Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D.6.4.3: FI-WARE User and Programmers Guide

Project acronym: FI-WARE
Project full title: Future Internet Core Platform
Contract No.: 285248
Strategic Objective: FI.ICT-2011.1.7 Technology foundation: Future Internet Core Platform
Project Document Number: ICT-2011-FI-285248-WP6-D.6.4.3
Project Document Date: 2014-07-09
Deliverable Type and Security: Public
Author: FI-WARE Consortium
Contributors: FI-WARE Consortium
1.1 Executive Summary

This document contains the User and Programmers guide of the Data/Context Management technical chapter in FI-WARE. The purpose of this manual is twofold:

- Provide the users with insights in the functionality of each GE and how to use it
- Provide the developers with the right pointers to develop using the Open Specifications

This document refers to release 3 of software of the Data/Context Management chapter GE implementations.
1.2 About This Document

This document comes along with the Software implementation of components, each release of the document being referred to the corresponding Software release (as per D.x.3), to provide documentation of the features offered by the components and interfaces to users/adopters. Moreover, it explains the way they can be exploited in their developments.

1.3 Intended Audience

The document targets users as well as programmers of FI-WARE Generic Enablers.

1.4 Chapter Context

FI-WARE will enable smarter, more customized/personalized and context-aware applications and services by the means of a set of assets able to gather, exchange, process and analyze massive data in a fast and efficient way. Nowadays, several well-known free Internet services are based on business models that exploit massive data provided by end users. This data is exploited in advertising or offered to 3rd parties so that they can build innovative applications. Twitter, Facebook, Amazon, Google and many others are examples of this.

The "Data/Context Management" FI-WARE chapter aims at providing outperforming and platform-like GEs that ease development and provision of innovative Applications that require management, processing and exploitation of context information as well as data streams in real-time and at massive scale. Combined with enablers coming from the Applications/Services Ecosystem and Delivery chapters, application providers will be able to build innovative business models such as the ones described above and beyond.

FI-WARE Data/Context Management GEs enables to:

- Record, subscribe for being notified about and query for context information coming from different sources.
- Model changes in context as events that can be processed to detect complex situations that will lead to generation of actions or the generation of new context information (therefore, also treatable as events).
- Processing large amounts of context information in an aggregated way, using map&reduce techniques, in order to generate knowledge that may also lead to execution of actions and/or creation of new context information.
- Process data streams (particularly, multimedia video streams) coming from different sources in order to generate new data streams as well as context information that can be further exploited.
- Process metadata that may be linked to context information, using standard semantic support technologies.
- Manage some context information, such Location information, in a standardized way.

A cornerstone concept within this chapter is the structural definition of Data Elements enclosing its "Data Type", a number of "Data Element attributes" (which enclose the following: Name, Type, Value) and, optionally, a set of "Metadata Elements" (which have also in turn Data-like attributes: Name, Type, Value). However, this precise definition remains unbound to any specific type of representation and is able to represent "Context Elements" and "Events" as "Data Element" structures. More comprehensive information is available at Fi-WARE Data/Context Chapter vision.

"Data" in Fi-WARE refers to information that is produced, generated, collected or observed that may be relevant for processing, carrying out further analysis and knowledge extraction. A cornerstone concept in Fi-WARE is that data elements are not bound to a specific format representation.

The following diagram shows the main components (Generic Enablers) that comprise the third release of Fi-WARE Data/Context chapter architecture.
More information about the Data Chapter and FI-WARE in general can be found within the following pages:

- **Data/Context Management Architecture**
- **Materializing Data/Context Management in FI-WARE**

## 1.5 Structure of this Document

The document is generated out of a set of documents provided in the public FI-WARE wiki. For the current version of the documents, please visit the public wiki at [http://wiki.fi-ware.eu/](http://wiki.fi-ware.eu/)

The following resources were used to generate this document:

**D.6.4.3 User and Programmers Guide front page**
- [Publish/Subscribe Broker - Orion Context Broker - User and Programmers Guide](#)
- [Publish/Subscribe GE - Context Awareness Platform - User and Programmer Guide](#)
- [Publish/Subscribe Semantic Extension - User and Programmer Guide](#)
- [CEP GE - IBM Proactive Technology Online User and Programmer Guide](#)
- [BigData Analysis - User and Programmer Guide](#)
- [Compressed Domain Video Analysis - User and Programmers Guide](#)
- [Unstructured Data Analysis - User and Programmer Guide](#)
- [Metadata Preprocessing - User and Programmers Guide](#)
- [LOCS - User and Programmers Guide](#)
- [Query Broker - User and Programmer Guide](#)
- [Semantic Application Support - Users and Programmers Guide](#)
- [StreamOriented - Users and Programmers Guide](#)

## 1.6 Typographical Conventions

Starting with October 2012 the FI-WARE project improved the quality and streamlined the submission process for deliverables, generated out of our wikis. The project is currently working on the migration of as many deliverables as possible towards the new system.
This document is rendered with semi-automatic scripts out of a MediaWiki system operated by the FIWARE consortium.

1.6.1 Links within this document

The links within this document point towards the wiki where the content was rendered from. You can browse these links in order to find the "current" status of the particular content.

Due to technical reasons part of the links contained in the deliverables generated from wiki pages cannot be rendered to fully working links. This happens for instance when a wiki page references a section within the same wiki page (but there are other cases). In such scenarios we preserve a link for readability purposes but this points to an explanatory page, not the original target page.

In such cases where you find links that do not actually point to the original location, we encourage you to visit the source pages to get all the source information in its original form. Most of the links are however correct and this impacts a small fraction of those in our deliverables.

1.6.2 Figures

Figures are mainly inserted within the wiki as the following one:

\[
[[\text{Image:...}|\text{size}|\text{alignment}|\text{Caption}]]
\]

Only if the wiki-page uses this format, the related caption is applied on the printed document. As currently this format is not used consistently within the wiki, please understand that the rendered pages have different caption layouts and different caption formats in general. Due to technical reasons the caption can't be numbered automatically.

1.6.3 Sample software code

Sample API-calls may be inserted like the following one.

\[
\text{http://[SERVER_URL]?filter=\text{name:Simth*}&index=20&limit=10}
\]

1.7 Acknowledgements

The current document has been elaborated using a number of collaborative tools, with the participation of Working Package Leaders and Architects as well as those partners in their teams they have decided to involve.
1.8  Keyword list


1.9  Changes History

<table>
<thead>
<tr>
<th>Release</th>
<th>Major changes description</th>
<th>Date</th>
<th>Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>First Version</td>
<td>2014-07-09</td>
<td>TID</td>
</tr>
</tbody>
</table>

1.10  Table of Contents

1.1  Executive Summary
1.2  About This Document
1.3  Intended Audience
1.4  Chapter Context
1.5  Structure of this Document
1.6  Typographical Conventions
1.7  Acknowledgements
1.8  Keyword list
1.9  Changes History
1.10  Table of Contents

2  Publish/Subscribe Broker - Orion Context Broker - User and Programmers Guide
2.1  Introduction
2.2  User Guide
2.3  Programmers Guide

3  Publish/Subscribe GE - Context Awareness Platform - User and Programmer Guide
3.1  Introduction
3.2  User guide
3.3  Programmer's guide

4  Publish/Subscribe Semantic Extension - User and Programmer Guide
4.1  Introduction
4.2  User Guide
### Future Internet Core Platform

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Programmer Guide</td>
<td>318</td>
</tr>
<tr>
<td>5</td>
<td>CEP GE - IBM Proactive Technology Online User and Programmer Guide</td>
<td>328</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>328</td>
</tr>
<tr>
<td>5.2</td>
<td>User Guide</td>
<td>328</td>
</tr>
<tr>
<td>5.3</td>
<td>Programmer Guide</td>
<td>328</td>
</tr>
<tr>
<td>6</td>
<td>BigData Analysis - User and Programmer Guide</td>
<td>329</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>329</td>
</tr>
<tr>
<td>6.2</td>
<td>User guide</td>
<td>329</td>
</tr>
<tr>
<td>6.3</td>
<td>Programmer guide</td>
<td>347</td>
</tr>
<tr>
<td>7</td>
<td>Compressed Domain Video Analysis - User and Programmers Guide</td>
<td>373</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>373</td>
</tr>
<tr>
<td>7.2</td>
<td>User Guide</td>
<td>373</td>
</tr>
<tr>
<td>7.3</td>
<td>Programmers Guide</td>
<td>394</td>
</tr>
<tr>
<td>8</td>
<td>Unstructured Data Analysis - User and Programmer Guide</td>
<td>400</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>400</td>
</tr>
<tr>
<td>8.2</td>
<td>User guide</td>
<td>400</td>
</tr>
<tr>
<td>8.3</td>
<td>Programmers Guide</td>
<td>400</td>
</tr>
<tr>
<td>9</td>
<td>Metadata Preprocessing - User and Programmers Guide</td>
<td>405</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>405</td>
</tr>
<tr>
<td>9.2</td>
<td>User Guide</td>
<td>405</td>
</tr>
<tr>
<td>9.3</td>
<td>Programmers Guide</td>
<td>405</td>
</tr>
<tr>
<td>10</td>
<td>LOCS - User and Programmers Guide</td>
<td>415</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>415</td>
</tr>
<tr>
<td>10.2</td>
<td>User Guide</td>
<td>415</td>
</tr>
<tr>
<td>10.3</td>
<td>Programmer guide</td>
<td>421</td>
</tr>
<tr>
<td>11</td>
<td>Query Broker - User and Programmer Guide</td>
<td>433</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction</td>
<td>433</td>
</tr>
<tr>
<td>11.2</td>
<td>User guide</td>
<td>433</td>
</tr>
<tr>
<td>11.3</td>
<td>Programmer guide</td>
<td>434</td>
</tr>
<tr>
<td>11.4</td>
<td>References</td>
<td>451</td>
</tr>
<tr>
<td>12</td>
<td>Semantic Application Support - Users and Programmers Guide</td>
<td>452</td>
</tr>
<tr>
<td>12.1</td>
<td>Introduction</td>
<td>452</td>
</tr>
<tr>
<td>12.2</td>
<td>User guide</td>
<td>455</td>
</tr>
</tbody>
</table>

D.6.4.3: FI-WARE Users and Programmers Guide
12.3 Programmers Guide........................................................................................................462

13 StreamOriented - Users and Programmers Guide.............................................................472
  13.1 Introduction ..................................................................................................................472
  13.2 User Guide ..................................................................................................................472
  13.3 Programmer Guide ......................................................................................................472
2 Publish/Subscribe Broker - Orion Context Broker - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

2.1 Introduction

Welcome to the Orion Context Broker User and Programmers Guide!

The Orion Context Broker is an NGSI9/10 server implementation to manage context information and context information availability. Using the Orion Context Broker, you are able to register context elements and manage them through updates and queries. In addition, you can subscribe to context information so when some conditions occur (an interval of time has passed or the context elements have changed) you receive a notification. These usage scenarios and the Orion Context Broker features are described in this document.

This User and Programmers Guide covers Orion Context Broker versions since 0.5.0 (corresponding to FI-WARE release 2.3.3). Please pay attention to the "Release Note" fragments along the document, as they contain important information regarding particular versions/releases, specially to the note on versions previous to 0.9.0 in the introduction to programmers guide.

Any feedback on this document is highly welcome, including bug reports, typos or stuff you think should be included but isn't. Please send it to the "Contact Person" email that appears in the Catalogue page for this GEI. Thanks in advance!

2.2 User Guide

Orion Context Broker runs as a backend service daemon. Thus, it doesn't have any Graphical User Interface (GUI). It is accessed through its REST API, described in the Programmers Guide section.

2.3 Programmers Guide

2.3.1 Introduction to Programmers Guide

This guide adopts a practical approach that we hope will help our readers to get familiar with the Orion Context Broker and have some fun in the process :).

The first two sections on context management using NGSI10 and context availability management using NGSI9 are the main ones. They describe the basic context broker functionality, both for context management (information about entities, such as the temperature of a car) or context availability management (information not about entities themselves, but about the providers of that information). Some remarks to take into account to use this stuff:
• Context management and context availability management are independent functionalities (corresponding to different parts of the NGSI interface, NGSI10 and NGSI9 respectively), so you can use the broker for one purpose, the other purpose, or both of them.

• Note that each main section is divided in two sub-sections: the first one on standard operations and the second on convenience operations. In fact, each sub-section is an independent tutorial (for a total of 4 tutorials summing up both sections) that can be done in a step-by-step manner, just copy-pasting the commands from this document. Note that since release 0.9.0, Orion Context Broker supports a JSON rendering (preferred by many developers, as it is simpler than using XML) so we provide two versions of each example: one in XML and one in JSON.

• Before starting one tutorial (or if you get lost in the middle and want to go back to the beginning :) restart Orion Context Broker as described in starting the broker for the tutorials.

• It is recommended to do the tutorial on standard operations first, then do the tutorial on convenience operations (some explanations and concepts described in the former are needed for the latter).

The third section describes some advanced topics of the Orion Context Broker. This is optional material and you don’t actually need it to build applications using the broker. Although you will find links from tutorials to the advanced topics section, our recommendation is that you first get familiar with the basic material before starting with the advanced topics.

The fourth section describes the FI-LAB context management platform. This information is specially useful for FI-LAB users, if you plan to use Orion Context Broker independently of FI-LAB, you can skip this section.

Finally, you have a section with additional information and resources, including code snipes contributed by users.

It is recommended that you get familiar with the theoretical concepts on which the NGSI model is based before starting. E.g. entities, attributes, etc. Have a look at the FI-WARE documentation about this, e.g. this public presentation or this public video.

Before starting, let’s introduce the example case that we will consider in the tutorials and how to run and interact with Orion Context Broker.

Important for users of releases previous to 0.10.0 (FI-WARE 3.3.2): this version solves some issues recently detected in the JSON encoding. In this sense, you will note some slight changes in some JSON payloads shown along this document compared to previous versions of this manual. Depending on how your context-based application generates and process the JSON, you may need to do some small changes. In particular, the way context broker behaves from 0.10.0 on follows this:

• Only "value" is a valid name for the JSON field used to specify an attribute value (previous versions of the documentation sometimes refer to "contextValues"). Not fully solved in 0.10.0, but solved in 0.11.0.

