In certain cases, external clouds may require static credentials for the authentication process. These credentials normally do not expire, like oauth2 tokens commonly do, and need to be obtained by the user previously. AGILE IDM also supports such scenario by allowing users to

store credentials for them, their workflows, sensors (or other entities). Credentials would be stored in a private attribute that is only visible by the owner of the entity (or the user itself in case they are stored in an attribute for the user).

Provided that the user has stored his static credentials for a cloud API (let us assume he has stored his Think Speak token in AGILE IDM already), the flow to support authentication with the external cloud depicted in follow the same steps described in Figure 4, with one difference in step 5.

In step 5, the cloud driver would query AGILE IDM, and instead of requesting it to authenticate the user, it would send only one request to retrieve the user information and providing the token received in step 4. From the identity management point of view, the user is querying IDM to see his own attributes; therefore, the cloud driver will get not only the public but also the private attributes of the user (the ones he can only see himself). Among those attributes, there will be a credential array in which the user has stored his static keys, e.g. Things Speak key. At this point the cloud driver can parse the array and use the credentials to interact with the external cloud (Things Speak in this case).

4.2.3 Pilots (WP8)

The pilot design and requirements analysis and specification (Menychtas 2016) contains the following requirements relevant for identity management:

***SEC_1 User Authentication**: The gateway must provide the required functionality for authenticating the users that will login in the Web Interface.*

This is covered by the login screen by the **Oauth2 provider** functionality, i.e. the login site of IDM.

***SEC_2 Identity Provider**: The gateway must provide means for applications, developed through the graphical interface and running in the gateway, to leverage the AGILE Identity Management in order to verify which user has been logged in, e.g. Oauth2 integration or similar.*

This requires the **entity registration** to register applications or workflows as Oauth2 clients. Once this step takes place, applications can use the implicit, client credentials, or authorization code grant flows defined by using IDM as an **oauth2 provider** to obtain tokens from IDM. The oauth2 authorization code example described in section 5.1.3 also maps the token to a web session. This is an easy to understand starting point for developers using AGILE IDM too.

***SEC_3 User Management:** The gateway must provide user account management functionality so as the users can update their profile and credentials, including the WebID.*

***SEC_6 Encryption Management:** The gateway must allow users to associate encryption keys with their identities to enable secure data sharing with external systems.*

SEC_3 and SEC_6 are covered the read and write policies to entities provided by **entity management** (specifically reflected in the **credential management** functionality description). In short, an attribute, e.g. credentials, can be configured in such a way that only the owner of the entity can see it. For example, if a user would want to store his/her keys or any other kind of credentials, IDM can be configured to hold a "credentials" attribute that is only visible for him/her. In this way when the user through an AGILE component, or from his own app queries IDM, queries the REST Entities API to find information his own attributes he will see the credentials attribute. On the other hand, when other users query IDM they will not be able to see this information since entities are declassified (as described in section 3.4.2) before they are returned to the user querying IDM, depending on his ability so read attributes.

In addition to the aforementioned requirements, AGILE IDM supports webid, which is a protocol used by Jolocom's application that will be leveraged in one of the project's use cases. Even though this was not pointed out as a specific requirement in previous deliverables, this provides additional support and simplifies the integration with the use case application for data sharing developed by Jolocom

5 Examples

We have produced a set of examples allowing developers to experiment, interact and have starting points for software development of components interacting with AGILE IDM. We have produced two kind of examples: on the one hand, applications interacting with IDM through OAuth2, or generating and reading groups, entities and users are available on the developer interact with AGILE IDM. These examples are available in an AGILE IDM examples repository (https://github.com/Agile-IoT/agile-idm-examples/) On the other hand, we have created an example to extend IDM to include a "new" authentication provider in a separate repository (https://github.com/Agile-IoT/agile-idm-additional-provider-example) . This tutorial could be helpful for developers extending IDM in general, but in particular, this knowledge is required to extend IDM to support clouds in Work Package 4, or may be useful to allow partners developing use cases to create their own strategies to interact with proprietary clouds or additional IoT services which are not supported by default in AGILE. Throughout this section, pictures depict the actual HTTP(s) requests between IDM and the example applications, which gives more detail than previous pictures that just described interactions in a more abstract way, e.g. Figure 2 Integration with AGILE Framework.

