Towards an Open, Secure, Decentralized and Coordinated Fog-to-Cloud Management Ecosystem

D4.7 mF2C Interfaces (IT-1)

Project Number	**730929**
Start Date	**01/01/2017**
Duration	**36 months**
Topic	**ICT-06-2016 - Cloud Computing**

Work Package	***WP4, mF2C Gearbox block design and implementation***
Due Date:	*M12*
Submission Date:	*22/12/2017*
Version:	*1.0*
Status	*Final*
Author(s):	*Cristóvão Cordeiro (SixSq), Daniele Lezzi (BSC), Gregor Cimerman (XLAB), Matic Cankar (XLAB), Alec Leckey (Intel), Román Sosa González (ATOS), Jens Jensen (STFC), Francisco Carpio (TUBS)*
Reviewer(s)	*Alec Leckey (Intel)* *Breogán Costa (WOS)* *Denis Ghuilhot (WOS)*

Keywords
Interfaces, design, implementation

Project co-funded by the European Commission within the H2020 Programme		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730929. Any dissemination of results here presented reflects only the consortium view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

This document and its content are property of the mF2C Consortium. All rights relevant to this document are determined by the applicable laws. Access to this document does not grant any right or license on the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the mF2C Consortium or Partners detriment.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	31/10/2017	Initial ToC	Cristóvão Cordeiro (SixSq)
0.2	16/11/2017	Add section 3	Cristóvão Cordeiro (SixSq)
0.3	05/12/2017	Merge STFC contribution	Jens Jensen (STFC), Cristóvão Cordeiro (SixSq)
0.4	05/12/2017	Merge XLAB contribution	Gregor Cimerman (XLAB), Matic Cankar (XLAB), Cristóvão Cordeiro (SixSq)
0.5	05/12/2017	Merge BSC contribution	Daniele Lezzi (BSC), Cristóvão Cordeiro (SixSq)
0.6	05/12/2017	Merge ATOS contribution	Román Sosa González (ATOS), Cristóvão Cordeiro (SixSq)
0.7	05/12/2017	Merge ATOS contribution	Roi Sucasas (ATOS), Cristóvão Cordeiro (SixSq)
0.8	14/12/2017	First Revision	Breogán Costa (WOS), Denis Guilhot (WOS)
0.9	15/12/2017	Add reviews and finalize	Cristóvão Cordeiro (SixSq)
1.0	22/12/2017	Quality check and final version to submit	Lara Lopez (ATOS)

Table of Contents

List of figures

List of tables

Executive Summary

The mF2C project is focused on delivering a management framework for the challenging fog-to-cloud domain.

This document described the design and implementation of the different interfaces required by mF2C components, including those in the data plane devoted to connecting the different agents and those in the control plane facilitating overall control.

The content presented in this deliverable is aligned with the ongoing technical work in T5.1, for IT-1.

1. Introduction

1.1 Introduction

Use cases in the mF2C project cover Smart City, Smart Boat and Smart Airport. These services leverage the mF2C platform to establish communication between nodes and the cloud in an attempt to connect across multiple underlying technologies. This enables the services to access components in a secure and resilient manner, consistently. Unreliable network connectivity, fast delivery of important information, low power consumption and reliable data storage were all taken into account when designing the mF2C platform. Enabling uniform communication was established early in the implementation. An important feature of the common interface is inter-module communication. In this document, the common interface will be introduced, which solves some of the common problems with inter-module communication.

In the first part, the mF2C system communication is presented and the challenges in Fog network and how to transfer information to Cloud are explained. The second part will contain Platform Managers interface and how the CIMI interface works to standardize the communication, which is then further explained in the following section where components' interfaces are exposed and functionality is documented. Lastly, the data management interface is presented, with the description of how the data resources are managed and accessed in Cloud and Fog.

1.2 Purpose

The objective of this deliverable is to present the results of M12 design and implementation of different interfaces required by mF2C and describe interfaces of communication systems and their interaction with different components which are exchanging information. Finally, the interfaces for transparently managing data in the Cloud and Fog will also be detailed.

1.3 Glossary of Acronyms

Acronym	Definition
AC	mF2C Agent Controller
API	Application Programming Interface
CIMI	Cloud Infrastructure Management Interface
DMTF	Data Management Task Force
GNSS	Global Navigation Satellite System
HTTP	Hypertext Transfer Protocol
ID	Identification
JSON	Javascript Object Notation
PM	mF2C Platform Manager
QoS	Quality of Service
REST	Representational state transfer
UC*x*	mF2C Use Case *x* (where *x* is 1, 2, 3)
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
XML	eXtensible Markup Language

Table 1. Acronyms

2. mF2C System Communication

2.1 Agent Interactions

Within the mF2C application, all modules can communicate with each other and even reach out to remote agents. For these interactions to occur as smoothly and consistently as possible, interfaces have been designed to help communications within the same agent, and also to other agents, on any mF2C platform layer.

2.1.1 Horizontal/Vertical Communication

The mF2C system relies on the deployment of the mF2C agent on the set of devices willing to join the F2C area and with capacity enough to run the agent demands. From that assessment and as already reported in previous deliverables, the mF2C ecosystem puts together a set of devices running the mF2C agent, thus endowed with all the envisioned mF2C functionalities and other devices (IoT elements at the very edge) that do not run the mF2C agent. The latter (usually "dummy" sensors and actuators at the edge), are considered as IoT devices that must be connected to a device running the mF2C agent through mF2C interfaces, to be part of the mF2C ecosystem and thus enabling other devices to reach out to them. The project defines such a communication as vertical, but this one is not the only one. Indeed, different devices running the mF2C agent in the envisioned hierarchical architecture will also communicate, thus setting "inter-layer" communications through mF2C interfaces, to enable wide access to any mF2C capable device. Different from vertical communication, horizontal communication, that is "intra-layer" communication may be also designed. In short, enabling horizontal communication will prevent the need for a device "A" in a particular layer willing to connect to another device "B" in the same layer, to handle the communication through a device "C" (connected to both) deployed in a higher layer. In other words, enabling horizontal communication would replace the need for vertical communication in those scenarios where 2 devices in the same layer want to communicate each other. As said, we do not resign to go for it, instead, for the sake of deployment priorities; we shift this capacity to be deployed in IT-2.

Regarding the horizontal communication, it is worth highlighting the specific scenario brought in when considering the edge communication, which is connecting IoT devices to a device running the agent. Indeed an IoT device, for example a sensor, must be connected to the agent device in order to share or use its resources. The sensor itself does not have the capacity for communicating with other agents nor was designed for it. Thus, the agent embeds some functionalities to collect IoT devices information. As for IT-1, this data collection process will be handled manually, that is, the owner of the agent device will manually introduce through a form the characteristics of the different IoT devices connected to this particular agent. This process will be improved for IT-2 turning into an automatic process with no need for the agent device owner to fill in a form. The mF2C system assumes that security is granted, so all processes required to keep this information updated and the communication alive must be carried out under strict security guarantees. When the sensor is categorized, its specific functionality is also determined, for example, temperature sensor offers the temperature, tracking sensor offers GNSS location (e.g.: GPS) and G-force measurements, and so on. Additionally, a policy is also allocated to each particular IoT device, describing usage and rules to be considered while using the sensor. As a consequence, each device running an agent has a database including information about its own resources (CPU, memory, etc.) and the different IoT devices connected to it. This information is later forwarded upstream in the hierarchy through the mF2C interfaces. In this scenario two devices running an agent are communicating with each other to support several mF2C functionalities, such as discovery, identification or categorization.