• JSON vectors ("[ ... ]") are enforced for the fields which value is a list of elements (e.g. "contextElements" in updateContextRequest). Thus, the use of a JSON object ("{ ... }") as value instead of a vector in the case of mono-element lists is not allowed anymore.
Important for users of releases previous to 0.9.0 (FI-WARE 3.2.3):

- You have probably noticed that the structure of this manual has changed, as previous version didn't have separated and independent sections for NGSI9 and NGSI10. The change has been motivated due to now NGSI9 and NGSI10 are completely independent functionalities. For example, you don’t need to do a registerContext request (NGSI9) to "create" and entity before doing an updateContext request (NGSI10) on it. We decided to make both independent after the feedback we get from developers during the Campus Party London and Santander 2013 hackathons, to make things easier in many context broker usage cases. Thus, the structure of the manual has changed to reflect that.

- In release 0.9.0 we have paid a lot of attention to improve our alignment with FI-WARE NGSI9/10 XSD in XML payloads in requests and responses (the XSD specification is included in this file). In this sense, you will note some slight changes in the XML payloads shown along this document compared to previous versions of this manual. Depending on how your context-based application generates and process the XML, you may need to do some small changes. In particular, the way context broker behaves from 0.9.0 on follows this:
  - <duration> precedes <registrationId> in RegisterContextResponse
  - <statusCode> has been renamed to <responseCode> in notifyContextResponse and notifyContextAvailabilityResponse
  - <subscriptionId> precedes <duration> in NGSI9 and NGSI10 subscriptions
  - Empty <throttling> is not allowed
  - Action type in updateContextRequest has to use uppercase (UPDATE, APPEND or DELETE)

2.3.1.1 Example Case
Let's assume we have a building with several rooms and that we want to use Orion Context Broker to manage its context information. The rooms are Room1, Room2, Room3 and Room4 and each room has two sensors: temperature and (atmospheric) pressure (except Room4, that only has a pressure sensor). In addition, let's consider that we have two cars (Car1 and Car2) with sensors able to measure speed and location (in GPS sense).
Most of the time we will use Room1 and Room2 in the tutorials. Room3, Room4, Car1 and Car2 will be used only in the section regarding context availability subscriptions.

The Orion Context Broker interacts with context producer applications (which provide sensor information) and a context consumer application (which processes that information, e.g. to show it in a graphical user interface). We will play the role of both kinds of applications in the tutorials.

2.3.1.2 Starting the broker for the tutorials

Before starting, you need to install the broker as described in the Installation and Administration Guide.

The tutorials assume that you don’t have any previous content in the Orion Context Broker database. In order to do so, follow the delete database procedure.

To start the broker (as root or using the sudo command):

```
/etc/init.d/contextBroker start
```

To restart the broker (as root or using the sudo command):

```
/etc/init.d/contextBroker restart
```

2.3.1.3 Starting accumulator server for the tutorials

Some part of the tutorial (the ones related with subscriptions and notifications) require a some process to play the role of the consumer application able to receive notifications. To that end, the contextBroker-test package (see in the administrator manual how to install) includes the
/usr/share/contextBroker/tests/accumulator-server.py. It is a very simple "dummy" application that just listens to a given URL (let's use localhost:1028/accumulate) and prints whatever it gets in the terminal window where it is executed. Run it using the following command:

```bash
# cd /usr/share/contextBroker/test
# ./accumulator-server.py 1028 /accumulate ::1 on
```

A word of advice on accumulator-server.py: you will probably see traces about "Broken Pipe" errors. Don't worry: this is due to the way in which the underlying HTTP library in Python deals with HTTP connection ends in some cases and it is completely harmless.

### 2.3.1.4 Issuing commands to the broker

To issue requests to the broker, we use the **curl** command line tool. We have chosen curl because it is almost ubiquitous in any GNU/Linux system and simplifies including examples in this document that can easily be copied and pasted. Of course, it is not mandatory to use curl, you can use any REST client tool instead (e.g. RESTClient). Indeed, in a real case, you will probably interact with the Orion Context Broker using a programming language library implementing the REST client part of your application. The basic patterns for all the curl examples in this document are the following (XML case):

- For POST:

  ```bash
  curl localhost:1026/<operation_url> -s -S [headers] -d @- <<EOF
  [payload]
  EOF
  ```

- For PUT:

  ```bash
  curl localhost:1026/<operation_url> -s -S [headers] -X PUT -d @- <<EOF
  [payload]
  EOF
  ```

- For GET:

  ```bash
  curl localhost:1026/<operation_url> -s -S [headers]
  ```

- For DELETE:

  ```bash
  curl localhost:1026/<operation_url> -s -S [headers] -X DELETE
  ```
Regarding [headers] you have to include the following ones:

- Accept header to specify which payload format (either XML or JSON) you want to receive in the response (if you don't specify a Accept header, the default is XML):

  ```sh
curl ... --header 'Accept: application/json' ...
curl ... --header 'Accept: application/xml' ...
```

- Only in the case of using payload in the request (i.e. POST or PUT), you have to use Context-Type header to specify the format (either XML or JSON):

  ```sh
curl ... --header 'Content-Type: application/json' ...
curl ... --header 'Content-Type: application/xml' ...
```

**Release note 0.9.0 (FI-WARE 3.2.3) or before:** versions previous to 0.9.0 doesn't support JSON at all. Version 0.9.0 support JSON for standard operations, but not for convenience operations.

Some additional remarks:

- We are using multi-line shell commands to provide the input to curl, using EOF to mark the beginning and the end of the multi-line block (*here-documents*). An alternative to this is to put the XML in a file (e.g. payload.xml), then use "--data-binary @payload.xml". In some cases (GET and DELETE) we omit "-d @-" as they don't use payload.

- In our examples we assume that the broker is listening on port 1026. Adjust this in the curl command line if you are using a different setting.

- In order to pretty-print XML in responses, you can use xmllint (examples along with tutorial are using this style):

  ```sh
  (curl ... | xmllint --format -) <<EOF
  ...
  EOF
  ```

- In order to pretty-print JSON in responses, you can use Python with msjson.tool (examples along with tutorial are using this style):

  ```sh
  (curl ... | python -mjson.tool) <<EOF
  ...
  EOF
  ```
Check that curl (and xmllint, if you plan to use it for pretty-printing) are installed in your system using:

```
# which curl
# which xmllint
```

2.3.1.5 **Text colour convention**
This document assumes the following convention for text colour in XML fragments:
- **Blue**: request messages
- **Green**: response messages
- **Purple**: notification messages

2.3.2 **Context management using NGSI10**

2.3.2.1 **Tutorial on NGSI10 standard operations**
This section describes the different standard NGSI10 operations that the Orion Context Broker supports, showing examples of requests and responses. We use the term "standard" as they are directly derived from the OMA NGSI specification, to distinguish them from the other family of operations ("convenience") which has been defined by the FI-WARE project to ease the usage of NGSI implementations (see the section on additional information later in this manual).

Don't forget to restart the broker before starting this tutorial as described previously in this document.

At the end of this section, you will have the basic knowledge to create applications (both context producers and consumers) using Orion Context Broker with NGSI10 standard operations:

- updateContext
- queryContext
- subscribeContext
- updateContextSubscription
- unsubscribeContext

2.3.2.1.1 **Entity Creation**
Orion Context Broker will start in an empty state, so first of all we need to make it aware of the existence of certain entities. In particular, we are going to "create" Room1 and Room2 entities, each one with two attributes (temperature and pressure). We do this using the updateContext operation with APPEND action type (the other main action type, UPDATE, will be discussed in a next section).
First, we are going to create Room1. Let’s assume that at entity creation time temperature and pressure of Room1 are 23 ºC and 720 mmHg respectively.

XML

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue>23</contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <contextValue>720</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @python -m jjson.tool) <<EOF
{
    "contextElements": [
    {
        "type": "Room",
        "isPattern": "false",
        "id": "Room1",
        "attributes": [
        {
            "name": "temperature",
            "type": "centigrade",
            "value": "23"
        },
        {
            "name": "pressure",
            "type": "mmHg",
            "value": "720"
        }
    ]
    },
    
},
    "updateAction": "APPEND"
}
EOF
The updateContext request payload contains a list of contextElement elements. Each contextElement is associated to an entity (whose identification is provided in the entityId element, in this case we provide the identification for Room1) and contains a list of contextAttribute elements ('attributes' for short, in JSON). Each contextAttribute provides the value for a given attribute (identified by name and type) of the entity. Apart from the list of contextElement elements, the payload includes also an updateAction element. We use APPEND, which means that we want to add new information.

**Release Note (any version):** Orion Context Broker doesn't perform any checking on types (e.g. it doesn't check that when a context producer application updates the value of the temperature, this value is formatted as a centigrade degree measure like "25.5" or "-40.23" and not something like "very cool" or "-275.2").

Upon receipt of this request, the broker will create the entity in its internal database, set the values for its attributes and will response with the following:

```xml
<?xml version="1.0"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue/>
          </contextAttribute>
          <contextAttribute>
            <name>pressure</name>
            <type>mmHg</type>
            <contextValue/>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```
```json
{  
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "temperature",
            "type": "centigrade",
            "value": ""
          },
          {
            "name": "pressure",
            "type": "mmHg",
            "value": ""
          }
        ],
        "id": "Room1"
      }
    }
  ]
}
```
As you can see, it follows the same structure as the request, just to acknowledge that the request was correctly processed for these context elements. You probably wonder why contextValue elements are empty in this case, but actually you don't need the values in the response because you were the one to provide them in the request.

Next, let's create Room2 in a similar way (in this case, setting temperature and pressure to 21 ºC and 711 mmHg respectively).

**XML**

```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room2</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

<type>centigrade</type>
<contextValue>21</contextValue>
</contextAttribute>
<contextAttribute>
    <name>pressure</name>
    <type>mmHg</type>
    <contextValue>711</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>APPEND</updateAction>
</updateContextRequest>

EOF

---

**JSON** *(since release 0.9.0)*

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool ) <<EOF
{
  "contextElements": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room2",
      "attributes": [
        {
          "name": "temperature",
          "type": "centigrade",
        
```
The response to this request is:

```
<?xml version="1.0"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room2</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <value>"21"</value>
          </contextAttribute>
          <contextAttribute>
            <name>pressure</name>
            <value>"711"</value>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      "updateAction": "APPEND"
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```

The response to this request is:

```
{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [{
          "name": "temperature",
          "type": "centigrade",
          "value": ""
        }]
      }
    }
  ]
}  
```

**JSON (since release 0.9.0)**
Apart from simple values (i.e. strings) for attribute values, you can also use complex structures or custom metadata. These are advance topics, described in this section and this other, respectively.

2.3.2.1.2 Query Context operation
Now let's play the role of a consumer application, wanting to access the context information stored by Orion Context Broker to do something interesting with it (e.g. show a graph with the room temperature in a graphical user interface). The NGSI10 queryContext request is used in this case, e.g. to get context information for Room1:
XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList/>
</queryContextRequest>
EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @ | python -mjson.tool) <<EOF
{"entities": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1"
  }
]
}
EOF
The response includes all the attributes belonging to Room1 and we can check that temperature and pressure have the values that we set at entity creation with updateContext (23ºC and 720 mmHg).

<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>23</contextValue>
          </contextAttribute>
          <contextAttribute>
            <name>pressure</name>
            <type>mmHg</type>
            <contextValue>720</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
{
    "contextResponses": [
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": "23"
                    },
                    {
                        "name": "pressure",
                        "type": "mmHg",
                        "value": "720"
                    }
                ],
                "id": "Room1",
                "isPattern": "false",
                "type": "Room"
            }
        },
        "statusCode": {
            "code": "200",
            "reasonPhrase": "OK"
        }
    ]
}
If you use an empty attributeList element in the request ('attributes' for short, in JSON), the response will include all the attributes of the entity. If you include an actual list of attributes (e.g. temperature) only that are retrieved, as shown in the following request:

**XML**

```
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
</queryContextRequest>
EOF
```

**JSON (since release 0.9.0)**

```
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
```
which response is as follows:

```xml
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>23</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```
<contextAttributeList>
</contextElement>
<statusCode>
 <code>200</code>
 <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</queryContextResponse>

JSON (since release 0.9.0)

{
 "contextResponses": [
 {
 "contextElement": {
 "attributes": [
 {
 "name": "temperature",
 "type": "centigrade",
 "value": "23"
 }
 ],
 "id": "Room1",
 "isPattern": "false",
 "type": "Room"
 },
 "statusCode": {
 "code": "200",
 "reasonPhrase": "OK"
 }
 }
]
Moreover, a powerful feature of Orion Context Broker is that you can use a regular expression for the entity ID. For example, you can query entities which ID starts with "Room" using the regex "Room.*". In this case, you have to set isPattern to "true" as shown below:

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
    <entityId type="Room" isPattern="false">
      <id>Room2</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
</queryContextRequest>
EOF

JSON (since release 0.9.0)
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF

{
    "entities": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room1"
        },
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room2"
        }
    ],
    "attributes": ["temperature"

EOF

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF

<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
    <entityIdList>
        <entityId type="Room" isPattern="true">

D.6.4.3: FI-WARE Users and Programmers Guide 32
<id>Room.*</id>
</entityId>
</entityIdList>
<attributeList>
  <attribute>temperature</attribute>
</attributeList>
</queryContextRequest>
EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
  {
    "type": "Room",
    "isPattern": "true",
    "id": "Room.*"
  }
  ],
  "attributes" : [
    "temperature"
  ]
}
EOF

Both produce the same response:

XML
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>23</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
    <statusCode>
      <code>200</code>
      <reasonPhrase>OK</reasonPhrase>
    </statusCode>
  </contextElementResponse>
  <contextElementResponse>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room2</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementResponse>
</queryContextResponse>
<type>centigrade</type>
<contextValue>21</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</queryContextResponse>

---

**JSON (since release 0.9.0)**

```json
{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "temperature",
            "type": "centigrade",
            "value": "23"
          }
        ],
        "id": "Room1",
        "isPattern": "false",
        "type": "Room"
      }
    }
  ]
}
```
Finally, note that you will get an error in case you try to query a non-existing entity or attribute, as shown in the following cases below:

XML
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room5</id>
    </entityId>
  </entityIdList>
  <attributeList/>
</queryContextRequest>
EOF

---

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room5"
    }
  ]
}
EOF

---

XML
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>humidity</attribute>
  </attributeList>
</queryContextRequest>
EOF

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room1"
    }
  ],
  "attributes": [

Both requests will produce the same error response:

**XML**

```xml
<?xml version="1.0"?>
<queryContextResponse>
  <errorCode>
    <code>404</code>
    <reasonPhrase>No context elements found</reasonPhrase>
  </errorCode>
</queryContextResponse>
```

**JSON (since release 0.9.0)**