5.1 AGILE IDM client examples

AGILE IDM is an Oauth2 server that implements the following grant flows:

- **client credentials**: this grant flow allows an application (without user interaction or web interface) to obtain an access token by providing the client id and its credentials.
- **implicit**: a web application obtains the token after the user has logged in. In this case, the browser will obtain the token directly. In this case, the server must neither execute server side requests to exchange codes for tokens nor to execute server side code to authenticate itself. Although this approach can be easier to implement, there is an additional risk since the browser obtains the token.
- **authorization code**: a web application obtains an authorization code after the user has logged in, and afterwards it executes calls (directly from server side) to exchange the authorization for an access token. In this flow, the browser does not access the access token, but an access code which is useless unless the proper authentication steps required to authenticate the server are performed. This makes the authorization code the most secure approach.

For readability purposes, all examples (for clients interacting with IDM) are organized in folders within the AGILE IDM examples repository. Each folder includes several examples to illustrate applications using the different kind of authentication grants. Further, in the case of the authorization code grant, a complete application generating requests to AGILE IDM to register entities, groups, update attributes, among other actions, is also provided.

5.1.1 Client Credentials IDM Client Example

This example shows how to create a client and authenticate using the client credentials Oauth2 flow, as per RFC 749, from an application. The main advantage offered by the client credentials flow is, that an application without a web-based user interface, can still get tokens and provide them to AGILE IDM. Naturally, once the client application obtains the token, it can be used with the AGILE framework to perform actions. However, this method does not take advantage of the external identity providers (such as google and github, webid, etc) included in AGILE IDM.

Set up

First of all, we need to create the first user with IDM, if it not already created: For this, execute the following command in the scripts folder of agile-idm-web-ui

```
node createUser.js --username=bob --password=secret  --auth=agile-local
```

Afterwards an Oauth2 client must be created, make sure to provide an existing user to the createClient script.

```
node createClient.js --client=MyAgileClient2 --name="My first example as IDM client" --secret
="Ultrasecretstuff" --owner=bob --auth_type=agile-local --uri=http://localhost:3002/auth/examp
le/callback
```

Overview

Assuming that an oauth2 client has been registered in AGILE IDM, the flow to obtain an access token for the client using the client credentials flow is depicted in the following picture.

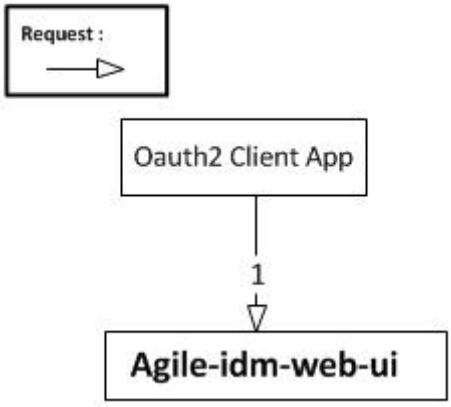

Figure 5 Interaction between Client Credentials application and AGILE IDM

1. the client calls AGILE IDM, and provides the credentials using the HTTP Basic authentication protocol, and specifying the client credential grant type. Afterwards AGILE IDM delivers an access token to the application, in case it has provided the proper credentials.

From this point on, the application can use this token to interact with IDM, or with any other AGILE component that has been integrated with AGILE IDM.

Authenticating with the Client Application

We have included a basic command line application that authenticates the client, and afterwards queries the client information as well as the user information.
In case of the client credentials flow the owner of the client is represented as the user authenticated.