Indeed, all communications will be handled through the mF2C interfaces, enabling when needed, the three processes above, and thus enabling that a resource registered to the mF2C ecosystem may be offered globally to all users/services within the mF2C system (according to the applied policies).

Certainly such resources availability will depend on mF2C permissions allocated to specific services /users so that access to a particular IoT resource may be granted or denied, although this decision must be carefully handled for real-time applications. All in all, devices with the mF2C agent installed will communicate to IoT devices, will put them all in a list of available resources and finally will handle the access to its IoT devices according to the policies in place (indeed these policies are also applicable to the own device resources)

2.1.2 Enclosed Communication

During the workshop hosted by STFC in November 2017, the following main types of communication were identified:

- REST web services;
- Non-rest (typically a proprietary protocol such as Google RPC).

Two of the main aspects of communicating are security and efficiency: the communication within the mF2C infrastructure must comply with the data security policy described in D3.1, yet the security and communications infrastructure must not impose too much of an overhead.

The following options for securing web services were discussed. Note that the intention is to provide all three methods through the secure communications library, so the best appropriate methods are always selected:

- Securing the socket. This means that the web server must authenticate itself with a host certificate (cf. RFC 2818, section 3.1) which is issued to the DNS name of the host. The client may authenticate itself as well;
- Securing the protocol: in REST, communications are carried over HTTP (RFCs 2616 and 7540), and client authentication can follow the standard HTTP protocol (cf. RFC 2617). Similarly, authorisation can be carried in the HTTP header, e.g. RFC 6750;
- Finally, the message itself can be secured. If JSON is used, it is possible to both sign and encrypt the message (RFCs 7515, 7516). However, in order to do this, the client must be in possession of a suitable key.

The options are not mutually exclusive: the library will likely need to support all three even for IT-1. Naturally, the support in the communications library should be transparent to the caller, leaving the application to focus on the business logic.

For securing the general communications, there are again three options:

- Secure the socket carrying the communications, as before;
- Make use of any security features of the communication, which may or may not be the same as 1 – e.g. Google RPC supports secure sockets with options for client authentication via the socket or via token based authentication; however, it should still be routed via the communications library in order to make use of the correct client identities.
- Wrap the communications: for example JWE (RFC 7516) is capable of encrypting an arbitrary octet string.

Regarding efficiency, the principles is:

- Avoid premature optimisation;
- Encrypt only when necessary: the data policy requires encryption only for PRIVATE data and only when communicated over untrusted networks.

Reuse communications channels: e.g. keeping sockets open when possible; make use of TLS context caching to ease the re-establishment of a socket, use HTTP keep-alive, etc.

2.1.3 Outward Communication

mF2C components sometimes need to communicate with entities outside of mF2C: after all, “the edge” is about interfacing the data and compute infrastructure to the environment around it. The most obvious such entity is the user: if they have an mF2C app installed on their phone, then information leaves the sphere of control of mF2C when this app shows information to the user.

Other external components can include the following:

- An external billing service, to which account records are sent, in order to invoice – or credit – the user for services rendered to (resp. by) them (UC2);
- An external cloud service provider;
- A device being controlled by (or otherwise interfaced to) mF2C services (e.g. a traffic light in UC1, or a boat in UC2).

The issues arising are:

1. The security policy (D3.1) should still be honoured, e.g. if private data is sent outside of mF2C, it should be sent only to the person whom it concerns;
2. How does the external entity know it can trust the information – they are not themselves part of the mF2C security infrastructure and cannot (easily) validate signatures;
3. Similarly, how entities are referenced inside and outside mF2C may change: a user, for instance, may have a unique identifier inside mF2C, but have a different identifier with their payment service;
4. Data sent outside of mF2C is no longer under the control of mF2C, and could be misused, thus leading to a bad reputation, or worse, for mF2C.

One option is to implement a gateway which processes data and connects mF2C’s data model with that of the outside world (i.e., security of cloud service providers, GDPR, etc.). Thus, for any given external entity, there should be a well-defined entity in mF2C which is responsible for communicating with it, translating security tokens etc., and for ensuring that the data security policy is honoured. For end users, this would be the mF2C services on their device, assuming the device is personal.

2.1.4 Inward Communication

The distributed environment proposed by mF2C comes with an evident need to share information and coordinate management operations amongst multiple nodes (agents), which can either be in the same network and next to each other, or running on remote locations, in a different layer of the mF2C platform.

For this reason, the Platform Manager comprises a standardized and IT ready interface which offers a single point of access for all operation and requests made to an mF2C agent. The implementation of this interface is covered in detail in section 0.

By restricting incoming requests to a single interface, the inner components and software running in the mF2C application will neither be exposed nor directly accessible from outside their own agent, thus strengthening the application's security.

The interface will be able to address any system resource that needs handling, either managing it directly or outsourcing the request to the respective mF2C module.

2.2 Data Plane Communication

The data plane communications in mF2C happen within the Data Management components in the PM and in the AC. Each agent has its own local database, which stores information about the agent,

the device associated to it, and the user that owns the device, including how the user wants to contribute the device to mF2C. Also, this database may store data from applications or services running on top of the platform, if the user allows it.

Despite mainly storing its own data, an agent can share data with another agent, in such a way that the shared data is accessible as if it was local from the point of view of the applications or the mF2C components that use it. This access will cause underlying data plane communications between the agents in different ways, depending on whether the data is for the management of mF2C or data used by applications.

Regarding the data to support the management of the mF2C platform (agents, devices, users…), an agent can only share it with its leader in a bottom-up fashion. This implements a design decision of mF2C, where each leader must know about all the devices in its cluster, but a regular device does not need to have any information about other devices.

On the contrary, communications involving application data don't have the same restrictions. Application data may be data created by applications or services running on the mF2C platform, or data ingested from sensors and consumed by applications. This data can be stored when read from the sensors, or may be accessed through the agent that is connected to the sensor without persisting it in the agent. This decision depends on the application requirements, and the data management component supports both of them. In this context, any agent must be able to reach the data required for the execution of a service or application, regardless of whether or not it belongs to one of its children. Thus, application data communications are horizontal and direct between agents, without the need of traversing the hierarchy.