```json
{
  "errorCode": {
    "code": "404",
    "reasonPhrase": "No context elements found"
  }
}
```

Additional comments:

- You can also use geographical scopes in your queries. This is an advance topic, described in this section.
• Note that by default only 20 entities are returned (which is fine for this tutorial, but probably not for a real utilization scenario). In order to change this behaviour, see the section on pagination in this manual.

2.3.2.1.3 Update context elements
You can update the value of entities attributes using the updateContext operation with UPDATE action type. The basic rule to take into account with updateContext is that APPEND creates new context elements, while UPDATE updates already existing context elements (however, current Orion Context Broker version interprets APPEND as UPDATE if the entity already exists).

Now we will play the role of a context producer application, i.e. a source of context information. Let’s assume that this application in a given moment wants to set the temperature and pressure of Room1 to 26.5 ºC and 763 mmHg respectively, so it issues the following request:

XML

(curl localhost:1026/NGSI10/updateContext \s -S --header 'Content-Type: application/xml' \-d @- | xmllint \-f) \<&EOF

<?xml version="1.0" encoding="UTF-8"?><updateContextRequest>
 <contextElementList>
  <contextElement>
   <entityId type="Room" isPattern="false">
    <id>Room1</id>
   </entityId>
   <contextAttributeList>
    <contextAttribute>
     <name>temperature</name>
     <type>centigrade</type>
     <contextValue>26.5</contextValue>
    </contextAttribute>
    <contextAttribute>
     <name>pressure</name>
     <type>mmHg</type>
     <contextValue>763</contextValue>
    </contextAttribute>
   </contextAttributeList>
  </contextElement>
 </contextElementList>
</updateContextRequest>
Future Internet Core Platform

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF

{
   "contextElements": [

   {
      "type": "Room",
      "isPattern": "false",
      "id": "Room1",
      "attributes": [

      {
         "name": "temperature",
         "type": "centigrade",
         "value": "26.5"
      },

      {
         "name": "pressure",
         "type": "mmHg",
         "value": "763"
      }
     
   }]

} EOF
As you can see, the structure of the request is exactly the same we used for `updateContext with APPEND for creating entities`, except we use `UPDATE` now as action type.

Upon receipt of this request, the broker will update the values for the entity attributes in its internal database and will response with the following:

```xml
<?xml version="1.0"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue/>
          </contextAttribute>
          <contextAttribute>
            <name>pressure</name>
            <type>mmHg</type>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```
<contextValue/>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</updateContextResponse>

JSON (since release 0.9.0)

{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "temperature",
            "type": "centigrade",
            "value": ""
          },
          {
            "name": "pressure",
            "type": "mmHg",
            "value": ""
          }
        ]
      }
    }
  ]
}
Again, the structure of the response is exactly the same we used for `updateContext` with APPEND for creating entities.

The `updateContext` operation is quite flexible as it allows you to update as many entities and attributes as you want: it is just a matter of which contextElements you include in the list. You could even update the whole database of Orion Context Broker (maybe including thousands of entities/attributes) in just one `updateContext` operation (at least in theory).

To illustrate this flexibility, we will show how to update Room2 in two separated `updateContext` request (setting its temperature to 27.4 °C and its pressure to 755 mmHg), each one targeting just one attribute. This also illustrates that you don’t need to include all the attributes of an entity in the `updateContext`, just the ones you want to update (the other attributes maintain their current value).

XML

```xml
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
```
```
<id>Room2</id>
</entityId>
<contextAttributeList>
<contextAttribute>
    <name>temperature</name>
    <type>centigrade</type>
    <contextValue>27.4</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>UPDATE</updateAction>
</updateContextRequest>
EOF

---

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
    "contextElements": [
    {
        "type": "Room",
        "isPattern": "false",
        "id": "Room2",
        "attributes": [
        {
            "name": "temperature",
            "type": "centigrade",

EOF
"value": "27.4"
}
]
}
)
",
"updateAction": "UPDATE"
}
EOF

XML

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
<contextElementList>
<contextElement>
    <entityId type="Room" isPattern="false">
        <id>Room2</id>
    </entityId>
    <contextAttributeList>
        <contextAttribute>
            <name>pressure</name>
            <type>mmHg</type>
            <contextValue>755</contextValue>
        </contextAttribute>
    </contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>UPDATE</updateAction>
EOF
The responses for these requests are respectively:

XML
<xml version="1.0"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room2</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue/>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
    <statusCode>
      <code>200</code>
      <reasonPhrase>OK</reasonPhrase>
    </statusCode>
  </contextResponseList>
</updateContextResponse>

---

JSON (since release 0.9.0)

```json
{
   "contextResponses": [
      {
```

D.6.4.3: FI-WARE Users and Programmers Guide 48
"contextElement": {  
  "attributes": [  
    {  
      "name": "temperature",  
      "type": "centigrade",  
      "value": ""  
    }  
  ],  
  "id": "Room2",  
  "isPattern": "false",  
  "type": "Room"  
},  
"statusCode": {  
  "code": "200",  
  "reasonPhrase": "OK"  
}  
]  
}  

---  

XML

```xml
<?xml version="1.0"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room2</id>
```
<entityId>
<contextAttributeList>
<contextAttribute>
  <name>pressure</name>
  <type>mmHg</type>
  <contextValue/>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</updateContextResponse>

JSON (since release 0.9.0)

{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "pressure",
            "type": "mmHg",
            "value": ""
          }
        ]
      }
    }
  ]
}
Now, you can use queryContext operation as previously described to check that Room1 and Room2 attributes has been actually updated.

Apart from simple values (i.e. strings) for attribute values, you can also use complex structures. This is an advance topic, described in this section.

2.3.2.1.4  Context subscriptions
The NGSI10 operations you know up to now (updateContext and queryContext) are the basic building blocks for synchronous context producer and context consumer applications. However, Orion Context Broker has another powerful feature that you can take advantage of: the ability to subscribe to context information so when "something" happens (we will explain the different cases for that "something") your application will get an asynchronous notification. In that way, you don't need to continuously repeat queryContext requests (i.e. polling), the Orion Context Broker will let you know the information when it comes.

Before starting to play with feature, start the accumulator server to capture notifications.

Actually, there are two kinds of subscribeContext: ONTIMEINTERVAL and ONCHANGE subscriptions, described in the next two subsections.

Release Note (any version): NGSI standard describes a third subscription type, called ONVALUE, but the current version of the Orion Context Broker doesn't support it.

ONTIMEINTERVAL
The following is the request corresponding to an ONTIMEINTERVAL subscription:

XML
(curl localhost:1026/NGSI10/subscribeContext
  -s -S --header
  'Content-Type: application/xml'
  -d @- | xmllint --format -) <<EOF
<?xml version="1.0"?>
<subscribeContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <duration>P1M</duration>
  <notifyConditions>
    <notifyCondition>
      <type>ONTIMEINTERVAL</type>
      <condValueList>
        <condValue>PT10S</condValue>
      </condValueList>
    </notifyCondition>
  </notifyConditions>
</subscribeContextRequest>
EOF

JSON (since release 0.9.0)
(curl
localhost:1026/NGSI10/subscribeContext
-s -S --header
'Content-Type: application/json'
--header 'Accept: application/json'
-d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room1"
    }
  ],
  "attributes": [
    "temperature"
  ],
  "reference":
  "http://localhost:1028/accumulate",
  "duration": "P1M",
  "notifyConditions": [
    {
      "type": "ONTIMEINTERVAL",
      "condValues": ["PT10S"]
    }
  ]
}
Let's examine in detail the different elements included in the payload:

- `entityIdList` and `attributeList` (‘entities’ and ‘attributes’ for short, in JSON) define which context elements will be included in the notification message. They work the same way as the XML elements with the same name in `queryContext request`. You can even include lists or patterns to specify entities. In this example, we are specifying that the notification has to include the temperature attribute for entity Room1.

- The callback URL to send notifications is defined with the reference element. We are using the URL of the accumulator-server.py program started before. Only one reference can be included per `subscribeContext request`. However, you can have several subscriptions on the same context elements (i.e. same entityIdList and attributeList) without any problem. Default URL schema (in the case you don't specify any) is "http", e.g. using "localhost:1028" as reference will be actually interpreted as "http://localhost:1028".

- Subscriptions have a duration, specified using the ISO 8601 standard format. Once that duration is expired, the subscription is simply ignored (however, it is still stored in the broker database and needs to be purged using the procedure described in the administration manual). You can extend the duration of a subscription updating it, as described later in this document. We are using "P1M" which means "one month".

- The `notifyCondition` element defines the "trigger" for the subscription. There is a `type` element (which value in this case is ONTIMERINTERVAL) and a `condValueList` element. The `condValueList` element structure depends on the type. In the case of ONTIMERINTERVAL, it includes exactly one `condValue` child element whose value is a time interval (using again, as usual in NGSI, the ISO 8601 format). A notification is sent with a frequency equal to that interval. In the example above we are using 10 seconds as interval.

The response corresponding to that request contains a subscription ID (a 24 hexadecimal number used for updating and cancelling the subscription - write it down because you will need it later in this tutorial) and a duration acknowledgement:

```
<?xml version="1.0"?>
<subscribeContextResponse>
  <subscribeResponse>
    <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
  </subscribeResponse>
</subscribeContextResponse>
```
If you look at the accumulator-script.py terminal window, you will see that a message resembling the following one is received each 10 seconds:

```
<notifyContextRequest>
  <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
  <originator>localhost</originator>
  <contextResponseList>
  </contextResponseList>
</notifyContextRequest>
```
<contextElement>
  <entityId type="Room" isPattern="false">
    <id>Room1</id>
  </entityId>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <contextValue>26.5</contextValue>
    </contextAttribute>
  </contextAttributeList>
</contextElement>

<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>

<notifyContextRequest>
</contextResponseList>
</contextElementResponse>

---

**JSON (since release 0.9.0)**

POST http://localhost:1028/accumulate

Content-Length: 492

User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json

Content-Type: application/json
Orion Context Broker notifies NGSI10 subscribeContext using the POST HTTP method (on the URL used as reference for the subscription) with a notifyContextRequest payload. Apart from the subscriptionId element (that matches the one in the response to subscribeContext request) and the originator element, there is a contextResponseList element which is the same that the one used in the queryContext responses.
Release Note (any version): currently, the originator is always "localhost". We will look into a more flexible way of using this in a later version.

You can do a small exercise: change the temperature value of Room1 (have a look at the update context elements section in this manual to see how to do it) and after that, check that in the next received notifyContextRequest for accumulator-server.py the contextValue element contains the new value. This exercise demonstrates that the Orion Context Broker always notifies the updated value in ONTIMEINTERVAL subscriptions.

Subscriptions can be updated using the NGSI10 updateContextSubscription. The request includes a subscriptionId that identifies the subscription to modify and the actual update payload. For example, if we want to change the notification interval to 5 seconds we will use the following (of course, replace the subscriptionId value after copy-paste with the one that you have got in the subscribeContext response in the previous step) command:

XML

```xml
(curl localhost:1026/NGSI10/updateContextSubscription -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0"?>
<updateContextSubscriptionRequest>
    <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
    <notifyConditions>
        <notifyCondition>
            <type>ONTIMEINTERVAL</type>
            <condValueList>
                <condValue>PT5S</condValue>
            </condValueList>
        </notifyCondition>
    </notifyConditions>
</updateContextSubscriptionRequest>
EOF
```

JSON (since release 0.9.0)

```json
(curl localhost:1026/NGSI10/updateContextSubscription -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @ | python -mjson.tool) <<EOF
{
```
"subscriptionId": "51c04a21d714fb3b37d7d5a7",
"notifyConditions": [
  {
    "type": "ONTIMEINTERVAL",
    "condValues": [
      "PT5S"
    ]
  }
]
}

EOF

The response is very similar to the one for subscribeContext request:

XML

```xml
<?xml version="1.0"?>
<updateContextSubscriptionResponse>
  <subscribeResponse>
    <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
  </subscribeResponse>
</updateContextSubscriptionResponse>
```

JSON (since release 0.9.0)

```json
{
  "subscribeResponse" : {
    "subscriptionId" : "51c04a21d714fb3b37d7d5a7",
  }
}
```

You can check in accumulator-server.py that the notification frequency has changed to 5 seconds.
Finally, you can cancel a subscription using the NGSI10 unsubscribeContext operation, that just uses de subscriptionId in the request payload (replace the subscriptionId value after copy-paste with the one that you get in the subscribeContext response in the previous step):

**XML**

```xml
(curl localhost:1026/NGSI10/unsubscribeContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0"?>
<unsubscribeContextRequest>
   <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
</unsubscribeContextRequest>
EOF
```

**JSON (since release 0.9.0)**

```json
(curl localhost:1026/NGSI10/unsubscribeContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @ | python -m json.tool) <<EOF
{
   "subscriptionId": "51c04a21d714fb3b37d7d5a7"
}
EOF
```

The response is just an acknowledgement of that the cancellation was successful.

**XML**

```xml
<?xml version="1.0"?>
<unsubscribeContextResponse>
   <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
   <statusCode>
      <code>200</code>
      <reasonPhrase>OK</reasonPhrase>
   </statusCode>
</unsubscribeContextResponse>
```
JSON (since release 0.9.0)

```
{
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    },
    "subscriptionId": "51c04a21d714fb3b37d7d5a7"
}
```

You can have a look at accumulator-server.py to check that the notification flow has stopped.

**Release Note (minor or equal to 0.6.0 FIWARE 3.1.1):** due to a bug in NGSI processing, use the following payload in the requests (using the "intermediate" `<unsubscribeContext>` XML element). Only XML case (JSON rendering was not included in version 0.6.0).