The sample application can be executed from a terminal to authenticate the client we just created like this:

node authenticateClient.js --client MyAgileClient2 --secret Ultrasecretstuff

It is also possible to execute the authentication script providing additional arguments such as protocol, port, and host like this:

node authenticateClient.js --client MyAgileClient2 --secret Ultrasecretstuff --host 127. 0.0.1 --protocol http -- port 3000

CURL

Alternatively one could authenticate a client using tools to generate http requests such as curl.

curl -X POST -u MyAgileClient2:Ultrasecretstuff -d grant_type=client_credentials http ://localhost:3000/oauth2/token

The previous command line call would use the proper HTTP Basic authentication mechanism to authenticate the client to AGILE IDM (assuming it is running in localhost in port 3000). The expected result looks like the following:

```
{
        "access_token":"1A9He ... rWXvrc9nqSdlj1vsEQE3INQyR0bRODEl",
        "token_type":"Bearer"
}
```

5.1.2 Implicit Authentication (Token flow) IDM Client Example

The implicit authentication enables applications that are not capable of generating server side requests to authenticate the client, to still rely on an identity provider.
A typical example of this, is a case in which an application that runs in a browser and executes JavaScript code. In this simplified authentication grant flow, the web browser obtains a token directly from IDM through a redirection.

The key consideration when this authentication grant is used is that the browser obtains access to the token directly. On the contrary, in the authorization code flow, the browser only obtains an authorization code, which is normally useless unless the browser managed to obtain the secret of the client application (which should never be shared with the browser).

Example Set Up

First of all, we need to create the first user with IDM, if it not already created. To achieve this, execute the following command in the scripts folder of agile-idm-web-ui:

node createUser.js --username=bob --password=secret --auth=agile-local

Now that a user exists, an oauth2 client must be registered. To achieve this, execute the following command line from the agile-idm-web-ui script folder. If any of the parameters of the command line execution are changed, the configuration file for the example available in the "conf" folder of the oauth2 client example should be updated:

node createClient.js --client=MyAgileClient --name="My first example as IDM client" --secret="Ultrasecretstuff" --owner=bob --auth_type=agile-local --uri=http://localhost:3010/

note: that this callback is different than the authorization code. It returns directly to a URL that servers HTML to the browser. In the case of an authorization code flow, this URL needs to redirect to an endpoint in the server that is capable of exchanging the authorization code for a valid access token after presenting the client credentials. In this case, the browser will get the access token directly after following the redirection to http://localhost:3010/index.html, therefore removing the burden on the server side to exchange the token. However, the browser (and the client side JavaScript code) needs to be trusted, otherwise this mechanism should not be used.

Afterwards, run identity management by executing this in the root of the agile-idm-web-ui folder:

node app.js

And subsequently run the oauht2 client example by executing this in the root of the oauth2 client example project:

node index.js

Then visit http://localhost:3010/

Overview

Assuming that an oauth2 client has been registered in AGILE IDM, the flow to obtain an access token for any AGILE user using the implicit grant is depicted in the following picture.

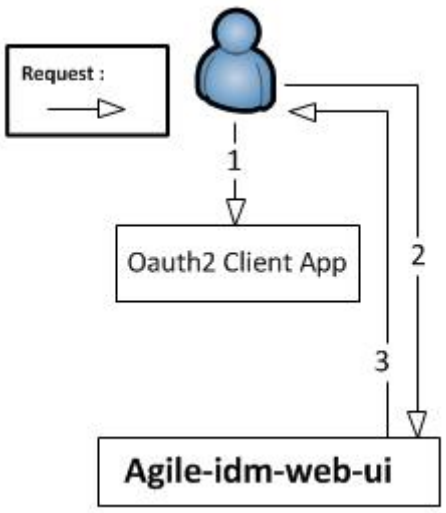

Figure 6 Interaction with Implicit Authorization AGILE IDM Client

From step 1 to 3, the authentication between IDM and the Oauth2 client occurs; however, it must be noted that in case AGILE IDM relies on another identity provider, such as google, to authenticate the user, additional steps will take place between step 2 and 3. In each step the following actions take place:

1. the user opens the Oauth2 client application, but he/she is not authenticated yet.
2. the Oauth2 client App redirects the user to AGILE IDM presenting its client id and a redirect URL that will be called by IDM on successful authentication of the user (if this URL matches the registration of the oauth client). Subsequently, the user authenticates with AGILE IDM using any of the authentication providers available.
3. On successful user authentication, IDM redirects the user with an access token and token type including the token and the token type as an URL fragment according to RFC 6749. At this point the browser can parse the URI and obtain the token and the token type to use it afterwards.

From this point on, the browser can use this token to interact with IDM, or with any other AGILE component that has been integrated with AGILE IDM.

5.1.3 Authorization Code IDM Client Example

The Authorization code is the most complex example available, and requires a full-fledged web application. For this reason, although it is referenced in the AGILE Examples repository, it has been extracted in a separate location (https://github.com/Agile-IoT/agile-idm-oauth2-client-example) which contains the following branches to showcase different aspects of AGILE IDM:

- **client**: a minimalistic application that uses AGILE IDM as its identity provider, and displays user information (user information, and token).

- **api-client**: an extended version of the client application, but on top it also offers some functionality to register entities on identity management through a web interface, and executes the proper REST API calls to AGILE IDM.

Example Set up

This example assumes a certain kind of entity set (defined in the configuration of agile-idm-web-ui). So, please copy the file in agile-ui-conf in the conf directory in agile-idm-web-ui for consistency.

First of all, we need to create the first user with IDM, if it not already created. To achieve this, execute the following command in the scripts folder of agile-idm-web-ui:

node createUser.js --username=bob --password=secret --auth=agile-local

Now that a user exists, an oauth2 client must be registered. To achieve this, execute the following command line from the agile-idm-web-ui script folder. If any of the parameters of the command line execution are changed, the configuration file for the example available in the "conf" folder of the oauth2 client example should be updated:

node createClient.js --client=MyAgileClient2 --name="My first example as IDM client" --secret="Ultrasecretstuff" --owner=bob --auth_type=agile-local --uri=http://localhost :3002/auth/example/callback

Afterwards, run identity management by executing this in the root of the agile-idm-web-ui folder:

node app.js

And subsequently run the oauht2 client example by executing this in the root of the oauth2 client example project:

node index.js

Example Overview

Assuming that an oauth2 client has been registered in AGILE IDM, the flow to obtain an access token for any AGILE user using the authorization code grant flow is depicted in the following picture.

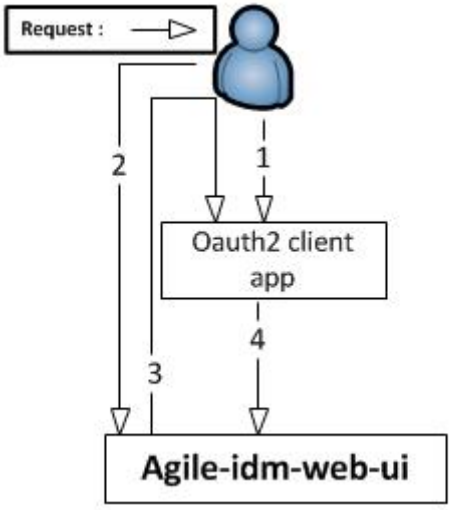

Figure 7 Interaction of Authorization Code AGILE IDM Client

From step 1 to 4 in Figure 1, the authentication between IDM and the Oauth2 client occurs; however, it must be noted that in case AGILE IDM relies on another identity provider, such as google, to authenticate the user, additional steps will take place between step 2 and 3. In each step the following actions take place:

1. The user opens the Oauth2 client application, but he/she is not authenticated yet.
2. The Oauth2 client App redirects the user to AGILE IDM presenting its client id and a redirect URL that will be called by IDM on successful authentication of the user (if this URL matches the registration of the Oauth2 client). Subsequently, the user authenticates with AGILE IDM using any of the authentication providers available.
3. On successful user authentication, IDM redirects the user with an authorization code (valid only for this client) to the client callback endpoint.
4. Once the Oauth2 client application has received the authorization code, it calls IDM providing the authorization token along with its client id and client secret, in order to exchange the authorization code for a valid access token for this user. Once IDM returns a token (if id and secret are valid), it will delete the authorization code afterwards. From this point on, the application can use this token to interact with IDM, or with any other AGILE component that has been integrated with AGILE IDM.

In the case of the more complex example (api-client branch), there will be subsequent requests from the Oauth2 client app to AGILE IDM, which contain and access token, using a bearer authorization token in HTTP, to store and read entities and groups from AGILE IDM when the user interacts with the graphical interface of the example.

Project Source Code Structure

The code for the oauth2 client example is structured in the following folders:

- **certs**: self-signed certificates used for TLS connection (for security reasons proper certificates should be used)

- **conf**: configuration of the current application. This includes the following configurations:

  - **oauth2**: contains the client configuration, which includes callback url to receive the authorization code from AGILE IDM, the authentication endpoints, client id and client secrets. All these values are already consistent with the example setup step by step guide, i.e. the proper client was generated in the set up phase.

  - **site**: ports to run in http and in https as well as certificates used for TLS (by default pointing to the ones available in the certs folder)

  - **idm**: this configuration property is needed in the api-client example, to build the URL to the REST Entity API of AGILE IDM. In the case of the simplest client which just uses IDM as an identity provider this configuration is not present.

  - **db**: This folder contains a minimalistic implementation of an in-memory database for tokens and users. These tokens and users are generated by the passport authentication strategy user (oauth2 client). On the one hand, having this in-memory database makes it easy to run the example due to the lack of additional installation of a database component. On the other hand, this may not be suitable for production environments because users are logged out whenever this example client is restarted.

- **passport**: this folder includes two building blocks of passport.

  - **the user serializer** is required to keep the user id in the session and provides a mapping to the user database. This component is required by passport to map users (who have been logged in) to their respective sessions. In this particular example, the session is ultimately implemented as a cookie between the web browser and the client.

  - **the passport strategy** is used to authenticate users with AGILE IDM, by relying on it to behave as an Oauth2 provider, and it also overrides the exchange of an access token by the user's information. The latter has been implemented in the *profileFromIDM* function.

- **routes**: this folder includes the proper express code to mount the routes for the oauth2 authorization code flow, the routes to retrieve user information. And optionally, in the case of the more complex example that also consumes the REST Entity API of AGILE IDM, it includes the routes required to receive requests from the browser. In turn these files (called groups, entities, and users depending on

their particular responsibilities) take care of executing the REST calls to AGILE IDM, and to render the proper views to the user in HTML form.

- **views**: includes the ejs views, which comprise a basic html template to represent the information to the user in HTML.

- **web-ui-conf**: this folder contains a configuration file that should be placed in AGILE IDM.

Also the root folder contains the following files:

- index.js: **main** file to be run, to try the example.

- package.json: used to specify project name, url and dependencies

- Gruntfile.js: file used to format and verify code structure. Also it checks for jshint errors in JavaScript

- .jsbeautify and .jshintrc: configurations for beautify and jshint used by the grunt file

To print debugging information of IDM, please have a look at the agile-idm-web-ui here https://github.com/Agile-IoT/agile-idm-web-ui/blob/master/README.md

Debug mode

If you define the following variable (to be 1) this module will print debugging information to standard output, and in case of exceptions, it will print the stack trace in the browser.

export DEBUG_IDM_WEB=1

If no variable is set or if any other value different than 1 is set, this component runs in quiet mode.