In both cases, the data can be used as if it was local. However, in order to avoid communications in the mF2C infrastructure and to provide resiliency to failures in the connections, an agent can create a local copy of (part of) the data held by other agents. This copy will be synchronized with the original one, following the consistency policy defined in the data model registered in the Data Manager. The consistency model can be different for every type of data. The current implementation follows a strong consistency model for all types of data, in such a way that communications between agents will only happen when some of the copies is modified.

For example, a leader can create a local copy of part of its children's mF2C data. This copy will be synchronized with the children's ones, by means of leader-to-child or child-to-leader communications that will only happen when some of the copies is modified. For application data, the behaviour of the synchronization is the same, but this time the communications are not restricted to bottom-up or top-down communications.

3. PM Interface

As already introduced in deliverable D4.3, the PM will be responsible for the orchestration of services based on the compute, storage and network resources and using a full-stack monitoring system, which receives telemetry data from different sources. It provides the interoperability necessary to communicate with other mF2C agents within the heterogeneous platform architecture defined in deliverable D2.6.

Since all the requests to the PM will address some sort of infrastructure resource, the CIMI [1] standard interface specification has been chosen, alongside with the RESTful HTTP-based protocol, as the resource management model for the mF2C infrastructure resources.

CIMI provides a standard for the management of any resource (services, applications, credentials, devices, etc.) within mF2C. Beside its ease of use and wide industry support, this high-level interface provides:

- consistent resource management patterns, making it easy to develop small, lightweight and infrastructure agnostic clients;
- auto discovery of new resources without changing clients, enabling dynamic evolution of the platform;
- standard mechanism for referencing other resources;
- flexibility to cover a wider range of resources than strictly those related to cloud infrastructure management.

All requests are translated into operations which are based on the HTTP protocol, allowing the most common HTTP verbs *GET, HEAD, PUT, POST* and *DELETE*, with the option to have a JSON message body. All custom operations which are non-CRUD are modelled as HTTP *POSTs* to the specific operation URI on the CIMI resource.

3.1 Handling Requests with the CIMI Server

Figure 1 illustrates how the CIMI server interface handles incoming requests for managing an infrastructure resource (that might be a service, a user, a device, etc.).

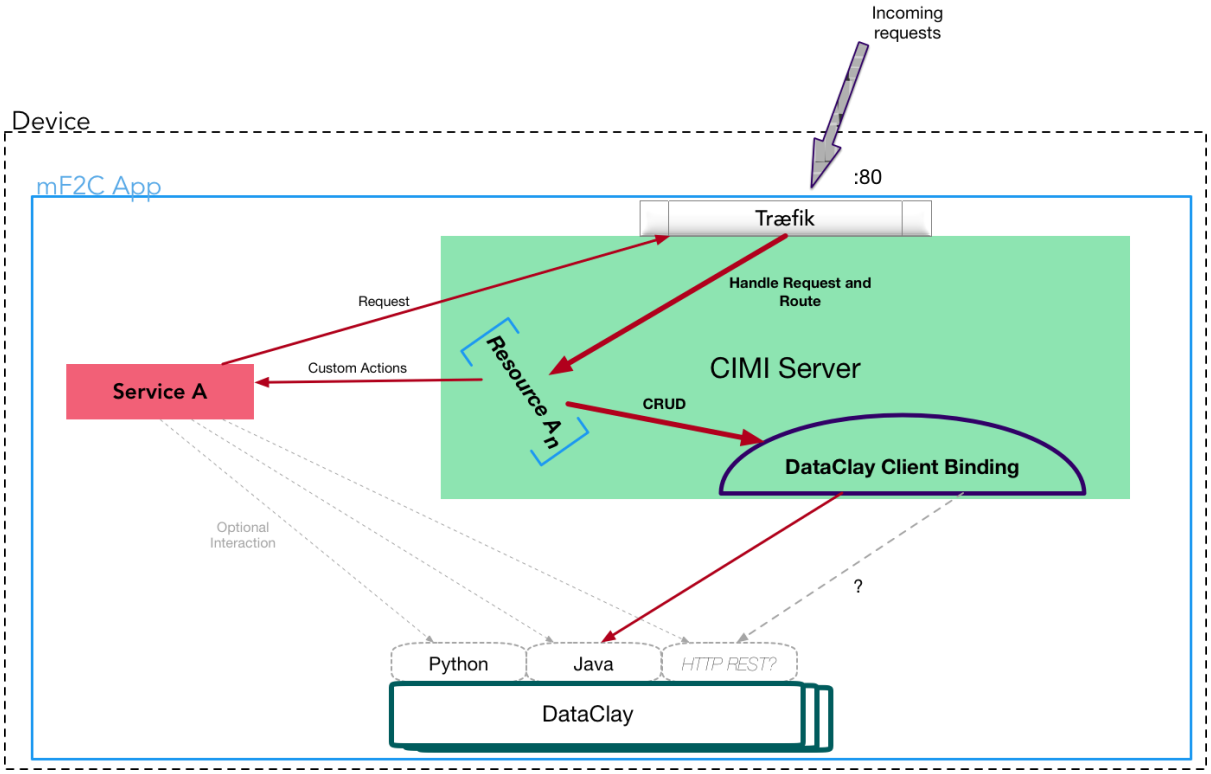

Figure 1 Managing a resource through CIMI

As Figure 1 shows, modules within the mF2C application can also communicate with the data management libraries to access the infrastructure resources. By doing so, one must assure that all data representation conventions are kept, and that proper resource validation is performed before actually making any write operations, otherwise it might break the resources' lifecycle as the CIMI server will no longer be able to manage (or even identify) those resources.

The CIMI server is written in Clojure [2] and runs as a standalone micro service. For handling HTTP requests CIMI uses a web application library called Ring [3] to build modular, customized and re-usable components for processing requests and building responses through handlers and high-level functions.

This simple Ring application server is used within a generic container running Aleph [4] for client-server networking, containing the handler and middleware summarized in Figure 2.