```
(curl localhost:1026/NGSI10/unsubscribeContext -s -S --header
 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<unsubscribeContextRequest>
  <unsubscribeContext>
    <subscriptionId>51c04a21d714fb3b37d7d5a7</subscriptionId>
  </unsubscribeContext>
</unsubscribeContextRequest>
EOF
```

**ONCHANGE**

We assume that the accumulator-server.py program is still running. Otherwise, start it as described here.

ONCHANGE subscriptions are used when you want to be notified not when a given time interval has passed but when some attribute changes. Let's consider the following example:
XML

(curl localhost:1026/NGSI10/subscribeContext -s -S --header "Content-Type: application/xml" -d @- | xmllint --format -) <<EOF
<?xml version="1.0"?>
<subscribeContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <duration>P1M</duration>
  <notifyConditions>
    <notifyCondition>
      <type>ONCHANGE</type>
      <condValueList>
        <condValue>pressure</condValue>
      </condValueList>
    </notifyCondition>
  </notifyConditions>
  <throttling>PT5S</throttling>
</subscribeContextRequest>
EOF

JSON (since release 0.9.0)
(curl localhost:1026/NGSI10/subscribeContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
   "entities": [
      {
         "type": "Room",
         "isPattern": "false",
         "id": "Room1"
      }
   ],
   "attributes": [
      "temperature"
   ],
   "reference": "http://localhost:1028/accumulate",
   "duration": "P1M",
   "notifyConditions": [
      {
         "type": "ONCHANGE",
         "condValues": [
            "pressure"
         ]
      }
   ],
   "throttling": "PT5S"
}
EOF
Having a look at the payload we can check that it is very similar to the one used in ONTIMEINTERVAL, with two exceptions:

- The notifyCondition element uses the type ONCHANGE (obviously :) but, in this case the condValueList contains an actual list of condValue elements, each one with an attribute name. They define the "triggering attributes", i.e. attributes that upon creation/change due to entity creation or update trigger the notification. The rule is that if at least one of the attributes in the list changes (e.g. some kind of "OR" condition), then a notification is sent. But note that that notification includes the attributes in the attributeList part, which doesn't necessarily include any attribute in the condValue. For example, in this case, when Room1 pressure changes the Room1 temperature value is notified, but not pressure itself. If you want also pressure to be notified, the request would need to include <attribute>pressure</attribute> within the attributeList (or to use an empty attributeList, which you already know means "all the attributes in the entity"). Now, this example here, to be notified of the value of temperature each time the value of pressure changes may not be too useful. The example is chosen this way only to show the enormous flexibility of subscriptions.

- The throttling element is used to specify a minimum inter-notification arrival time. So, setting throttling to 5 seconds as in the example above makes that a notification will not be sent if a previous notification was sent less than 5 seconds ago, no matter how many actual changes take place in that period. This is to not stress the notification receptor in case of having context producers that update attribute values too frequently. Actually, throttling is not an "exclusive" field for ONCHANGE subscriptions: from a theoretical point of view it can be used in ONTIMEINTERVAL subscriptions but, given that in that case you can precisely control the notification frequency it doesn't have any practical sense.

As in ONTIMEINTERVAL subscriptions, the response consists of a subscription ID, a duration acknowledgement and (given that we used throttling in the request) a throttling acknowledgement:

```xml
<?xml version="1.0"?>
<subscribeContextResponse>
  <subscribeResponse>
    <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
    <duration>P1M</duration>
    <throttling>PT5S</throttling>
  </subscribeResponse>
</subscribeContextResponse>
```
Let's have a look now at accumulator-server.py. We will see one (and just one by the moment, no matter how much you wait) notifyContextRequest, similar to this one:

**XML**

```
POST http://localhost:1028/accumulate
Content-Length: 739
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/xml

<notifyContextRequest>
  <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
  <originator>localhost</originator>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</notifyContextRequest>
```
Future Internet Core Platform

```xml
<entityId>
<contextAttributeList>
    <contextAttribute>
        <name>temperature</name>
        <type>centigrade</type>
        <contextValue>26.5</contextValue>
    </contextAttribute>
</contextAttributeList>
</contextElement>
<contextElementResponse>
    <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
    </statusCode>
</contextElementResponse>
</contextResponseList>
</notifyContextRequest>

JSON (since release 0.9.0)

POST http://localhost:1028/accumulate
Content-Length: 492
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/json

{
    "subscriptionId" : "51c0ac9ed714fb3b37d7d5a8",
    "originator" : "localhost",
```
"contextResponses" : [ 
    
    "contextElement" : { 
    "attributes" : [ 
    
    "name" : "temperature",
    "type" : "centigrade",
    "value" : "26.5"
    
    }],
    "type" : "Room",
    "isPattern" : "false",
    "id" : "Room1"
    },
    "statusCode" : { 
    "code" : "200",
    "reasonPhrase" : "OK"
    }
    
    ]
    ]
    ]

You may wonder why accumulator-server.py is getting this message if you don't actually do any update. This is because the Orion Context Broker considers the transition from "non existing subscription" to "subscribed" as a change.

**Release Note (any version):** NGSI specification is not clear on if an initial notifyContextRequest has to be sent in this case or not. On one hand, some developers have told us that it might be useful to know the initial values before starting to receive notifications due to actual changes. On the other hand, an application can get the initial status using queryContext. Thus, this behavior could be changed in a later version. What's your opinion? :)

Now, do the following exercise, based on what you know from update context: Do the following 4 updates, in sequence and letting pass more than 5 seconds between one and the next (to avoid losing notifications due to throttling):
• update Room1 temperature to 27: nothing happens, as temperature is not the triggering attribute
• update Room1 pressure to 765: you will get a notification with the current value of Room1 temperature (27)
• update Room1 pressure to 765: nothing happens, as the broker is clever enough to know that the previous value to the updateContext request was also 765 so no actual update have occurred and consequently no notification is sent.
• update Room2 pressure to 740: nothing happens, as the subscription is for Room1, not Room2.

Next, try to check how throttling is enforced. Update Room1 pressure fast, without letting pass 5 seconds and you will see that the second notification doesn't arrive to accumulator-server.py.

You can update and cancel ONCHANGE subscriptions in the same way as ONTIMEINTERVAL subscriptions. You can do that as a final exercise in this section of the tutorial, e.g try to set a new throttling value, check that it works as expected and cancel after that.

### 2.3.2.1.5 Summary of NGSI10 standard operations URLs

Each standard operation has a unique URL. All of them use the POST method. The summary is below:

- <host:port>/ngsi10/updateContext
- <host:port>/ngsi10/queryContext
- <host:port>/ngsi10/subscribeContext
- <host:port>/ngsi10/updateContextSubscription
- <host:port>/ngsi10/unsubscribeContext

### 2.3.2.2 Tutorial on NGSI10 convenience operations

This section describes the different convenience operations described as part of the FI-WARE NGSI REST API NGSI10 that Orion Context Broker supports, showing examples of requests and responses. Convenience operations are a set of operations that have been defined by FI-WARE project to ease the usage of NGSI implementations as a complement to the standard operations defined in the OMA NGSI specification (see the section on additional information later in this manual).

Don't forget to restart the broker before starting this tutorial as described previously in this document.

At the end of this section, you will have learnt to use convenience operations as a handy alternative to some standard operations described in the previous section. It is highly recommended to do that tutorial before, to get familiar with update and query context, etc. and to be able to compare between the two approaches.
2.3.2.2.1  *Convenience Entity Creation*

Orion Context Broker will start in an empty state, so first of all we need to make it aware of the existence of certain entities. Thus, let's first create Room1 entity with temperature and pressure attributes (with its initial values)

**XML**

```xml
(curl localhost:1026/NGSI10/contextEntities/Room1 -s -S --header 'Content-Type: application/xml' -X POST -d @- | xmllint --format -) << EOF
<?xml version="1.0" encoding="UTF-8"?>
<appendContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <contextValue>23</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>pressure</name>
      <type>mmHg</type>
      <contextValue>720</contextValue>
    </contextAttribute>
  </contextAttributeList>
</appendContextElementRequest>
EOF
```

**JSON (since release 0.10.0)**

```json
(curl localhost:1026/NGSI10/contextEntities/Room1 -s -S --header 'Content-Type:
```
application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
    "attributes": [
    {
        "name": "temperature",
        "type": "centigrade",
        "value": "23"
    },
    {
        "name": "pressure",
        "type": "mmHg",
        "value": "720"
    }
    ]
}
EOF

the response is:

```xml
<?xml version="1.0"?>
<appendContextElementResponse>
  <contextResponseList>
    <contextAttributeResponse>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
        </contextAttribute>
      </contextAttributeList>
    </contextAttributeResponse>
  </contextResponseList>
</appendContextElementResponse>
```
<type>centigrade</type>
<contextValue/>
</contextAttribute>
<contextAttribute>
  <name>pressure</name>
  <type>mmHg</type>
  <contextValue/>
</contextAttribute>
</contextAttributeList>
<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextAttributeResponse>
</contextResponseList>
</appendContextElementResponse>

---

**JSON (since release 0.10.0)**

```
{
  "contextResponses": [
    {
      "attributes": [
        {
          "name": "temperature",
          "type": "centigrade",
          "value": ""
        },
        {
```

D.6.4.3: FI-WARE Users and Programmers Guide 71
"name": "pressure",
"type": "mmHg",
"value": ""
}
],
"statusCode": {
"code": "200",
"reasonPhrase": "OK"
}
}
]
}
}

Now, let's do the same with Room2:

```
XML

(curl localhost:1026/NGSI10/contextEntities/Room2 -s -S --header 
'Content-Type: application/xml' -X POST -d @- | xmllint --format -) <<
EOF
<?xml version="1.0" encoding="UTF-8"?>
<appendContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <contextValue>21</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>pressure</name>
      <type>mmHg</type>
    </contextAttribute>
  </contextAttributeList>
</appendContextElementRequest>
```
<contextValue>711</contextValue>
</contextAttribute>
<contextAttributeList>
</appendContextElementRequest>
EOF

**JSON (since release 0.10.0)**

(curl localhost:1026/NGSI10/contextEntities/Room2 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "attributes" : [

  {
    "name" : "temperature",
    "type" : "centigrade",
    "value" : "21"
  },

  {
    "name" : "pressure",
    "type" : "mmHg",
    "value" : "711"
  }
  ]
}

EOF

which response is:

XML
<?xml version="1.0"?>
<appendContextElementResponse>
  <contextResponseList>
    <contextAttributeResponse>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue/>
        </contextAttribute>
        <contextAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <contextValue/>
        </contextAttribute>
      </contextAttributeList>
    </contextAttributeResponse>
    <statusCode>
      <code>200</code>
      <reasonPhrase>OK</reasonPhrase>
    </statusCode>
  </contextResponseList>
</appendContextElementResponse>

**JSON** *(since release 0.10.0)*

```json
{
    "contextResponses": [
```
Comparing to entity creation based on standard operation we observe the following differences:

- We are using the POST verb on the /ngsi10/contextEntities/{EntityID} resource to create new entities.
- We cannot create more than one entity at a time using convenience operation requests.
- The payload of request and response in convenience operations are very similar to the ones used in standard operations, the contextAttributeList and contextResponseList elements are the same.
- Entity type cannot be registered. Thus, we cannot specify whether "Room1" is of type "Room" or "Space". This lack of typing has some important implications.
Apart from simple values (i.e. strings) for attribute values, you can also use complex structures or custom metadata. These are advance topics, described in this section and this other, respectively.

2.3.2.2.2  Convenience Query Context

Finally, let’s describe convenience operations for querying context information. We can query all the attribute values of a given entity, e.g. Room1 attributes:

```xml
curl localhost:1026/ngsi10/contextEntities/Room1 -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```bash
curl localhost:1026/NGSI10/contextEntities/Room1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

which response is:

```xml
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    <entityId type="" isPattern="false">
      <id>Room1</id>
    </entityId>
    <contextAttributeList>
      <contextAttribute>
        <name>temperature</name>
        <type>centigrade</type>
        <contextValue>23</contextValue>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
</contextElementResponse>
```
<contextAttribute>
  <name>pressure</name>
  <type>mmHg</type>
  <contextValue>720</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
=statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>

---

**JSON (since release 0.10.0)**

```json
{
  "contextElement": {
    "attributes": [
      {
        "name": "temperature",
        "type": "centigrade",
        "value": "23"
      },
      {
        "name": "pressure",
        "type": "mmHg",
        "value": "720"
      }
    ],
    "attributes": [
      {
        "name": "temperature",
        "type": "centigrade",
        "value": "23"
      },
      {
        "name": "pressure",
        "type": "mmHg",
        "value": "720"
      }
    ]
  }
}
```
We can also query a single attribute of a given entity, e.g. Room2 temperature:

**XML**

```bash
curl localhost:1026/ngsi10/contextEntities/Room2/attributes/temperature -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```bash
curl localhost:1026/NGSI10/contextEntities/Room2/attributes/temperature -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

which response is:

**XML**

```xml
<?xml version="1.0"?>
<contextAttributeResponse>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
    </contextAttribute>
  </contextAttributeList>
</contextAttributeResponse>
```
<type>centigrade</type>
<contextValue>21</contextValue>
</contextAttribute>
</contextAttributeList>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextAttributeResponse>

**JSON (since release 0.10.0)**

```json
{
    "attributes": [
        {
            "name": "temperature",
            "type": "centigrade",
            "value": "21"
        }
    ],
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    }
}
```

Comparing to [standard queryContext operation](#) we observe the following differences:

- Convenience operations use the GET method without payload in the request (simpler than standard operation)
The response contextElementResponse element used in the response of the convenience operation to query all the attributes of an entity has the same structure as the one that appears inside responses for standard queryContext. However, the contextAttributeResponse element in the response of the convenience operation used as response to the query of a single attribute of an entity is new.

It is not possible to use convenience operations to query for lists of entities, entity patterns nor lists of attributes.

**Since release 0.9.0 (FI-WARE 3.2.3):** you can also query by all the entities belonging to the same type, either all the attributes or a particular one, as shown below. First, create an couple of entities of type Car using standard updateContext APPEND operations (given that, as described in previous section, you cannot create entities with types using convenience operations):

```xml
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Car" isPattern="false">
        <id>Car1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>speed</name>
          <type>km/h</type>
          <contextValue>75</contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>fuel</name>
          <type>liter</type>
          <contextValue>12.5</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

XML
Future Internet Core Platform

</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
  
  
  
  "type": "Car",
  "isPattern": "false",
  "id": "Car1",
  "attributes": [
  
  
  "name": "speed",
  "type": "km/h",
  "value": "75"
  },
  
  
  "name": "fuel",
  "type": "liter",
  "value": "12.5"
  ]
}
EOF
}(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Car" isPattern="false">
        <id>Car2</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>speed</name>
          <type>km/h</type>
          <contextValue>90</contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>fuel</name>
          <type>liter</type>
          <contextValue>25.7</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
EOF
</contextElementList>
</updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

---

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "contextElements": [
  {
    "type": "Car",
    "isPattern": "false",
    "id": "Car2",
    "attributes": [
    {
      "name": "speed",
      "type": "km/h",
      "value": "90"
    },
    {
      "name": "fuel",
      "type": "liter",
      "value": "25.7"
    }
  ]
  }
],