To debug the agile-idm-core or the agile-idm-storage components that are within agile-idm-web-ui please set the environment variables DEBUG_IDM_CORE or DEBUG_IDM_STORAGE to 1 respectively. These two components will log to standard output debugging information.

5.2 Integrate a new Authentication Mechanism

Now we cover an example to extend an installation of agile-idm-web-ui to support an additional (dummy) identity provider, also available in github (https://github.com/Agile-IoT/agile-idm-additional-provider-example).

The dummy provider uses a local passport strategy. This strategy allows the provider to obtain username and password from the agile user interface.
The strategy allows only users registered with agile which have username "alice" and password "secret". This strategy must not be used in production environments, but it comprises a minimalistic example to integrate a new strategy with AGILE IDM.

5.3 Set up

To install the new strategy just execute the following commands in a terminal to checkout agile-idm-web-ui v 1.0.1:

```
git clone https://github.com/Agile-IoT/agile-idm-web-ui
cd agile-idm-web-ui
git checkout v1.0.1
```

Then copy the strategy and the routes to agile-idm-web-ui and install it (still from the agile-idm-web-ui directory):

```
cp ../routes/my-demo.js  ./routes/providers/
cp ../strategies/my-demo.js ./lib/auth/providers/
cp ../agile-ui-conf.js  ./conf/
npm install
```

Then, set-up an admin user, for example using the agile-local strategy. Also create a client (to execute the oauth2 example available here:https://github.com/Agile-IoT/agile-idm-oauth2-client-example):

```
cd scripts
node createUser.js --username=bob --password=secret   --auth=agile-local
node createClient.js --client=MyAgileClient2 --name="My first example as IDM client"
--secret="Ultrasecretstuff" --owner=bob --auth_type=agile-local --uri=http://localhost
:3002/auth/example/callback
```

Now create the user expected by this strategy:

```
node createUser.js --username=alice   --auth=my-demo
```

Note that the client has a new authentication type called *my_demo*. This matches the authentication type that is registered in this example.

Then, execute IDM:

```
cd ../
node app.js
```

Then clone, install and start the demo app according to the agile-idm-oauth2-example readme.

If you want to set-up IDM in just one step, execute the "build-idm.sh" script, which will set-up AGILE IDM and start it.

5.4 Overview

The code present in the repository just contains the strategy file, and the routes file. In general for new strategies, the filenames
(without the .js suffix) needs to match the authentication type in the code. This allows agile-idm-web-ui to load strategies consistently.

5.4.1 Routes

The routes file is in charge of setting the location of the proper routes needed for the authentication mechanism, and specify the authentication strategy that should be used by passport. This needs to be freely defined by the programmer because although each authentication strategy can require a different amound of endpoints to authenticate the user. All the routes provided by the router instanciated in the routes file for the strategy are mounted in the /auth/ path for the agile-idm-web-ui server.

Here is the explanation of some extracts of the routes file:

Initial Login Page

```
router.route('/my-demo').post(     passport.authenticate('my-demo' , {
successReturnToOrRedirect: '/',      failureRedirect: '/login'     })   );
```

This code specifies that when the user posts credentials to "/auth/my-demo" the webserver will pass this request to the *my-demo* strategy. Also, in case of authentication failure, the user is redirected to the login page.

Login Request

```
router.route('/my-demo').get(function (req, res) {
  var options = [{
    "name": "username",
    "type": "text",
    "label": "company_name"
  }, {
    "name": "password",
    "type": "password",
    "label": "password"
  }];
  res.render('local', {
    auth_type: 'my-demo',
    fields: options
  });
});
```

This call on the express router ensures that when there is a get request to the "/auth/my-demo" URL, the agile-idm-web-ui login page is rendered, prompting the user for a field called "company_name" and password (as specified byt he options passed to the render function). The login page will automatically include in the possible authentication mechanisms selection all the identity providers (including the new "my-demo" strategy, as long as it has been properly registered).