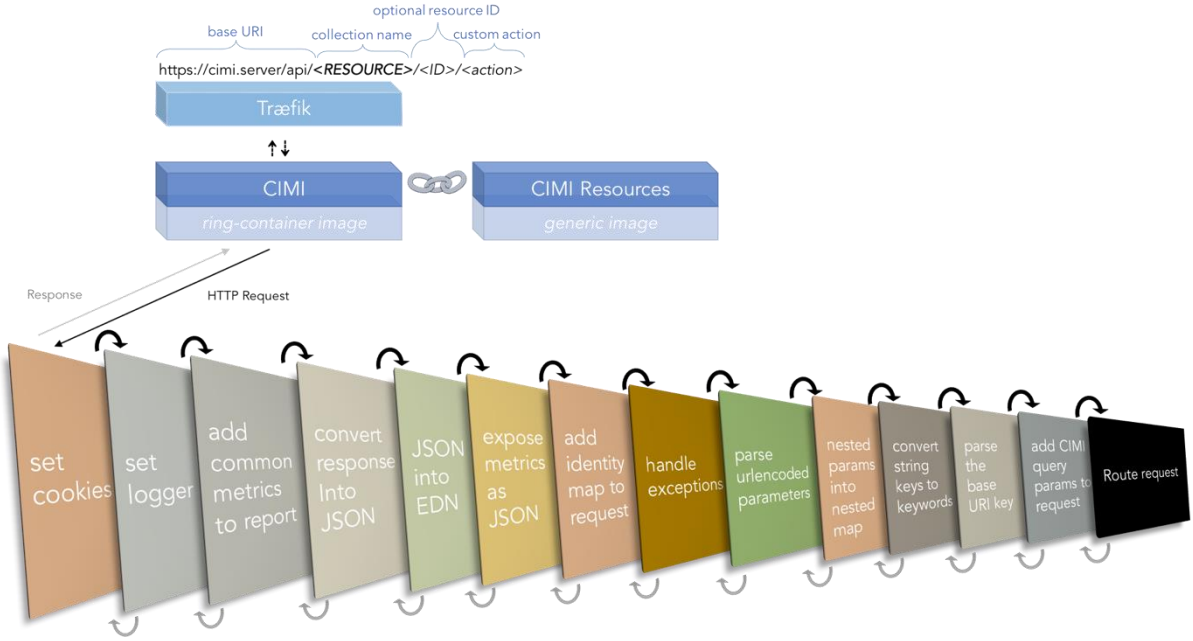

Figure 2 Ring container HTTP handler and middleware description

With the CIMI server, all the functionality for user authorization and authentication (via user/password, OAuth, API keys and OIDC), HTTP request serialization, content validation, logging and dynamic routing are provided out of the box, and for this reason the proposed main path for data management within the mF2C application is illustrated by the red arrows in Figure 1:

I. most (if not all) of the incoming requests to the mF2C agent will go through the CIMI server;
II. the CIMI server will handle the requests and return the dynamic routing for the respective defined resource in server's classpath;
III. all CRUD operations will be handled by DataClay Java binding within the CIMI server;
IV. custom and module specific operations shall be outsourced as asynchronous jobs (see section 5.17.1 of the CIMI specification [1]) to the respective application components/modules, using an HTTP client binding;
V. if other application components/modules/services also need to perform any sort of data management, there will be two different paths:
    i. have all CRUD operation be handled by the CIMI server by using its RESTful interface locally, or
    ii. interact directly with DataClay (using either the Python or Java client libraries), **respecting however**, the compliance guidelines for data management described in section 3.2.

For any user/costumer of the CIMI interface (may that be an end-user, a developer or an application), the API server provides advanced features for manipulating results when searching collections of resources, through request query parameters which are either recognized and supported by the interface or silently ignored otherwise. These CIMI-defined queries have already been detailed in section 6.1.3 of deliverable D4.3.

3.2 Developers' Interface - Compliance Guide

As it will be detailed below in section 4, each mF2C module (or component) will be exposing its own interface for the other components to communicate with, including the CIMI server.

In order to keep each component as application agnostic as possible, the design and implementation of the components' APIs shall:

- be RESTful;
- be self-discoverable at */api*;
- be documented;
- be secured when exposed and reachable from outside the mF2C agent where they run;
- respond with the standard HTTP status codes;
- handle request bodies and response types in JSON format, in which field names must be *camelCase*;
- when applicable, provide versioning (i.e. */api/v1*).

From an architectural styling perspective, developers must be aware of the REST designing style that the CIMI server abides by, which might also be useful for individual components when implementing their own REST APIs:

- "The key abstraction of information in REST is a resource" [5];
- CIMI resources are representations of actual virtual or physical resources available in the infrastructure;
- resource shall be uniquely identified by an URI;
- resources are stateless;
- the interface only provides HTTPS endpoints;
- access control is performed locally for each API endpoint and respective concerning resource;
- every HTTP method which is not allowed is rejected with a *405* response code [6];
- all input (including body content type) is validated before the requested operation occurs;
- unexpected or missing content types shall be rejected with either *406* or *415* response codes;
- when handling and responding to errors, generic error messages shall be used, without any technical details of the internal implementation of the API;
- logging is a must;
- sensitive information shall never be passed in the endpoint URL;
- for POST and PUT methods it should go in the request body;
- for GET method it should go in an HTTP header;
- resource and response representations are encoded in JSON (*"application/json"*);
- all identifiers (resource names, attributes, etc.) are case sensitive, do not start with a digit and shall only use the set of characters defined in section 5.2 of the CIMI specification [1];
- resources are serialized as JSON objects wrapping all their attributes;
- a *CloudEntryPoint* special resource is provided, with READ and UPDATE operations, allowing the user/consumer to discover and access all the existing infrastructure resources which he/she can use.

As stated in Figure 1, components might also want or need to interact directly with DataClay (either for the ease of use or for very specific resource management operations). In these cases, as said above, data consistency must be ensured otherwise the application's business logic might be compromised if different services access and modify the same data asynchronously and without any respect for the data structure and data management constraints (a simple example would be a component asynchronously modifying a resource attribute which is used as a unique identifier, and because the operation does not go through a traditional relational database and there is no pre-validation on the component side, one might end up violating a primary key constraint, causing a conflict between two unrelated system resources and consequently raising exceptions on other application components using the same resource).

Being said this, developers should consider the following common (section 5 in the CIMI specification [1]) attributes which might be present on the infrastructure resource's schema:

resourceURI: unique URI [7] associated with the type of (CIMI) resource being serialized. It can either be a URL or URN. For the user/consumer, this is an optional attribute.

operations: some of the resource representations might include this as a list. This is set on the server side, based on the resource ACLs set by the user/consumer. Each operation includes a *"rel"* and *"ref"* field, for uniquely identifying the operation and the URI to which the operation's request shall be sent, respectively.

properties: each resource has an attribute placeholder called "properties", which users/consumers can (optionally) use to store any key/value pair, knowing that the server will not try to understand nor try to take any action based on these values.

id: immutable and unique URI, infrastructure wise, identifying the resource. When augmented to an URL, it provides the endpoint for querying the respective resource.

name: human-readable name of the resource assigned by its creation.

description: human-readable description of the resource, also assigned by its creator.

created: immutable "dateTime" timestamp (serialized as string in JSON) of when the resource was created. Its format shall be consistent and unambiguous.

updated: initially the same as *"created"*, it is a mutable "dateTime" timestamp of when the last explicit attribute update was made on the resource.

parent: a reference to the parent resource, only applicable when composition relationship between resources exists.

resourceMetadata: reference to a "ResourceMetadata" instance associated with the resource.

acl: used for access control, contains an "owner" attribute to identifying who owns the resource, and a "rules" list, enumerating the different rights and types of accesses other users have on the resource. The *"operations"* attribute is dynamically built based on this.