EOF
Request to get all the attributes:

**XML**

```bash
curl localhost:1026/ngsi10/contextEntityTypes/Car -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```bash
curl localhost:1026/NGSI10/contextEntityTypes/Car -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

Response:

```xml
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Car" isPattern="false">
          <id>Car1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
```

```
<contextElementResponse>
  <contextElement>
    <entityId type="Car" isPattern="false">
      <id>Car2</id>
    </entityId>
    <contextAttributeList>
      <contextAttribute>
        <name>speed</name>
        <type>km/h</type>
        <contextValue>90</contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>fuel</name>
        <type>liter</type>
        <contextValue>12.5</contextValue>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
</contextElementResponse>
<contextValue>25.7</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</queryContextResponse>

**JSON** *(since release 0.10.0)*

```json
{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "speed",
            "type": "km/h",
            "value": "75"
          },
          {
            "name": "fuel",
            "type": "liter",
            "value": "12.5"
          }
        ]
      }
    }
  ]
}
```

},
"id": "Car1",
"isPattern": "false",
"type": "Car"
},
"statusCode": {
  "code": "200",
  "reasonPhrase": "OK"
}
},
{
  "contextElement": {
    "attributes": [
      {
        "name": "speed",
        "type": "km/h",
        "value": "90"
      },
      {
        "name": "fuel",
        "type": "liter",
        "value": "25.7"
      }
    ],
    "id": "Car2",
    "isPattern": "false",
    "type": "Car"
  },
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  }
}
"reasonPhrase": "OK"

Request to get only one attribute (e.g. speed):

XML

```
curl localhost:1026/ngsi10/contextEntityTypes/Car/attributes/speed -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

JSON (since release 0.10.0)

```
curl localhost:1026/NGSI10/contextEntityTypes/Car/attributes/speed -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -m json.tool
```

Response:

XML

```
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Car" isPattern="false">
          <id>Car1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
```

D.6.4.3: FI-WARE Users and Programmers Guide 88
<name>speed</name>
<type>km/h</type>
<contextValue>75</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
<contextElementResponse>
<contextElement>
<entityId type="Car" isPattern="false">
  <id>Car2</id>
</entityId>
<contextAttributeList>
<contextAttribute>
  <name>speed</name>
  <type>km/h</type>
  <contextValue>90</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
JSON (since release 0.10.0)

```json
{
  "contextResponses": [
  {
    "contextElement": {
      "attributes": [
      {
        "name": "speed",
        "type": "km/h",
        "value": "75"
      }
    ],
    "id": "Car1",
    "isPattern": "false",
    "type": "Car"
    }
  },
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  }
  }
}
```
Future Internet Core Platform

"type": "km/h",
"value": "90"
}
],
"id": "Car2",
"isPattern": "false",
"type": "Car"
},
"statusCode": {
  "code": "200",
  "reasonPhrase": "OK"
}
}
]

Additional comments:

- You can also use geographical scopes in your queries. This is an advance topic, described in this section.

- Note that by default only 20 entities are returned (which is fine for this tutorial, but probably not for a real utilization scenario). In order to change this behaviour, see the section on pagination in this manual.

2.3.2.2.3 Convenience Update Context
Let's set the Room1 temperature and pressure values:

XML

(curl localhost:1026/NGSI10/contextEntities/Room1/attributes -s -S --header 'Content-Type: application/xml' -X PUT -d @- | xmllint --format - ) << EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextElementRequest>
  <contextAttributeList>
Future Internet Core Platform

<contextAttribute>
  <name>temperature</name>
  <type>centigrade</type>
  <contextValue>26.5</contextValue>
</contextAttribute>

<contextAttribute>
  <name>pressure</name>
  <type>mmHg</type>
  <contextValue>763</contextValue>
</contextAttribute>
</contextAttributeList>
</updateContextElementRequest>
EOF

---

**JSON (since release 0.10.0)**

(curl localhost:1026/NGSI10/contextEntities/Room1/attributes -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -X PUT -d @- | python -mjson.tool) << EOF
{
  "attributes" : [
  {
    "name" : "temperature",
    "type" : "centigrade",
    "value" : "26.5"
  },
  {
    "name" : "pressure",
    "type" : "mmHg",
    "value" : "763"
  }
  ]
}
EOF
the response is:

XML

```xml
<?xml version="1.0"?>
<updateContextElementResponse>
  <contextResponseList>
    <contextAttributeResponse>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue/>
        </contextAttribute>

        <contextAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <contextValue/>
        </contextAttribute>
      </contextAttributeList>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextAttributeResponse>
  </contextResponseList>
</updateContextElementResponse>
```

JSON (since release 0.10.0)
Now, let’s do the same with Room2:

```xml
(curl localhost:1026/NGSI10/contextEntities/Room2/attributes -s -S --header 'Content-Type: application/xml' -X PUT -d @ - | xml1int --format - ) << EOF
```
<?xml version="1.0" encoding="UTF-8"?>
<updateContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <contextValue>27.4</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>pressure</name>
      <type>mmHg</type>
      <contextValue>755</contextValue>
    </contextAttribute>
  </contextAttributeList>
</updateContextElementRequest>
EOF

---

**JSON** *(since release 0.10.0)*

(curl localhost:1026/NGSI10/contextEntities/Room2/attributes -s -S --header "Content-Type: application/json" --header "Accept: application/json" -X PUT -d @- | python -mjson.tool) << EOF
{
    "attributes" : [
    {
        "name" : "temperature",
        "type" : "centigrade",
        "value" : "27.4"
    },
    {
        "name" : "pressure",
        "type" : "mmHg",
        "value" : "755"
    }
}
which response is:

```xml
<?xml version="1.0"?>
<updateContextElementResponse>
    <contextResponseList>
        <contextAttributeResponse>
            <contextAttributeList>
                <contextAttribute>
                    <name>temperature</name>
                    <type>centigrade</type>
                    <contextValue/>
                </contextAttribute>
                <contextAttribute>
                    <name>pressure</name>
                    <type>mmHg</type>
                    <contextValue/>
                </contextAttribute>
            </contextAttributeList>
        </contextAttributeResponse>
        <statusCode>
            <code>200</code>
            <reasonPhrase>OK</reasonPhrase>
        </statusCode>
    </contextResponseList>
</updateContextElementResponse>
```
You can update a single attribute of a given entity (previous to release 0.11.0 you can only do it that attribute uses metadata ID).

Comparing to standard updateContext operation we observe the following differences:

- We cannot update more than one entity at a time using convenience operation requests.
The payload of request and response in convenience operations are very similar to the ones used in standard operations, the contextAttributeList and contextResponseList elements are the same.

Apart from simple values (i.e. strings) for attribute values, you can also use complex structures or custom metadata. These are advance topics, described in this section and this other, respectively.

2.3.2.2.4 Convenience operations for context subscriptions

Since release 0.9.0 (FI-WARE 3.2.3): new functionality.

You can use the following convenience operations to manage context subscriptions:

- POST /ngsi10/contextSubscriptions, to create the subscription, using the same payload as standard subscribeContext operation.
- PUT /ngsi10/contextSubscriptions/{subscriptionID}, to update the subscription identified by {subscriptionID}, using the same payload as standard updateContextSubscription operation. The ID in the payload must match the ID in the URL.
- DELETE /ngsi10/contextSubscriptions/{subscriptionID}, to cancel the subscription identified by {subscriptionID}. In this case, payload is not used.

2.3.2.2.5 Summary of NGSI10 convenience operations URLs

Convenience operations use a URL to identify the resource and a HTTP verb to identify the operation on that resource following the usual REST convention: GET is used to retrieve information, POST is used to create new information, PUT is used to update information and DELETE is used to destroy information.

You find a summary in the following document.

2.3.3 Context availability management using NGSI9

2.3.3.1 Tutorial on NGSI9 standard operations

This section describes the different standard NGSI9 operations that the Orion Context Broker supports, showing examples of requests and responses. We use the term "standard" as they are directly derived from the OMA NGSI specification, to distinguish them from the other family of operations ("convenience") which has been defined by the FI-WARE project to ease the usage of NGSI implementations (see the section on additional information later in this manual).

Don't forget to restart the broker before starting this tutorial as described previously in this document.

At the end of this section, you will have the basic knowledge to create applications (both context producers and consumers) using Orion Context Broker with NGSI9 standard operations:

- registerContext
- discoverContextAvailability
- subscribeContextAvailability
Future Internet Core Platform

- updateContextAvailabilitySubscription
- unsubscribeContextAvailability

2.3.3.1.1  *Register Context operation*

First of all you have to register Room1 and Room2. In order to do so, we use the following NGSI9 registerContext operation:

```xml
(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <entityId type="Room" isPattern="false">
          <id>Room2</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <isDomain>false</isDomain>
        </contextRegistrationAttribute>
        <contextRegistrationAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <isDomain>false</isDomain>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
    </contextRegistration>
  </contextRegistrationList>
</registerContextRequest>
<<EOF
```
<providingApplication>http://mysensors.com/Rooms</providingApplication>

</contextRegistration>
</contextRegistrationList>
<duration>P1M</duration>
</registerContextRequest>

---

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<<EOF
{
   "contextRegistrations": [ 
   {
      "entities": [
         {
            "type": "Room",
            "isPattern": "false",
            "id": "Room1"
         },
         {
            "type": "Room",
            "isPattern": "false",
            "id": "Room2"
         }
      ],
      "attributes": [
         {
            "name": "temperature",
            "type": "centigrade",
            "isDomain": "false"
         }
      ]
   }
}

EOF
The payload includes a list of contextRegistration elements, each one with the following information:

- A list of entities to be registered. In our case, they are the Room1 and Room2 entities. For each entity we specify a type (in this case, we are using "Room" as type) and an ID (which are "Room1" and "Room2" respectively). The isPattern field is not actually used in registerContext, so it always has a value of "false".

- A list of attributes to register for the entities. In our case, they are the temperature and pressure attributes. For each one, we define a name, a type and whether it is a domain attribute or not.
  
  - **Release Note (any version):** Orion Context Broker doesn't perform any checking on types (e.g. it doesn't check that when a context producer application updates the value of the temperature, this value is formatted as a centigrade degree measure like "25.5" or "-40.23" and not something like "very cool" or "-275.2"). In addition, domain attributes are not supported, so isDomain must always be set to "false".

- The URL of the providing application. By "providing application" (an NGSI concept) we mean the URL that represents the actual context information for the entities and attributes being registered. In our example we are assuming that all the sensors are provided by http://mysensors.com/Ro</p>
Finally, note that the payload includes a duration element. The duration element sets the duration of the registration so after that time has passed it can be considered as expired (however, duration can be extended). We use the ISO 8601 standard for duration format. We are using "P1M" which means "one month" (a very large amount, probably enough time to complete this tutorial :).

We will get the following response (XML case):

```
<?xml version="1.0"?><registerContextResponse>
  <duration>P1M</duration>
  <registrationId>52a744b011f5816465943d58</registrationId>
</registerContextResponse>
```

The registrationId (whose value will be different when you run the request, as it is generated using the timestamp of the current time :) is a 24 hexadecimal digit which provides an unique reference to the registration. It is used for updating the registration as explained later in this manual.

2.3.3.1.2 Discover Context Availability operation

So now the broker has registration information about Room1 and Room2. How can we access that information? Using the NGSI9 discoverContextAvailability operation. For example, we can discover registrations for Room1 using:

```
(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
```
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList/>
</discoverContextAvailabilityRequest>

JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room1"
    }
  ]
}
EOF

This would produce the following response:

XML

<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      ...
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
<contextRegistration>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <contextRegistrationAttributeList>
    <contextRegistrationAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <isDomain>false</isDomain>
    </contextRegistrationAttribute>
    <contextRegistrationAttribute>
      <name>pressure</name>
      <type>mmHg</type>
      <isDomain>false</isDomain>
    </contextRegistrationAttribute>
  </contextRegistrationAttributeList>
  <providingApplication>http://mysensors.com/Rooms</providingApplication>
</contextRegistration>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</discoverContextAvailabilityResponse>

JSON (since release 0.9.0)

{
    "contextRegistrationResponses": [
        {
            "contextRegistration": {
                "attributes": [
                    {  

D.6.4.3: FI-WARE Users and Programmers Guide 104
"isDomain": "false",
"name": "temperature",
"type": "centigrade"
}
,
{
  "isDomain": "false",
  "name": "pressure",
  "type": "mmHg"
}
],
"entities": [
{
  "id": "Room1",
  "isPattern": "false",
  "type": "Room"
}
],
"providingApplication": "http://mysensors.com/Roms"

Note that we used an empty attributeList in the request ('attributes' for short, in JSON). Doing so, the discover searches for Room1, no matter which attributes have been registered. If we want to be more precise, we can include the name of an attribute to search for:

XML

(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
<entityId type="Room" isPattern="false">
  <id>Room1</id>
</entityId>
</entityIdList>
<attributeList>
  <attribute>temperature</attribute>
</attributeList>
</discoverContextAvailabilityRequest>
EOF

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "false",
      "id": "Room1"
    }
  ],
  "attributes": [
    "temperature"
  ]
}
EOF

which produces the following response:

**XML**
```xml
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room1</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Rooms</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
```

**JSON (since release 0.9.0)**

```json
{
  "contextRegistrationResponses": [
    {
      "contextRegistration": {
        "attributes": [
          {
            "isDomain": "false",
```

D.6.4.3: FI-WARE Users and Programmers Guide 107
"name": "temperature",
"type": "centigrade"
}
],
"entities": [
{
"id": "Room1",
"isPattern": "false",
"type": "Room"
}
],
"providingApplication": "http://mysensors.com/R

If the broker doesn't have any registration information, it will return a response telling so. Thus, the following request:

```
XML
(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>humidity</attribute>
  </attributeList>
</discoverContextAvailabilityRequest>
```
</discoverContextAvailabilityRequest>

EOF

---

**JSON (since release 0.9.0)**

```
(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
    "entities": [
    {
        "type": "Room",
        "isPattern": "false",
        "id": "Room1"
    }
  ],
  "attributes": [
    "humidity"
  ]
}
EOF
```

would produce the following response:

---

**XML**