5.4.2 Authentication Strategy

The authentication strategy code depends on the kind of strategy being integrated; that is to say, the code using an Oauth2 provider will differ from code using a local stragy or using other kinds of authentication; however, this example covers a simple scenario of local authentication in which the server gets a username and password from the user. For other options, the source code of the github, google, pam, agile-local or webid implementations is availble.

Here is the key part of the authentication strategy:

```
passport.use(auth_type, new LocalStrategy(
      function (username, password, done) {
        var default_exp = null;
        console.log("body"+username+"p"+password);

        db.users.findByUsernameAndAuthType(username, auth_type, function (err, user) {
          if (err) {
            return done(err);
          }
          if (!user) {
            return done(null, false);
          }
          if ( username !== "alice" &&  password !== "secret") {
            return done(null, false);
          }
          var token = tokens.uid(30);
          db.accessTokens.save(token, user.id, null, "bearer",

              [conf.gateway_id], default_exp, null, function (err) {
            if (err) {
              return done(err);
            }
            return done(null, user);
          });
        });
      }
    ));
    console.log('finished registering passport strategy');
    return true;

   } catch (e) {
    console.log('FAIL TO register a strategy');
    console.log('ERROR: error loading ' + auth_type + ' passport strategy: ' + e);
    return false;
   }
  }
```

This function specifies passport to user a local strategy, whenever the passport strategy called "my-demo" is used, as it has been referenced in the routes file.

In this function a callback is used to pass the username and password expected from the form.

Then this authentication strategy looks up the user in the local database using the *db.users.findByUsernameAndAuthType* function from the agile-idm-web-ui users database.

In case no errors happen and the user is found, this strategy checks that the user has the proper username and password. Users with different username and passwords are rejected calling the callback without error and passing the second argument as false done(null, false). In case the username and password match, the strategy must generate and store a token in the tokens database. This is done by the *db.accessTokens.save*. This function is called passing the following arguments: a randomly generated token, the user id, a null client (this will be automatically updated to store the client id of the client requesting the token, once oauth2 flow is completed automatically), the token type is "bearer", with scope [gateway_id], a null default expiration value which equals to a non-expiring token, and a null refresh token since there are no refresh tokens for this local authentication. In cases when there are refresh tokens from Oauth2 providers, they should be provided in this argument.

6 Additional Resources

In this section we include additional resources for developers. Mainly, we focus on a video and interactive documentation.

6.1 Video

We have produced a tutorial for the installation and testing of the graphical interface provided by the Oauth2 client for the authorization code flow. This video is available in the AGILE's YouTube channel (https://www.youtube.com/watch?v=ORJtNrWhd6U) and additionally, it has been posted to the resources section of the official AGILE website: http://agile-iot.eu/resources/

6.2 Documentation

Additionally, the entity REST API for AGILE IDM has been described using swagger syntax and can be opened in a regular web as shown in Figure 8 when the following link is visited:

http://petstore.swagger.io/?url=https://raw.githubusercontent.com/Agile-IoT/agile-idm-web-ui/c40cb3414ffeb7b9370b9babb70166805e59ff20/docs/api/swagger/entity-api-swagger.yml#!/User/ReadUser