The above attributes, either optional or mandatory, are fields which the CIMI server will always consider as common and acceptable attributes. Any other attribute that might be included in a resource representation will either be intrinsically defined in the resource's schema or have a dynamic nature, being capable to adopting any field name as long as prefixed with a known namespace. The following snippet exemplifies a GET request for a *mockResource* identified by *ID*:

```
GET /api/mock-resource/d94cb989-5ecb-4ff8-b8b5-b2abde7e59ac
Host: https://cimi.server
Accept: application/json

{
  "id" : "mock-resource/d94cb989-5ecb-4ff8-b8b5-b2abde7e59ac",
  "created" : "2017-10-03T12:55:22.820Z",
  "updated" : "2017-10-03T12:55:22.820Z",
  "acl" : {
    "owner" : {
      "principal" : "test",
      "type" : "USER"
    },
    "rules" : [ {
      "type" : "ROLE",
      "principal" : "USER",
```

```
        "right" : "VIEW"
      }, {
        "type" : "ROLE",
        "principal" : "ADMIN",
        "right" : "ALL"
      } ]
    },
    "operations" : [ {
      "rel" : "edit",
      "href" : "mock-resource/d94cb989-5ecb-4ff8-b8b5-b2abde7e59ac"
    }, {
      "rel" : "delete",
      "href" : "mock-resource/d94cb989-5ecb-4ff8-b8b5-b2abde7e59ac"
    } ],
    "resourceURI" : "http://sixsq.com/slipstream/1/MockResource",
    "namespace:freetext" : "whatever",
    "mockResourceTemplate" : {
      "href" : "mock-resource-template/mocker"
    }
}
```

Translating the above snippet to a descriptive schema, we get:

```
(…)

{
  "id" -> real and unique URI, read-only and mandatory, created by
  the CIMI server. Should not be modified by other components,
  "created" -> a date time that can be dynamically created by the
  CIMI server, which also is mandatory and read-only,
  "updated" -> read-only and mandatory date time timestamp, edited on
  every resource update,
  "acl" -> user can optionally set the ACLs. If not set, defaults
  will be applied. Specifies who owns the resource and who can do
  what with it,
  "operations" -> not mandatory, but read-only, generated by the CIMI
  server according to the above ACLs,
  "resourceURI" -> mandatory and read-only. The CIMI server will have
  a schema default depending on the type of resource,
  "ns:freetext" -> this is an extra attribute. The user/consumer
  needs to ensure (create if needed) that the namespace "ns" exists
  and then the suffix (anything after ":") can be any text one
  desires,
  "mockResourceTemplate" -> some resources can rely on
  ResourceTemplates which basically represent the set of metadata and
  instructions used to instantiate some other Resource (in this case,
  the MockResource). Template Resources usually have "describe"
  actions which users/consumers can use to find out what the resource
  schema is and which attributes are mandatory
}
```

3.3 mF2C Resources

This section proposes an initial set of resources for the mF2C infrastructure that need to be created and made available. The following schemas present a minimal set of attributes to be included in the resources representation (either by the user/consumer or by the interface providers), respecting the foundation defined above in section 3.2. Please note that all attributes and overall resources are merely propositions at this stage, being subject to modifications throughout IT-1 and IT-2.

Remarks:

id, created, updated, resourceURI and *operations* are controlled and filled in automatically by the CIMI server

3.3.1 Machine

Based on the Machine resource (see table 16 in the CIMI specification[1]).

```
{
  "id": URI,
  "name": string,
  "description": string,
  "created": dateTime,
  "updated": dateTime,
  "properties": { string: string, … },
  "acl": {
    "owner": { "principal": string, "type": string },
    "rules": [
      { "type": string, "principal": string, "right": string },…
    ]
  },
  "operations": [ { "rel" : string, "href" : URI }, … ],
  "resourceURI": URI,
  "state": string,
  "cpu": integer,
  "memory": integer,
  "disk": integer,
  "cpuArch": string,
  "cpuSpeed": integer
}
```

Remarks:

- *state* should only allow the following values: CREATING, STARTING, STARTED, STOPPING, STOPPED, PAUSING, PAUSED, SUSPENDING, SUSPENDED, CAPTURING, RESTORING, DELETING, ERROR, FAILED;
- *cpuArch* should only allow the following values: 68000, Alpha, ARM, Itanium, MIPS, PA_RISC, POWER, PowerPC, x86, x86_64, z/Architecture, SPARC;
- this is not a CIMI Machine resource (for the time being) as some attributes have been simplified.

3.3.2 Agreement

```
{
  "id": URI,
  "name": string,
  "created": dateTime,
  "updated": dateTime,
  "resourceURI": URI,
  "state": string,
  "contract": {
    "id": string,
    "type": string,
    "name": string,
    "provider": { "id": string, "name": string },
    "client": { "id": string, "name": string },
    "creation": dateTime,
    "expiration": dateTime,
```

[1] http://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0.pdf

```
      "guarantees": [
        { "name": string, "constraint": string }
      ]
    },
    "assessment": {
      "first_execution": dateTime,
      "last_execution": dateTime
    }
  }
```

Remarks:

- state should only allow the following values: started, stopped, terminated.
- contract.type should only allow the value “agreement”. The value “template” is reserved for SLA templates, a functionality not foreseen for IT-1.

3.3.3 User

Resources managed by the User Management module:

User:

```
{
      "id": URI,
      "name": string,
      "description": "user ...",
      "created": dateTime,
      "updated": dateTime,
      "resourceURI": URI,
      "profile": { href: profile/id },
      "sharingModel: { href: sharing-model/id}
}
```

Profile:

```
{
      "id": URI,
      "name": string,
      "description": "profiling ...",
      "created": dateTime,
      "updated": dateTime,
      "resourceURI": URI,
      "id_key": string,
      "email": string,
      "service_consumer": boolean,
      "resource_contributor": boolean
}
```

Sharing model:

```
{
      "id": URI,
      "name": string,
      "description": "sharing model ...",
      "created": dateTime,
      "updated": dateTime,
      "resourceURI": URI,
      "max_apps": integer,
      "GPS_allowed": boolean,
      "max_CPU_usage": integer,
      "max_memory_usage": integer,
      "max_storage_usage": integer,
      "max_bandwidth_usage": integer,
      "battery_limit": integer
}
```

4. Components' Interfaces

Technical details about the REST interfaces that each component will be exposing.

Explain the reasoning behind the choice of interface and slightly introduce how it works, generally speaking.

Address each component individually, providing a summarized technical explanation of their interface.

4.1 Lifecycle Manager

The Lifecycle Management is responsible for managing the lifecycle of the applications to be executed by the mF2C infrastructure, including the submission and termination of these applications, among other operations.

Get available methods

Returns the list of available methods to call on the lifecycle interface

- URI /api/v1/ lifecycle /
- Method: GET
- Params: none

Submits a service

Submits a service and returns a JSON object with the result / status of the operation.

- URI /api/v1/lifecycle
- Method: POST
- Params: none
- Body: service properties (JSON object)

Terminates a service

Submits a service and returns a JSON object with the result / status of the operation.