```
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <errorCode>
    <code>404</code>
    <reasonPhrase>No context element registrations found</reasonPhrase>
  </errorCode>
</discoverContextAvailabilityResponse>
```
JSON (since release 0.9.0)

{
    "errorCode": {
        "code": "404",
        "reasonPhrase": "No context element registrations found"
    }
}

You can also search for a list of entities, e.g. to discover temperature in both Room1 and Room2:

XML

(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
    <entityIdList>
        <entityId type="Room" isPattern="false">
            <id>Room1</id>
        </entityId>
        <entityId type="Room" isPattern="false">
            <id>Room2</id>
        </entityId>
    </entityIdList>
    <attributeList>
        <attribute>temperature</attribute>
    </attributeList>
</discoverContextAvailabilityRequest>
EOF
JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1"
  },
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room2"
  }
  ],
  "attributes": [
    "temperature"
  ]
}
EOF

which will produce the following response:

```
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
```

D.6.4.3: FI-WARE Users and Programmers Guide 111
<entityId type="Room" isPattern="false">
    <id>Room1</id>
</entityId>

<entityId type="Room" isPattern="false">
    <id>Room2</id>
</entityId>

<contextRegistrationAttributeList>
    <contextRegistrationAttribute>
        <name>temperature</name>
        <type>centigrade</type>
        <isDomain>false</isDomain>
    </contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rooms</providingApplication>

</contextRegistration>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</discoverContextAvailabilityResponse>

---

**JSON (since release 0.9.0)**

```json
{
    "contextRegistrationResponses": [ {
        "contextRegistration": {
            "attributes": [ {
                "isDomain": "false",
                "name": "temperature",
                "type": "centigrade"
            } ]
        }
    } ]
}
```
Finally, a powerful feature of Orion Context Broker is that you can use a regular expression for the entity ID. For example, you can discover entities which ID starts with "Room" using the regex "Room.*". In this case, you have to set isPattern to "true" as shown below:

```
XML
(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="true">
      <id>Room.*</id>
    </entityId>
  </entityIdList>
</discoverContextAvailabilityRequest>
```
<attributeList>
  <attribute>temperature</attribute>
</attributeList>
</discoverContextAvailabilityRequest>

---

**JSON (since release 0.9.0)**

```
(curl localhost:1026/NGSI9/discoverContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "true",
      "id": "Room.*"
    }
  ],
  "attributes": [
    "temperature"
  ]
}
EOF
```

This will produce the exact same response as the previous example.

Note that by default only 20 registrations are returned (which is fine for this tutorial, but probably not for a real utilization scenario). In order to change this behaviour, see the section on pagination in this manual.

### 2.3.3.1.3 Context availability subscriptions

The NGSI9 operations you know up to now (registerContext and discoverContextAvailability) are the basic building blocks for synchronous context producer and context consumer applications. However, Orion Context Broker has another powerful feature that you can take advantage of: the ability to context information availability so when "something" happens (we will explain the different cases for that "something") your application will get an asynchronous notification. In that way, you don't need to
continuously repeat `discoverContextAvailability` requests (i.e. polling), the Orion Context Broker will let you know the information when it comes.

We assume that the accumulator-server.py program is still running. Otherwise, start it as described in the previous section.

Context availability subscriptions are used when we want to be notified not about context information (i.e. the values of attributes of some entities) but about the availability of the context sources themselves. We will clarify what this means with an example.

Let's consider that your context consumer application wants to be notified each time the Orion Context Broker gets aware of a new Room registration, e.g. because a new Room icon has to be drawn in the graphical user interface that the application is offering to final users. Thus, each time a new entity of type "Room" is registered in the broker (using `registerContext` operation), the broker must be able to send notifications.

In order to configure this behavior, we use the following NGSI9 `subscribeContextAvailability` request:

```
XML
(curl localhost:1026/NGSI9/subscribeContextAvailability -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0"?>
<subscribeContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <duration>P1M</duration>
</subscribeContextAvailabilityRequest>
EOF
```

```
JSON (since release 0.9.0)
```

(curl localhost:1026/NGSI9/subscribeContextAvailability -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Room",
      "isPattern": "true",
      "id": ".*"
    }
  ],
  "attributes": [
    "temperature"
  ],
  "reference": "http://localhost:1028/accumulate",
  "duration": "P1M"
}
EOF

The payload has the following elements:

- entityIdList and attributeList (‘entities’ and ‘attributes’ for short, in JSON) define which context availability information we are interested in. They are used to select the context registrations to include in the notifications. They work in the same way as the XML elements with the same name in discoverContextAvailability request. In this case, we are stating that we are interested in context availability about "temperature" attribute in any entity of type "Room" ("any" is represented by the ".*" pattern, which is a regular expression that matches any string).

- The callback URL to send notifications is defined with the reference element. We are using the URL of the accumulator-server.py program started before. Only one reference can be included per subscribeContextAvailability request. However, you can have several subscriptions on the same context availability elements (i.e. same entityIdList and attributeList) without any problem. Default URL schema (in the case you don't specify any) is "http", e.g. using "localhost:1028" as reference will be actually interpreted as "http://localhost:1028".

- Subscriptions have a duration (specified in the duration elements in the same format as registerContext request). Once that duration expires, the subscription is ignored (however, it is still stored in the broker database and needs to be purged using the procedure described in the administration manual). You can extend the duration of a subscription updating it, as described later in this document. We are using "P1M" which means "one month".
As you can see, the structure of subscriptionContextAvailability is similar to the structure of NGSI10 subscribeContext, although in this case we don't use notifyConditions nor throttling.

The response to the subscribeContextAvailability request is a subscription ID (a 24 hexadecimal number used for updating and cancelling the subscription - write it down because you will need it in later steps of this tutorial) and a duration acknowledgement. Again, pretty similar to a subscribeContext.

```
<?xml version="1.0"?>
<subscribeContextAvailabilityResponse>
   <subscriptionId>52a745e011f5816465943d59</subscriptionId>
   <duration>P1M</duration>
</subscribeContextAvailabilityResponse>
```

```
{  
   "duration": "P1M",
   "subscriptionId": "52a745e011f5816465943d59"
}
```

Looking at accumulator-server.py, we will see the following initial notification:

```
POST http://localhost:1028/accumulate
Content-Length: 940
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/xml

<notifyContextAvailabilityRequest>
```
<subscriptionId>52a745e011f5816465943d59</subscriptionId>
<contextRegistrationResponseList>
  <contextRegistrationResponse>
    <contextRegistration>
      <entityIdList>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <entityId type="Room" isPattern="false">
          <id>Room2</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <isDomain>false</isDomain>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
      <providingApplication>http://mysensors.com/Rooms</providingApplication>
    </contextRegistration>
  </contextRegistrationResponse>
</contextRegistrationResponseList>

JSON (since release 0.9.0)

POST http://localhost:1028/accumulate
Content-Length: 638
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Orion Context Broker notifies NGSI9 subscribeContextAvailability using the POST HTTP method (on the URL used as reference for the subscription) with a notifyContextAvailabilityRequest payload. Apart from
the subscriptionId element (that matches the one in the response to subscribeContextAvailability request) and the originator element, the contextResponseList element is the same than the one used in the discoverContextAvailability responses.

**Release Note (any version):** currently, the originator is always "localhost". We will look into a more flexible way of using this in a later version.

The initial notification includes all the currently registered entities that match the entityIdList/attributeList used in subscribeContextAvailability request. That is, the registration corresponding to Room1 and Room2 temperature. Note that, although Room1 and Room2 registered two attributes (temperature and pressure) only temperature is shown, as the attributeList in subscribeContextAvailability only includes temperature.

**Release Note (any version):** The NGSI specification is not clear on if an initial notifyContextAvailabilityRequest has to be sent in this case or not. On one hand, some developers have told us that it might be useful to know the initial registrations before starting to receive notifications due to new registrations. On the other hand, an application can get the initial status using discoverContextAvailability. Thus, this behavior could be changed in a later version. What is your opinion? :)

Let's see what happens when we register a new room (Room3) with temperature and pressure:

**XML**

```
(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
  <registerContextRequest>
    <contextRegistrationList>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room3</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
      </contextRegistration>
    </contextRegistrationList>
  </registerContextRequest>
```

<name>pressure</name>
<type>mmHg</type>
<isDomain>false</isDomain>
</contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rs</providingApplication>

</contextRegistration>
</contextRegistrationList>
<duration>P1M</duration>
</registerContextRequest>

EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d '@- | python -m json.tool') <<EOF
{
    "contextRegistrations": [
    {
        "entities": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room3"
        }
    ],
    "attributes": [
    {
        "name": "temperature",
        "type": "centigrade",
        "isPattern": "false"
    },
},
EOF
As expected, the accumulator-server.py will be notified of the new registration. Again, although Room3 registration includes temperature and pressure, only the first attribute is included in the notification.

XML

```xml
POST http://localhost:1028/accumulate
Content-Length: 840
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/xml

<notifyContextAvailabilityRequest>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room3</id>
          </entityId>
        </entityIdList>
        <duration> "P1M"
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</notifyContextAvailabilityRequest>
```
<entityIdList>
<contextRegistrationAttributeList>
<contextRegistrationAttribute>
  <name>temperature</name>
  <type>centigrade</type>
  <isDomain>false</isDomain>
</contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rooms</providingApplication>

</contextRegistration>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</notifyContextAvailabilityRequest>

**JSON (since release 0.9.0)**

POST http://localhost:1028/accumulate
Content-Length: 522
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/json

{
  "subscriptionId" : "52a745e011f5816465943d59",
  "contextRegistrationResponses" : [
    {
      "contextRegistration" : {
        "entities" : [
          {
            "type" : "Room",
            "isPattern" : "false",
          }
        ]
      }
    }
  ]
}
We can also check that context registrations not matching the subscription doesn’t trigger any notifications. For example, let’s register a room (Room4) with only attribute pressure (remember that the subscription only includes temperature in attributeList).

```
<?xml version="1.0"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="Room" isPattern="false">
          <id>Room4</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          "id" : "Room3"
        }
      ],
      "attributes" : [
        {
          "name" : "temperature",
          "type" : "centigrade",
          "isDomain" : "false"
        }
      ],
      "providingApplication" : "http://mysensors.com/Rooms"
    }
  }
</registerContextRequest>
```

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="Room" isPattern="false">
          <id>Room4</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
<name>pressure</name>
<type>mmHg</type>
<isDomain>false</isDomain>
</contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rooms</providingApplication>

</contextRegistration>
</contextRegistrationList>
<duration>P1M</duration>
</registerContextRequest>

EOF

---

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
    "contextRegistrations": [
    {
        "entities": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room4"
        }
    ],
    "attributes": [
    {
        "name": "pressure",
        "type": "mmHg",
        "isDomain": "false"
    }
    ]
}
You can now check that no new notification arrives to accumulator-server.py.

As with context subscriptions, context availability subscriptions can be updated (using the NGSI9 updateContextAvailabilitySubscription). The request includes the subscriptionId that identifies the subscription to modify, and the actual update payload. For example, let’s change subscription entities to something completely different: cars instead of rooms and all the attributes are removed (i.e. an empty attributeList element). As always you have to replace the subscriptionId value after copy-pasting with the value you got from the subscribeContextAvailability response in the previous step).

XML

(curl localhost:1026/NGSI9/updateContextAvailabilitySubscription -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - )
<<EOF
<?xml version="1.0"?>
<updateContextAvailabilitySubscriptionRequest>
  <entityIdList>
    <entityId type="Car" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList/>
  <duration>P1M</duration>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
</updateContextAvailabilitySubscriptionRequest>
EOF

JSON (since release 0.9.0)
(curl localhost:1026/NGSI9/updateContextAvailabilitySubscription -s -S
--header "Content-Type: application/json" --header "Accept: application/json" -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Car",
      "isPattern": "true",
      "id": ".*"
    }
  ],
  "duration": "P1M",
  "subscriptionId": "52a745e011f5816465943d59"
}
EOF

The response is very similar to the one for subscribeContextAvailability request:

XML

<?xml version="1.0"?>
<updateContextAvailabilitySubscriptionResponse>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
  <duration>P1M</duration>
</updateContextAvailabilitySubscriptionResponse>

JSON (since release 0.9.0)

{
  "duration": "P1M",
  "subscriptionId": "52a745e011f5816465943d59"
}
Given that there are currently no car entities registered, you will not receive any initial notification. So, let's register two cars: Car1 with an attribute named speed and Car2 with an attribute named location (don't worry about the ISO6709 reference in the XML, it's just standard for geo-location and nothing significant for this tutorial).

XML

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="Car" isPattern="false">
          <id>Car1</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>speed</name>
          <type>km/h</type>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
      <providingApplication>http://mysensors.com/Cars</providingApplication>
    </contextRegistration>
  </contextRegistrationList>
  <duration>P1M</duration>
</registerContextRequest>
EOF

JSON (since release 0.9.0)
(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
   "contextRegistrations": [ 
   {
      "entities": [ 
         {
            "type": "Car",
            "isPattern": "false",
            "id": "Car1"
         }
      ],
      "attributes": [ 
         {
            "name": "speed",
            "type": "km/h",
            "isDomain": "false"
         }
      ],
      "providingApplication": "http://mysensors.com/Cars"
   }
   ],
   "duration": "P1M"
}
EOF

 XML

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<registerContextRequest>
  <contextRegistrationList>
<contextRegistration>
  <entityIdList>
    <entityId type="Car" isPattern="false">
      <id>Car2</id>
    </entityId>
  </entityIdList>
  <contextRegistrationAttributeList>
    <contextRegistrationAttribute>
      <name>location</name>
      <type>ISO6709</type>
      <isDomain>false</isDomain>
    </contextRegistrationAttribute>
  </contextRegistrationAttributeList>
  <providingApplication>http://mysensors.com/Cars</providingApplication>
</contextRegistration>

<providingApplication>http://mysensors.com/Cars</providingApplication>
<duration>P1M</duration>
</registerContextRequest>

EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextRegistrations": [
    {
      "entities": [
        {
          "type": "Car",
          "isPattern": "false",
          "id": "Car2"
        }
      ]
    }
  ]
}

EOF
As both registrations match the entityIdList and attributeList used in the updateContextAvailabilitySubscription, we will get a notification for each car registration, as can be seen in accumulator-server.py:

```
<notifyContextAvailabilityRequest>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
              