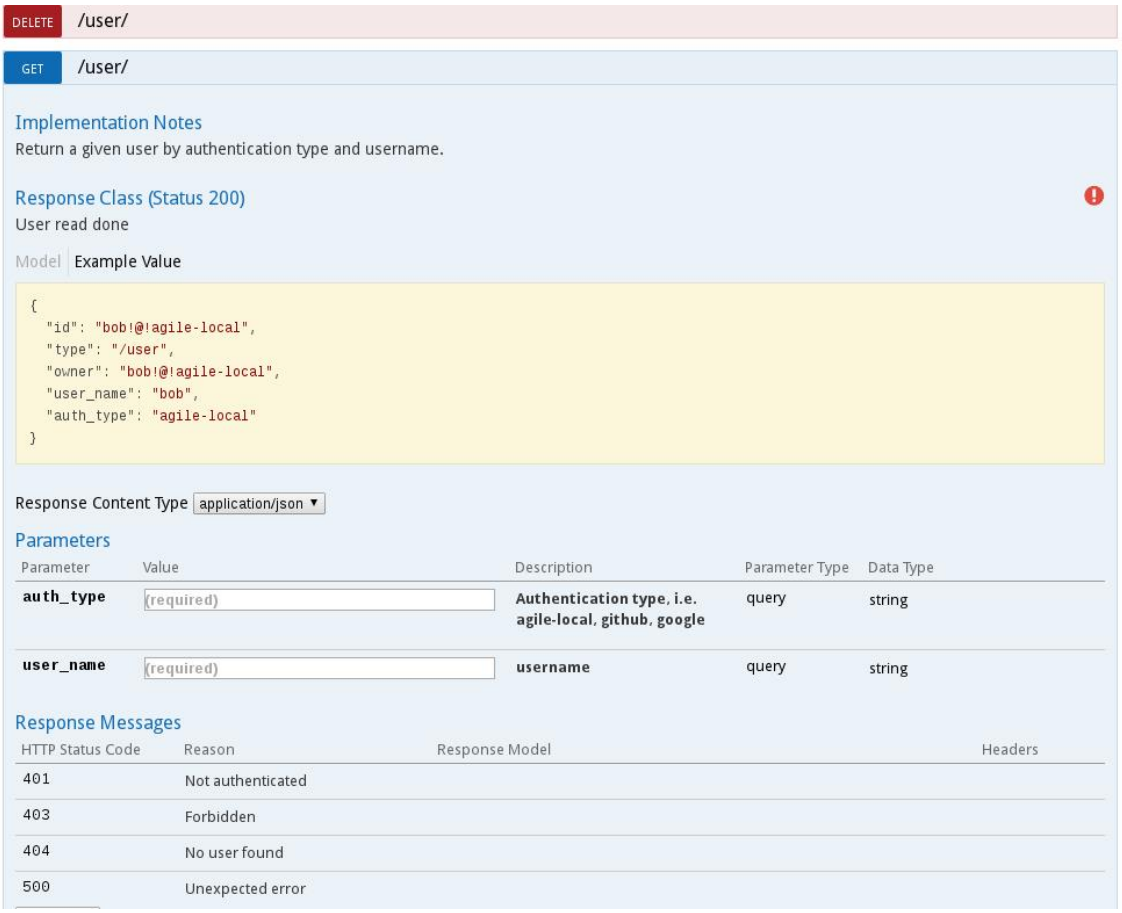

Figure 8 API Description Screenshot

7 Bibliography

Bishop, Matt. “Role Based Access Control.” In *Introduction to COMPUTER SECURITY*, by Matt Bishop, 92-95. Boston: Addison-Wesley, 2010.

Charalampos, Doukas, and Seda Polat Erdeniz. “D3.1 Requirements and Specification and SW Architecture.” AGILE Deliverable, 2016.

Cuellar, Jorge, Santiago Suppan, and Henrich Poehls. *Privacy-enhanced tokens for authorization in ACE.* IETF-Draft, IETF, 2015.

Hu, Vincent, et al. “Guide to Attribute Based Access Control (ABAC) Definition and Considerations.” NIST Special Publication 800-162 DRAFT, 2013.

Menychtas, Andreas. “D8.1 Pilot Design and analysis requirements and specification.” AGILE Deliverable, 2016.

Pernul, Günther, Guido Schryen, and Rolf Schillinger. *FORSEC Sicherheit hochgradig Vernetzter IT-Systeme.* Dec 2016. https://www.bayforsec.de/ (accessed December 20, 2016).

Schreckling, Daniel, Juan David Parra, and Wolfram Gottschlich. “D5.4.2 The COMPOSE Security Framework.” COMPOSE Deliverable, 2015.