- URI /api/v1/lifecycle
- Method: DELETE
- Params: none
- Body: service properties (JSON object)

Service operation & termination

Starts / stops / restarts / terminates a service and returns a JSON object with the result / status of the operation.

- URI /api/v1/lifecycle
- Method: PUT
- Params: none
- Body:
    - service properties
    - operation: start / stop / restart

Service status

Gets the service status and returns a JSON object with the status of the service

- URI /api/v1/lifecycle/:id_service
- Method: GET

- Params: service ID

Notifications and warnings handler

Process notifications (SLA Manager) and warnings (User Management Assessment)

- URI /api/v1/lifecycle/:id_service
- Method: PUT
- Params: service ID
- Body:
    - Notification / warning content

4.2 Landscaper

The landscaper is a tool to build an architectural model of a computing infrastructure for a fog/cloud infrastructure. The API methods allows a developer to retrieve elements in graph form

Get available methods

Returns the list of available methods to call on the landscape interface

- URI /api/v1/landscape/
- Method: GET
- Params: none

Get Graph

Returns the full landscape graph as node-link JSON object

- URI /api/v1/landscape/graph/
- Method: GET
- Params: none

Get Sub-Graph by node id

Returns the subgraph using a node id as the starting point

- URI /api/v1/landscape/subgraph/<node_id>
- Method: GET
- Params: node_id The identity of the node which will be used to extract the subgraph

Get Sub-Graph by Property

Returns a subgraph of all nodes matching the supplied properties. Currently used properties include: 'name', 'layer', 'category', 'type', 'attributes'

- URI /api/v1/landscape/subgraph/
- Method: GET
- Params: properties: Dictionary of properties, keys and values

Get Node by Id

Returns a subgraph using a node id as the starting point

- URI /api/v1/landscape/node/<node_id>
- Method: GET
- Params: node_id The identity of the node which will be used to query

Get Machines

Returns a list of physical machines in this landscape

- URI /api/v1/landscape/machines

- Method: GET
- Params: none

4.3 SLA Manager

The SLA Management component is responsible for managing the SLAs between the parties involved in a service on the mF2C platform: the platform and the platform users.

The API resources on IT-1 are the provider and the agreement, and the API provides methods for CRUD and other management operations.

Get Providers

Returns the list of providers

- URI /api/v1/sla/providers
- Method: GET

Get Provider

Returns a provider by its ID.

- URI /api/v1/sla/providers/<provider-id>
- Method: GET
- Params
- provider-id (URL): the ID of the provider

Create Provider

Creates a provider.

- URI /api/v1/sla/provider
- Method: POST
- Body: the provider to create.

Delete Provider

Deletes a provider.

- URI /api/v1/sla/providers/<provider-id>
- Method: DELETE

Get Agreements

Returns the existing agreements, optionally filtering by one or more properties. The mandatory property for IT-1 is the active property.

- URI /api/v1/sla/agreements
- Method: GET

Get Agreement

Returns an agreement by its ID.

- URI /api/v1/sla/agreements/<agreement-id>
- Method: GET

Create Agreement

Creates a service agreement. The SLA Management (at the leader) must check that the required resources are available at the current level.

- URI /api/v1/sla/agreements

- Method: POST
- Body: the agreement to create.

Update Agreement

Updates the content of an agreement. On IT-1, the text of an agreement cannot be modified; only metadata. This method is intended to modify the evaluation status of an agreement, i.e., it is going to be evaluated or not.

- URI /api/v1/sla/agreements/<agreement-id>
- Metod: PUT

4.4 Recommender

The Recommender module stores recipes of deployment configurations that best match services to be executed. This is based on the results of the analytics module.

Get available methods

Returns the list of available methods to call on the landscape interface

- URI /api/v1/recommender/
- Method: GET
- Params: none

Get Recipe

Returns a deployment configuration for this service identifier. If this service has had its performance analysed before then this configuration will exist, otherwise no config will be returned

- URI /api/v1/recommender/recipe/
- Method: GET
- Params:
- ServiceId: id of service to be deployed.
- tags: A dictionary of tags, used to refine the query

Store Result

Stores the results of an analysis task that was performed on this service's deployment.

- URI /api/v1/recommender/recipe/store
- Method: POST
- Params:
- ServiceId: id of service to be deployed.
- Results: JSON of results, output of the analysis task
- tags: A dictionary of tags, metadata of the results

4.5 Task Manager

The Task Manager receives the request to start an application (composed of tasks) from the Lifecycle Manager.

Start Application

Receives the request for starting a COMPSs application. If the task

- URI /api/v1/executeApp
- Method: POST
- Params: app_class, method_name, arguments
- Body:

- Input: JSON with the application definition
- Results: JSON with application ID

Get Application Status

Receives the request for the status of an application

- URI /api/v1/getActivityStatus
- Method: POST
- Params: application ID
- Body:
- Input: JSON with the application ID
- Output: JSON with the status of the application

4.6 Task Scheduler

The Task Scheduler receives the order to execute a task. A Task includes the dependencies with other tasks. It also maintains a list of resources, constantly updated by the Policies, where to execute the tasks. This is an internal API used by the runtime.

Execute Task

Receives the request for the execution of a task of a running application

- Method: public void executeTask(TaskProducer producer, Task task)
- Params:
    - TaskProducer: process to be notified when the task ends.
    - Task: task, with dependencies, to be executed.

Update Resources

Receives the list of resources from the policies and updates/adds/removes a node in the Decision Engine

- Method: public <T extends WorkerResourceDescription> void updatedResource(Worker<T> r, ResourceUpdate<T> modification)
- Params:
    - Worker: resource to be modified.
    - ResourceUpdate: action on the resource.

4.7 PM Data Manager

The Data Management receives requests from the Task Scheduler in order to perform the following operations. Both the Data Management and the Task Scheduler are based on java-based technologies and, thus, they communicate through a Java API to avoid unnecessary latencies.

Get Object by ID

Returns the reference to a persistent object identified by objectID.

- Method: public Object getObjectByID(String objectId)
- Params: The object identifier of the requested object
- Return: A reference to the object so that it can be manipulated

Get location of an object

Returns a list with the locations of the corresponding object.

- Method: public String[] getLocations(String objectId)
- Params: The object identifier of which the locations are requested

Return: The set of locations where the object is replicated

Replicate object

Replicate an object on a specific node.

- Method: public void newReplica(String objectId, String backendId)
- Params: The object identifier of the object to be replicated, and the destination node
- Return: None

Create a version of an object

Replicate an object on a specific node with a new version.

- Method: public String newVersion(String objectId, String backendId)
- Params: The object identifier and the destination node
- Return: The objectId given to the version created

Consolidate a version of an object

Commits the changes made to a version of an object.

- Method: public void consolidateVersion(String versionId)
- Params: The object identifier of the version
- Return: None

4.8 PM Policies

This component controls the amount of resources available to the Task Scheduler.