```

POST http://localhost:1028/accumulate
Content-Length: 825
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/xml

<notifyContextAvailabilityRequest>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          
EOF
Future Internet Core Platform

<entityId type="Car" isPattern="false">
  <id>Car1</id>
</entityId>

<contextRegistrationAttributeList>
  <contextRegistrationAttribute>
    <name>speed</name>
    <type>km/h</type>
    <isDomain>false</isDomain>
  </contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Cars</providingApplication>

JSON (since release 0.9.0)

POST http://localhost:1028/accumulate
Content-Length: 529
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/json

{
  "subscriptionId" : "52a745e011f5816465943d59",
  "contextRegistrationResponses" : [
    
    "contextRegistration" : {
      "entities" : [
{
    "type" : "Car",
    "isPattern" : "false",
    "id" : "Car1"
},

"attributes" : [
    {
        "name" : "speed",
        "type" : "km/h",
        "isDomain" : "false"
    }
],

"providingApplication" : "http://mysensors.com/Cars"
}

---

XML

POST http://localhost:1028/accumulate
Content-Length: 831
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/xml

<notifyContextAvailabilityRequest>
    <subscriptionId>52a745e011f5816465943d59</subscriptionId>
    <contextRegistrationResponseList>
        <contextRegistrationResponse>
            <contextRegistration>
                <entityIdList>
<entityId type="Car" isPattern="false">
  <id>Car2</id>
</entityId>

<contextRegistrationAttributeList>
  <contextRegistrationAttribute>
    <name>location</name>
    <type>ISO6709</type>
    <isDomain>false</isDomain>
  </contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Cars</providingApplication>

<notifyContextAvailabilityRequest>

---

**JSON** *(since release 0.9.0)*

POST http://localhost:1028/accumulate
Content-Length: 535
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/json

{
  "subscriptionId": "52a745e011f5816465943d59",
  "contextRegistrationResponses": [
    {
      "contextRegistration": {
        "entities": [
          
        ]
      }
    }
  ]
}
Finally, you can cancel a subscription using the NGSI9 unsubscribeContextAvailability operation, just using the subscriptionId in the request payload (replace the subscriptionId value after copy-pasting with the one you received in the subscribeContextAvailability response in the previous step).

**XML**

```
(curl localhost:1026/NGSI9/unsubscribeContextAvailability -s -S --
header 'Content-Type: application/xml' -d @- | xmllint --format - )
<<EOF
<?xml version="1.0"?>
<unsubscribeContextAvailabilityRequest>
    <subscriptionId>52a745e011f5816465943d59</subscriptionId>
</unsubscribeContextAvailabilityRequest>
EOF
```

**JSON (since release 0.9.0)**
The response is just an acknowledgement that the cancellation was successful.

XML

```xml
<?xml version="1.0"?>
<unsubscribeContextAvailabilityResponse>
  <subscriptionId>52a745e011f5816465943d59</subscriptionId>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
</unsubscribeContextAvailabilityResponse>
```

JSON (since release 0.9.0)

```json
{
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  },
  "subscriptionId": "52a745e011f5816465943d59"
}
```
After cancelling, you can try to register a new car (e.g. Car3) to check that no new notification is sent to accumulator-server.py.

2.3.3.1.4 Summary of NGSI9 standard operations URLs
Each standard operation has a unique URL. All of them use the POST method. The summary is below:

- `<host:port>/ngsi9/registerContext`
- `<host:port>/ngsi9/discoverContextAvailability`
- `<host:port>/ngsi9/subscribeContextAvailability`
- `<host:port>/ngsi9/updateContextAvailabilitySubscription`
- `<host:port>/ngsi9/unsubscribeContextAvailability`

2.3.3.2 Tutorial on NGSI9 convenience operations
The following section describes the different convenience operations described as part of the FI-WARE NGSI REST API NGSI9 that Orion Context Broker supports, showing examples of requests and responses. Convenience operations are a set of operations that have been defined by FI-WARE project to ease the usage of NGSI implementations as a complement to the standard operations defined in the OMA NGSI specification (see the section on additional information later in this manual).

**Don't forget to restart the broker before starting this tutorial as described previously in this document.**

At the end of this section, you will have learnt to use convenience operations as a handy alternative to some standard operations described in the previous section. It is highly recommended to do that tutorial before, to get familiar with register, discover, etc. to be able to compare between the two approaches.

2.3.3.2.1 Convenience Register Context
First of all, we register Room1 and Room2 with attributes temperature and pressure, using the following commands:

```
(curl localhost:1026/ngsi9/contextEntities/Room1/attributes/temperature -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format -) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>
</registerProviderRequest>
EOF
```
<providingApplication>http://mysensors.com/Rooms</providingApplication>
</registerProviderRequest>
EOF

**JSON (since release 0.10.0)**

(curl localhost:1026/NGSI9/contextEntities/Room1/attributes/temperature -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) << EOF
{
  "duration" : "P1M",
  "providingApplication" : "http://mysensors.com/Rooms"
}
EOF

**XML**

(curl localhost:1026/ngsi9/contextEntities/Room1/attributes/pressure -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>
  <providingApplication>http://mysensors.com/Rooms</providingApplication>
</registerProviderRequest>
EOF

**JSON (since release 0.10.0)**

(curl localhost:1026/NGSI9/contextEntities/Room1/attributes/pressure -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) << EOF
{  
  "duration" : "P1M",
  "providingApplication" : "http://mysensors.com/Rooms"
}
EOF

---

XML

(curl
localhost:1026/ngsi9/contextEntities/Room2/attributes/temperature -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>

<providingApplication>http://mysensors.com/Rooms</providingApplication>
</registerProviderRequest>
EOF

---

JSON (since release 0.10.0)

(curl
localhost:1026/NGSI9/contextEntities/Room2/attributes/temperature -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) << EOF
{
  "duration" : "P1M",
  "providingApplication" : "http://mysensors.com/Rooms"
}
EOF

---

XML
(curl localhost:1026/ngsi9/contextEntities/Room2/attributes/pressure -s -S --header "Content-Type: application/xml" -d @- | xmllint --format - ) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>
  <providingApplication>http://mysensors.com/Rooms</providingApplication>
</registerProviderRequest>
EOF

JSON (since release 0.10.0)

(curl localhost:1026/NGSI9/contextEntities/Room2/attributes/pressure -s -S --header "Content-Type: application/json" --header "Accept: application/json" -d @- | python -mjson.tool) << EOF
{
  "duration": "P1M",
  "providingApplication": "http://mysensors.com/Rooms"
}
EOF

So, what's the difference compared to standard registerContext operation?

- We needed four requests, instead of just one request in the standard operation case.
- We are using more operations, but the payload used in each operation is much simpler. This payload is a simplified version of the payload in registerContext, including only duration and providing application.
- Type cannot be registered. Thus, we cannot specify whether temperature is in centigrades or Fahrenheit degrees or if "Room1" is in type "Room" or "Space". This lack of typing has some important implications.
- From the Orion Context Broker perspective, there are 4 independent registrations (i.e. 4 different registration IDs) to all effects (e.g. updating, extending duration).
- It is possible to use /ngsi9/contextEntities/Room1 (without the attribute part). In that case, you are registering an entity without attributes (whatever it means in the context of your application :). Note you cannot specify attributes in the registerProviderRequest element.
The response to each of these requests is the same as the response to a standard registerContext (one response for each of the four requests, with a different ID):

```
XML

<?xml version="1.0"?>
<registerContextResponse>
  <registrationId>51c1f5c31612797e4fe6b6b6</registrationId>
  <duration>P1M</duration>
</registerContextResponse>

JSON (since release 0.10.0)

{
  "duration": "P1M",
  "registrationId": "51c1f5c31612797e4fe6b6b6"
}
```

2.3.3.2.2  Convenience Discover Context Availability

Using convenience operations you can discover registration information for a single entity or for an entity-attribute pair. For example, to discover registrations for Room1 (no matter the attributes):

```
XML

curl localhost:1026/ngsi9/contextEntities/Room1 -s -S --header 'Content-Type: application/xml' | xmllint --format -

JSON (since release 0.10.0)

curl localhost:1026/NGSI9/contextEntities/Room1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

which produces the following response:
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="" isPattern="false">
            <id>Room1</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type/>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Rooms</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
<isDomain>false</isDomain>
</contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rooms</providingApplication>
</contextRegistration>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</discoverContextAvailabilityResponse>

**JSON (since release 0.10.0)**

```json
{
    "contextRegistrationResponses": [
    {
        "contextRegistration": {
            "attributes": [
            {
                "isDomain": "false",
                "name": "temperature",
                "type": ""
            }
            ],
            "entities": [
            {
                "id": "Room1",
                "isPattern": "false",
                "type": ""
            }
            ],
            "providingApplication": "http://mysensors.com/Rooms"
        }
    },
```
Now, let's discover registrations for Room2-temperature:

**XML**

```
curl localhost:1026/ngsi9/contextEntities/Room2/attributes/temperature
-s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```
curl localhost:1026/NGSI9/contextEntities/Room2/attributes/temperature
-s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -m json.tool
```
The response is as follows:

**XML**

```xml
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="" isPattern="false">
            <id>Room2</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type/>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Rooms</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
```

**JSON (since release 0.10.0)**

```json
{
   "contextRegistrationResponses": [
    {
```
"contextRegistration": {
  "attributes": [
    {
      "isDomain": "false",
      "name": "temperature",
      "type": ""
    }
  ],
  "entities": [
    {
      "id": "Room2",
      "isPattern": "false",
      "type": ""
    }
  ],
  "providingApplication": "http://mysensors.com/Rooms"
}
}
}

Discovery of not registered elements (e.g. Room5 or the humidity of Room1) will produce an error. E.g. the following requests:

**XML**

```bash
curl localhost:1026/ngsi9/contextEntities/Room2/attributes/humidity -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```bash
curl localhost:1026/NGSI9/contextEntities/Room2/attributes/humidity -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```
will produce the following error response:

**XML**

```xml
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <errorCode>
    <code>404</code>
    <reasonPhrase>No context element found</reasonPhrase>
  </errorCode>
</discoverContextAvailabilityResponse>
```

**JSON (since release 0.10.0)**

```json
{
  "errorCode": {
    "code": "404",
    "reasonPhrase": "No context element found"
  }
}
```

Compared to **standard discoverContextAvailability operation**:

- Convenience operations use the GET method without needing any payload in the request (simpler than the standard operation)

- The structure of the response XML (i.e. discoverContextAvailabilityResponse) is the same in both cases. However, there are two differences in the content. First, each attribute always comes in a different contextRegistrationResponse element (as each attribute corresponds to a different registration, as explained before). Secondly, as registrations done using convenience operations aren't typed, the type fields are empty for entities and attributes.

- It is not possible to use convenience operations to discover lists of entities, entity patterns, nor lists of attributes.

**Since release 0.9.0 (FI-WARE 3.2.3)**: you can also discover by all the entities belonging to the same type, either all the attributes or a particular one, as shown below. First, register an couple of entities of type
Car using standard registerContext operations (given that, as described in previous section, you cannot register entities with types using convenience operations):

**XML**

```
(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
    <registerContextRequest>
        <contextRegistrationList>
            <contextRegistration>
                <entityIdList>
                <entityId type="Car" isPattern="false">
                    <id>Car1</id>
                </entityId>
                </entityIdList>
                <contextRegistrationAttributeList>
                <contextRegistrationAttribute>
                    <name>speed</name>
                    <type>km/h</type>
                    <isDomain>false</isDomain>
                </contextRegistrationAttribute>
                </contextRegistrationAttributeList>
            <providingApplication>http://mysensors.com/Cars</providingApplication>
        </contextRegistration>
        </contextRegistrationList>
        <duration>P1M</duration>
    </registerContextRequest>
EOF
```

**JSON (since release 0.9.0)**

```
(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
```
{"contextRegistrations": [
{
 "entities": [
{
 "type": "Car",
 "isPattern": "false",
 "id": "Car1"
 }
],
 "attributes": [
{
 "name": "speed",
 "type": "km/h",
 "isDomain": "false"
 }
],
 "providingApplication": "http://mysensors.com/Cars"
 }
],
 "duration": "P1M"
}

EOF

XML

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
 <registerContextRequest>
   <contextRegistrationList>
     <contextRegistration>
       <entityIdList>
         <entityId type="Car" isPattern="false"/>
<id>Car2</id>
</entityId>
</entityIdList>
<contextRegistrationAttributeList>
<contextRegistrationAttribute>
  <name>fuel</name>
  <type>liters</type>
  <isDomain>false</isDomain>
</contextRegistrationAttribute>
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Cars</providingApplication>
</contextRegistration>
</contextRegistrationList>
<duration>P1M</duration>
</registerContextRequest>

---

### JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextRegistrations": [
  {
    "entities": [
    {
      "type": "Car",
      "isPattern": "false",
      "id": "Car2"
    }
  ],
  "attributes": [
{  
    "name": "fuel",
    "type": "liters",
    "isDomain": "false"
}
]

"providingApplication": "http://mysensors.com/Cars"
}
]

"duration": "P1M"
}
EOF

Request without specifying attributes:

```
XML
```

```bash
curl localhost:1026/ngsi9/contextEntityTypes/Car -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON (since release 0.10.0)**

```bash
curl localhost:1026/ngsi9/contextEntityTypes/Car -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

Response:

```
XML
```

```xml
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
   <contextRegistrationResponseList>
   <contextRegistrationResponse>
```

D.6.4.3: FI-WARE Users and Programmers Guide
<contextRegistration>
  <entityIdList>
    <entityId type="Car" isPattern="false">
      <id>Car1</id>
    </entityId>
  </entityIdList>
  <contextRegistrationAttributeList>
    <contextRegistrationAttribute>
      <name>speed</name>
      <type>km/h</type>
      <isDomain>false</isDomain>
    </contextRegistrationAttribute>
  </contextRegistrationAttributeList>
  <providingApplication>http://mysensors.com/Cars</providingApplication>
</contextRegistration>

<contextRegistrationResponse>
  <contextRegistration>
    <entityIdList>
      <entityId type="Car" isPattern="false">
        <id>Car2</id>
      </entityId>
    </entityIdList>
    <contextRegistrationAttributeList>
      <contextRegistrationAttribute>
        <name>fuel</name>
        <type>liters</type>
        <isDomain>false</isDomain>
      </contextRegistrationAttribute>
    </contextRegistrationAttributeList>
    <providingApplication>http://mysensors.com/Cars</providingApplication>
  </contextRegistration>
</contextRegistrationResponse>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</discoverContextAvailabilityResponse>

---

JSON (since release 0.10.0)

{
  "contextRegistrationResponses": [
  {
    "contextRegistration": {
      "attributes": [
      {
        "isDomain": "false",
        "name": "speed",
        "type": "km/h"
      }
      ],
      "entities": [
      {
        "id": "Car1",
        "isPattern": "false",
        "type": "Car"
      }
      ],
      "providingApplication": "http://mysensors.com/Cars"
    }
  },
  {
    "contextRegistration": {
      "attributes": [
      {
        "isDomain": "false",
        "name": "fuel",
        "type": "fuel"
      }
      ],
      "entities": [
      {
        "id": "Car2",
        "isPattern": "false",
        "type": "Car"
      }
      ],
      "providingApplication": "http://mysensors.com/Cars"
    }
  }
]}
"type": "liters"
}
],
"entities": [
{
"id": "Car2",
"isPattern": "false",
"type": "Car"
}
],
"providingApplication": "http://mysensors.com/Cars"
}
)}
]

Request specifying one attribute (e.g. speed):

XML

```
curl localhost:1026/ngsi9/contextEntityTypes/Car/attributes/speed -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

JSON (since release 0.10.0)

```
curl localhost:1026/ngsi9/contextEntityTypes/Car/attributes/speed -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -mjson.tool
```

Response:

XML

```
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
```
<contextRegistrationResponseList>
  <contextRegistrationResponse>
    <contextRegistration>
      <entityIdList>
        <entityId type="Car" isPattern="false">
          <id>Car1</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>speed</name>
          <type>km/h</type>
          <isDomain>false</isDomain>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
    </contextRegistration>
  </contextRegistrationResponse>
</contextRegistrationResponseList>

<providingApplication>http://mysensors.com/Cars</providingApplication>
</contextRegistrationResponse>
</contextRegistrationResponseList>
</discoverContextAvailabilityResponse>

---

JSON (since release 0.10.0)

{
  "contextRegistrationResponses": [
    {
      "contextRegistration": {
        "attributes": [
          {
            "isDomain": "false",
            "name": "speed",
            "type": "km/h"
          }
        ]
      }
    }
  ]
}
Note that by default only 20 registrations are returned (which is fine for this tutorial, but probably not for a real utilization scenario). In order to change this behaviour, see the section on pagination in this manual.