Manage Resources

Add a resource available to the runtime

- URI /api/v1/scheduler/resource/<resource_name>/
- Method: PUT
- Params:
  - resource_name: ip or hostname
- Body: JSON with the resource definition
- Results: none

Updates the resources available to the runtime

- URI /api/v1/scheduler/resource/<resource_name>/
- Method: POST
- Params:
  - resource_name: ip or hostname
- Body: JSON with the updated resource definition
- Results: none

Updates the resources available to the runtime

- URI /api/v1/scheduler/resource/<resource_name>/
- Method: DELETE
- Params:
  - resource_name: ip or hostname
- Body: none
- Results: none

4.9 Intelligent Instrumentation

The Intelligent Instrumentation module will monitor telemetry metrics in order to potentially throttle the collection parameters of probes depending on device constraints.

Currently this module is set to run as a daemon/service and will not have a public API.

4.10 Distributed Query Engine

The Distributed Query Engine provides an abstraction layer to accessing telemetry data from multiple sources from a single URI.

Get available methods

Returns the list of available methods to call from the query engine

- URI /api/v1/queryengine/
- Method: GET
- Params: none

Show Metrics

Retrieves a list of metrics which are available for querying. Tags help to refine the metrics list, by narrowing the search window, such as setting the source to one particular machine.

- URI /api/v1/queryengine/metrics
- Method: GET
- Params:
- tags: A dictionary of tags, used to refine the query (should include source=identify_of_source_node)

Get Metric

Retrieves a list data points for a metric between the start and end time specified.

- URI /api/v1/queryengine/<metric>
- Method: GET
- Params:
- <metric>: Name of the metric to retrieve
- start: Start time in milliseconds
- end: End time in milliseconds
- tags: A dictionary of tags, used to refine the query (should include source=identify_of_source_node)
- all_tags: Flag, if set to true all tags are returned in the results, if set to False then just the timestamp & value are returned. (default true)

Get Last Metric

Retrieves the last metric value

- URI /api/v1/queryengine/<metric>/last
- Method: GET
- Params:
- <metric>: Name of the metric to retrieve
- tags: A dictionary of tags used to refine the query (should include source=identify_of_source_node)
- all_tags: Flag, if set to true all tags are returned in the results, if set to False then just the timestamp & value are returned. (default true)

Get Mean Metric

Retrieves the mean value for a given metric over the supplied duration

- URI /api/v1/queryengine/<metric>/mean
- Method: GET
- Params:
- <metric>: Name of the metric to retrieve
- tags: A dictionary of tags used to refine the query (should include source=identify_of_source_node)
- duration: The relative duration.

4.11 Analytics

The Analytics module provides an API to trigger a workload performance analysis

Get available methods

Returns the list of available methods to call from the query engine

- URI /api/v1/analytics/
- Method: GET
- Params: none

Show Heuristics

Retrieves a list of available heuristics (installed plugins) to perform on a service deployment.

- URI /api/v1/analytics/heuristics
- Method: GET
- Params: none

Perform a heuristic

Executes a heuristic on the telemetry related to a service deployment. Returns a list of Models generated by the heuristic (if any).

- URI /api/v1/analytics/heuristic
- Method: POST
- Params:
- heuristic: (str) name of the heuristic to be executed
- service_id: Id of the service to be analysed
- metadata: (dict) parameters to be given to the heuristic

4.12 Service Management Categorization

This module is responsible for the categorization of tasks received from the mapping block inside the service management based on the specific resource requirements such as network, storage, CPU or memory, as well as other metrics specified by the user. Currently, this module will not be accessible from other modules outside the service management block. All interfaces are set as private.

This module will not be implemented in IT-1.

4.13 Mapping

This module is responsible for the mapping of tasks received from the Platform Manager. This module will not be implemented in IT-1. However, the required interfaces for the future implementation are:

Get available methods

Returns the list of available methods to call on the mapping interface

- URI /api/v1/mapping/
- Method: GET
- Params: none

Submit a task

Entry point to compute a task on the Service Management block. Returns the results of the computation: information, errors or warnings.

- URI /api/v1/mapping/submit
- Method: POST
- Params: none
- Body: *task - JSON object representing a task.*

Task operation

Start, stop, restart or delete a task. Returns the result of the operation: information, errors or warnings.

- URI /api/v1/mapping/<task_id>/<options>
- Method: PUT
- Params:
  - *<task_id>: Id of the task.*
  - *<options>: type of operation {start, stop, restart, delete}.*

4.14 Allocation

This module is responsible for the allocation of available resources to the various requests, trying to meet security and privacy rules, cost models, while guaranteeing overall optimal resources usage. Currently, this module will not be accessible from other modules outside the service management block. All interfaces are set as private.

This module will not be implemented in IT-1.

4.15 QoS Providing

Check QoS

Interface to check the QoS of a specific service. The body of the request contains the user requirements represented as a JSON object. The interface returns the result of the operation: information, errors or warnings.

- URI /api/v1/qos/<service_id>
- Method: POST
- Params:
  - <service_id>: Id of the service.
- Body:
  - JSON object representing the user requirements.

4.16 Discovery

Starts broadcast

Starts broadcasting beacons (Leader)

- URI /api/v1/resource-management/discovery/broadcast

- Method: POST
- Params: none
- Body: broadcast information: *broadcast_frequency and interface_name*(JSON object)

Stops broadcast

Stops broadcasting beacons (Leader)

- URI /api/v1/resource-management/discovery/broadcast
- Method: PUT
- Params: none

Get list of found leaders

Returns a list of found leaders following a scan for mF2C beacons (Agent)

- URI /api/v1/resource-management/discovery/scan
- Method: GET
- Params: none

4.17 AC Policies

Get broadcast frequency

Retrieve the broadcast frequency to be used by the discovery component

- URI /api/v1/resource-management/policies/discovery/broadcast
- Method: GET
- Params: none

Get keepalive timeout

Retrieve the keepalive timeout

- URI /api/v1/resource-management/policies/discovery/keepalive
- Method: GET
- Params: none

4.18 Identification

Generate ID

Calculate the resource identifier using the hash function

- URI /api/v1/resource-management/identification/generate
- Method: POST
- Params: IDKey, user's email

Update ID

Change the current ID for a new one

- URI /api/v1/resource-management/identification/update
- Method: POST
- Params: IDKey, user's email

Revoke ID

Local revocation of the resource ID

- URI /api/v1/resource-management/identification/revoke
- Method: POST

- Params: none

4.19 Resource Management Categorization

Get Resource Specifications

Retrieve the resource information – Hardware, Power, Software, and Network

- URI /api/v1/resource-management/categorization/retrieve
- Method: GET
- Params: none

Set attached components, IoTs, security requirements, and some features and behaviours

User will manually provide the information about attached components, IoTs, security requirements and features and behaviour information

- URI /api/v1/resource-management/categorization/set
- Method: POST
- Params: none

Set shareable resource information

User will manually fill up the form for providing the shareable amount of resources by checking the availability of resource components

- URI /api/v1/resource-management/categorization/set
- Method: GET, POST
- Params: hardware information, software information, network information, power information, attached IoT information, attached resource component information, behaviours and features information, security requirements information
- Body: shareable resource information(JSON object)

4.20 Monitoring

The Monitoring module includes an API to load, unload and retrieve telemetry probe information.

Get available probes

Returns a list of all probes deployed on this node

- URI /v1/plugins/
- Method: GET
- Params: none

Get probe types

Retrieves a list of probes on this node for a given type, name, and version

- URI /v1/plugins/:type/:name/:version
- Method: GET
- Example: curl -L http://localhost:8181/v1/plugins/collector/mongodb/1
- Params: type, name, and/or version

Load new probe

Loads a new telemetry probe onto a node

- URI /v1/plugins/
- Method: POST
- Params: probe to be loaded

- Example: curl -X POST -F plugin=@plugin-collector-mock1 http://localhost:8181/v1/plugins

Unload new probe

Unloads a telemetry probe from a node

- URI /v1/plugins/:type/:name/:version:
- Method: DELETE
- Params: probe to be unloaded
- Example: curl -X DELETE http://localhost:8181/v1/plugins/collector/mock/1

Get a probe's configuration

Returns the configuration for a given probe

- URI /v1/plugins/:type/:name/:version/config
- Method: GET
- Params: type, name, and/or version
- Example: curl -L http://localhost:8181/v1/plugins/collector/mock/1/config

Set a probe's configuration

Returns the configuration for a given probe

- URI /v1/plugins/:type/:name/:version/config
- Method: PUT
- Params: type, name, and/or version
- Example: curl -L -X PUT http://localhost:8181/v1/plugins/collector/mock/1/config --data '{"password": "xyz"}'

4.21 AC Data Manager

The AC Data Manager component provides an API to store and retrieve data objects, and to get or set (part of) the information contained in an object. The specific functions that allow to perform these actions are defined in the classes that are registered in the Data Manager.

Store an object

Stores a new data object of the specified class in the Data Manager, optionally with the given alias.

- URI api/v1/db/<class>/:alias
- Method: POST
- Params: alias given to the object for later retrieval by name (optional)
- Body: the data contained in the object (JSON object following the resource specification corresponding to <class>)
- Example: POST http://localhost:8181/api/v1/db/device?alias=my_alias

Get an object

Gets the data object of the specified class identified by alias.

- URI api/v1/db/<class>/:alias
- Method: GET
- Params: alias given to the object Example: POST
- Example: GET http://localhost:8181/api/v1/db/agent?alias=my_alias

Update an object

Updates part of the data of the object with the specified object id (oid). This call will be mapped to the execution of the specified <setter_method> defined in <class>.

- URI api/v1/db/<class>/<setter_method>/:oid/
- Method: PUT
- Params: identifier of the object
- Body: object information to be modified (JSON object following the corresponding resource specification)
- Example: PUT http://localhost:8181/api/v1/db/agent/add_child?oid=1234

Get information from an object

Gets part of the data of the object with the specified object id (oid). This call will be mapped to the execution of the specified <getter_method> defined in <class>.

- URI api/v1/db/<class>/<getter_method>/:oid/
- Method: GET
- Params: identifier of the object
- Example: GET http://localhost:8181/api/v1/db/agent/get_static_info?oid=1234

New replica

Replicates the object with the specified object id (oid) in the specified location (backend).

- URI api/v1/db/<class>/replica/:oid/:backend
- Method: PUT
- Params: identifier of the object and destination backend
- Example: PUT http://localhost:8181/api/v1/db/agent/replica?oid=1234/backend=local

Delete an object

Makes the object with the specified alias inaccessible.

- URI api/v1/db/<class>/:alias/
- Method: DELETE
- Params: alias of the object
- Example: DELETE http://localhost:8181/api/v1/db/agent?oid=1234

4.22 Profiling

The Profiling sub component is responsible for managing the user's profile data.

Get available methods

This method returns the list of all available methods to call on the user management interface. This includes the profiling, assessment and sharing model methods.

- URI /api/v1/ user-management /
- Method: GET
- Params: none

Get Profile properties from user

Returns the user's profile (JSON object)

- URI /api/v1/user-management/profiling/:user_id
- Method: GET
- Params:
    - user_id: user ID

Initializes the user's profile – User registration

Initializes the user's profile and returns a JSON object with all the profile information.

- URI /api/v1/user-management/profiling
- Method: POST
- Params: none
- Body: user's information: *email*, *id_key* … (JSON object)

Updates the user's profile

Updates the user's profile and returns a JSON object with all the profile information.

- URI /api/v1/user-management/profiling
- Method: PUT
- Params: none
- Body: user's information: *email*, *service_consumer*, *resource_contributor* … (JSON object)

Deletes the user's profile

Deletes the user's profile and returns a JSON Object with the result of the operation.

- URI /api/v1/user-management/profiling
- Method: DELETE
- Params: none
- Body: user id

4.23 Assessment

The User Management Assessment sub component is responsible for checking that the mF2C applications act according to the defined sharing model and profile properties.

Get Assessment process status

Returns the status of the assessment process (JSON object)

- URI /api/v1/user-management/assesment-process
- Method: GET
- Params: none

Start / Stop Assessment process

Starts / stops the assessment process

- URI /api/v1/user-management/assesment-process
- Method: PUT
- Params: none
- Body: operation value ('start', 'stop')

4.24 Sharing Model

The Sharing Model sub component is responsible for managing the device's resources shared by the user.

Get Sharing Model values from user/device

Returns the collection of resources shared by the user/device (JSON object)

- URI /api/v1/user-management/sharingmodel/:user_id
- Method: GET
- Params:
    - user_id: user ID

Initializes the Sharing Model values

Initializes the Sharing Model properties and returns the JSON object representation of the resulting sharing model

- URI /api/v1/user-management/sharingmodel
- Method: POST
- Params: none
- Body: user ID and collection of resources to be shared (JSON object)

Updates the Sharing Model values

Updates the Sharing Model properties and returns the JSON object representation of the resulting sharing model

- URI /api/v1/user-management/sharingmodel
- Method: PUT
- Params: none
- Body: user ID and collection of resources to be shared (JSON object)

Deletes the Sharing Model values

Resets the Sharing Model properties and returns a JSON Object with the result of the operation.

- URI /api/v1/user-management/sharingmodel
- Method: DELETE
- Params: none
- Body: user ID

References

[1] “CIMI standard interface specification,” [Online]. Available: http://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0.pdf.

[2] “Clojure,” [Online]. Available: https://clojure.org/.

[3] “Ring,” [Online]. Available: https://github.com/ring-clojure/ring.

[4] “Aleph,” [Online]. Available: http://aleph.io/.

[5] “REST Security Cheat Sheet,” [Online]. Available: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet.

[6] “HTTP status codes,” [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

[7] “RFC,” [Online]. Available: http://www.faqs.org/rfcs/rfc2396.html.