2.3.3.2.3 **Convenience operations for context availability subscriptions**

Since release 0.9.0 (FI-WARE 3.2.3): new functionality.

You can use the following convenience operations to manage context availability subscriptions:

- **POST /ngsi9/contextAvailabilitySubscriptions**, to create the subscription, using the same payload as standard subscribeAvailabilityContext operation.

- **PUT /ngsi9/contextAvailabilitySubscriptions/{subscriptionID}**, to update the subscription identified by {subscriptionID}, using the same payload as standard updateContextAvailabilitySubscription operation. The ID in the payload must match the ID in the URL.

- **DELETE /ngsi9/contextAvailabilitySubscriptions/{subscriptionID}**, to cancel the subscription identified by {subscriptionID}. In this case, payload is not used.

2.3.3.2.4 **Summary of NGSI9 convenience operations URLs**

Convenience operations use a URL to identify the resource and a HTTP verb to identify the operation on that resource following the usual REST convention: GET is used to retrieve information, POST is used to create new information, PUT is used to update information and DELETE is used to destroy information.

You find a summary in the following document.
2.3.4 Advanced topics

2.3.4.1 Pagination

Since 0.14.0

In order to help clients to organize query and discovery request with a large number of response (for example, thing how costly could be return a query matching 1,000,000 results in a single HTTP response to a queryContext request) queryContext (and related convenience operations) and discoverContextAvailability (and related convenience operations) allow pagination. The mechanism is based on three URI parameters:

- **limit**, in order to specify the maximum number of entities or context registrations (for queryContext and discoverContextAvailability respectively) (default is 20, maximum allowed is 1000).
- **offset**, in order to skip a given number of elements at the beginning (default is 0)
- **details** (allowed values are “on” and “off”, default is “off”), in order to get a global errorCode for the response including a count of total elements (in the case of using “on”). Note that using details “on” slightly breaks NGSI standard, which states that global errorCode must be used only in the case of general error with the request. However, we think it is very useful for a client to know in advance how many results in total the query has (and if you want to keep strict with NGSI, you can simply ignore the details parameter :)

Result are returned in increasing entity/registration creation time, in order to ensure that if a new entity/registration in created while the client is going through all the results, the new results are added at the end (thus avoiding duplication results).

Let’s illustrate with an example. A given client cannot process more than 100 results in a single response and the queryContext includes a total of 322 results. The client could use the following (only URL is included, for the sake of completeness).

```
POST <orion_host>:1026/NGSI10/queryContext?limit=100&details=on
...
(The first 100 elements are returned, along with the following errorCode in the response,
which allows the client to know how many entities are in sum and, therefore, the number of subsequent queries to do)

<errorCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
```
Note that the request uses an “out of bound” offset you will get a 404 NGSI error, as shown below:

2.3.4.2 Custom attribute metadata

Since 0.13.0: new in this release.
Apart from metadata elements to which Orion pays special attention (e.g. ID, location, etc.), users can attach their own metadata to entity attributes. These metadata elements are processed by Orion in a...
transparent way: it simply stores them in the database at updateContext (and notifyContext time in federated scenarios) and retrieve it in queryContext or notifyContext.

For example, to create an entity Room1, with attribute "temperature" and associate metadata "accuracy" to "temperature":

```xml
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue>26.5</contextValue>
          <metadata>
            <contextMetadata>
              <name>accuracy</name>
              <type>float</type>
              <value>0.8</value>
            </contextMetadata>
          </metadata>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

<updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

JSON

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "contextElements": [

  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1",
    "attributes": [

      {
        "name": "temperature",
        "type": "centigrade",
        "value": "26.5",
        "metadatas": [

          {
            "name": "accuracy",
            "type": "float",
            "value": "0.8"
          }
        ]
      }

  ]
}
}
EOF
Metadata can be updated regardless of the attribute value being updated or not. For example, next updateContext shows how "accuracy" is updated to 0.9 although the value of the temperature itself is still 26.5:

XML

```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue>26.5</contextValue>
          <metadata>
            <contextMetadata>
              <name>accuracy</name>
              <type>float</type>
              <value>0.9</value>
            </contextMetadata>
          </metadata>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

EOF
```json
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "contextElements": [
  
  
  "type": "Room",
  "isPattern": "false",
  "id": "Room1",
  "attributes": [
  
  "name": "temperature",
  "type": "centigrade",
  "value": "26.5",
  "metadatas": [
  
  "name": "accuracy",
  "type": "float",
  "value": "0.9"
```

```json"
Metadata can be added after attribute creation. For example, if we want to add metadata "average" to "temperature" (in addition to the existing "precision"):

XML

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList> 
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room1</id>
      </entityId>
      <contextAttributeList> 
        <contextAttribute>
          <name>temperature</name>
          <type>centigrade</type>
          <contextValue>26.5</contextValue>
        <metadata>
          <contextMetadata>
<name>average</name>
<type>centigrade</type>
<value>22.4</value>
</contextMetadata>
</metadata>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>UPDATE</updateAction>
</updateContextRequest> EOF

---

**JSON**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1",
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": "26.5",
      "metadatas": []
    }
  }
}

EOF
We can check that temperature includes both attributes

```
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    
```

```xml
<name>"average"</name>,
<type>"centigrade"</type>,
:value>"22.4"

EOF
```

```
curl localhost:1026/ngsi9/contextEntities/Room1 -s -S | xmllint --format -
```
<entityId type="" isPattern="false">
  <id>Room1</id>
</entityId>
<contextAttributeList>
  <contextAttribute>
    <name>temperature</name>
    <type>centigrade</type>
    <contextValue>26.5</contextValue>
    <metadata>
      <contextMetadata>
        <name>average</name>
        <type>centigrade</type>
        <value>22.4</value>
      </contextMetadata>
      <contextMetadata>
        <name>accuracy</name>
        <type>float</type>
        <value>0.9</value>
      </contextMetadata>
    </metadata>
  </contextAttribute>
</contextAttributeList>
<contextElement>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
</contextElementResponse>

JSON
{
  "contextElements": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1",
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": "26.5",
      "metadatas": [
      {
        "name": "average",
        "type": "centigrade",
        "value": "22.4"
      },
      {
        "name": "accuracy",
        "type": "float",
        "value": "0.9"
      }
      ]
    }
  ]
  }
}
Note that, from the point of view of ONCHANGE subscription, changing the metadata of a given attribute or adding a new metadata element is considered a change even if attribute value itself hasn't changed. Metadata elements cannot be deleted once introduced: in order to delete metadata elements you have to remove the entity attribute (using updateContext DELETE), then re-create it (using updateContext APPEND).

You can use any name for your custom metadata except for the ones used for some metadata names that are interpreted by Orion:

- **ID**
- **location**
- **creData** (reserved for future use)
- **modDate** (reserved for future use)

### 2.3.4.3 Default duration

**Release Note (0.6.0 FIWARE 3.1.1):** new in this release.

If you don't specify a duration in registerContext, subscribeContext or subscribeContextAvailability a default of 24 hours is used. You will get a confirmation of the duration in these cases in the response, e.g. for a registerContext:

**XML**

```xml
<?xml version="1.0"?>
<registerContextResponse>
  <registrationId>51bf1e0ada053170df590f20</registrationId>
  <duration>PT24H</duration>
</registerContextResponse>
```

**JSON (since release 0.9.0)**

```javascript
{ "code": "200", "reasonPhrase": "OK" };
```
2.3.4.4  Mixing standard and convenience operations

Although the tutorials in this manual introduce standard and convenience operations independently for the sake of clarity, you can mix their use without any problem. Note that the set of URLs used by standard operations (NGSI10 and NGSI9) and the set of URLs used by convenience operations (NGSI10 and NGSI9) are orthogonal.

However, take into account that most convenience operations don’t allow to specify any type for entities nor for attributes, so the rules described in using empty types section always apply.

We propose the following exercise: register entities/attributes using standard operations as described in the standards operations tutorial, then discover these attributes using convenience operations.

2.3.4.5  The actual meaning of a Providing Application

The register context operation (both in standard and convenience cases) uses a field named "providing application" which is basically a URL that identifies the source of the context information for the entities/attributes included in that registration:

**XML**

```
...
<providingApplication>http://mysensors.com/Rooms</providingApplication>
...
```

**JSON (since release 0.9.0)**

```
...
"providingApplication" : "http://mysensors.com/Rooms"
...
```
Some questions may arise about this field. For example, how does the Orion Context Broker use it? Does it force to run the context producer application (i.e. the application issuing update context operations) in mysensor.com host? Is the broker accessing directly to that URL in some case? Let’s try to clarify this a bit.

Basically, the current version of Orion Context Broker just stores this field in the database. Thus, applications can access this URL using the discover context availability operation and do whatever they want with it. This is typically the case when the Orion Context Broker is used just as a repository for NGSI9 registrations, but the actual management of context information is done by other components of the architecture. In FI-WARE, this is the case for Orion Context Broker running as Configuration Management GE. In this case, the IoT Broker GE is the stateless component that implements NGSI10 and issues discover context availability requests to the Orion Context Broker (acting as NGSI9 repository) to be informed about the whereabouts of the sources of context information that it needs to process NGSI10 requests.

There is no relation between the providing application URL in a registration and the applications that can issue update context requests on the entities/attributes of that registration. In fact, any application can do update context operations no matter where it runs as long as it has connectivity to the host and port where the Orion Context Broker is running (more information on this on the security considerations section of this manual).

Finally, the Orion Context Broker doesn’t access the providing application URL directly under any circumstances in its current version. In fact, you can use even "fake" providing applications.

**Release Note (any version):** however, in next versions this may change, in the following way:
- The Orion Context Broker may periodically check providing application URLs to update values for attribute entities without needing explicit update context operations from applications.
- Upon reception of a query context operation on an attribute whose value is currently unknown (or obsolete if the last update was done a long time ago), the Orion Context Broker may use the providing application URL to issue a queryContext and get the current value. This is what we call "uncached queries".

### 2.3.4.6 Updating registrations

The response to a register context request (both in standard and convenience) includes a registration ID (a 24 hexadecimal digit number):

```xml
<?xml version="1.0"?>
<registerContextResponse>
  <registrationId>51bf1e0ada053170df590f20</registrationId>
</registerContextResponse>
```
This ID can be used to update the registration. There is no special operation to update a registration (in this sense, it is different from context subscriptions and context availability subscriptions, which have updateContextSubscription and updateContextAvailabilitySubscription operations). The update is done issuing a new registerContextRequest, with the registrationId set:

```
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="Room" isPattern="false">
          <id>Room8</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>humidity</name>
          <type>percentage</type>
          <isDomain>false</isDomain>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
    </contextRegistration>
  </contextRegistrationList>
</registerContextRequest>

```
</contextRegistrationAttributeList>

<providingApplication>http://mysensors.com/Rooms</providingApplication>

</contextRegistration>
</contextRegistrationList>
<duration>P1M</duration>
<registrationId>51bf1e0ada053170df590f20</registrationId>
</registerContextRequest>
EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI9/registerContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextRegistrations": [
    {
      "entities": [
        {
          "type": "Room",
          "isPattern": "false",
          "id": "Room8"
        }
      ],
      "attributes": [
        {
          "name": "humidity",
          "type": "percentage",
          "isDomain": "false"
        }
      ],
      "providingApplication": "http://mysensors.com/Rooms"
    }
  ]
}
EOF
This "update registration" replaces the existing registration associated to that ID with the new content, including expiration recalculation.

Surprisingly, there is no way in NGSI to cancel a registration. A workaround is to update it with a non-existing entity and duration 0, but in order to do an actual delete you will need to remove the registration from the database (check the administration manual about managing database).

2.3.4.7 Updating subscriptions

You have previously seen in this document that context subscriptions and context availability subscriptions can be updated. However, differently from registerContext, not everything can be updated. Let's look at this closely, depending on the type of subscription.

2.3.4.7.1 What can be updated in a context subscription?

The payload for updateContextSubscription is similar to the one for a subscribeContext request. However, not all the fields can be included, as not everything can be updated. In particular, the following fields cannot be updated:

- subscriptionId (although you must include it in updateContextSubscription to refer to the subscription)
- entityIdList
- attributeList
- reference

However, the following fields can be modified:

- notifyConditions
- throttling
- duration

2.3.4.7.2 What can be updated in a context availability subscription?

The payload used by an updateContextAvailabilitySubscription is pretty similar to the one in the subscribeContextAvailability request. However, not all the fields can be included, as not everything is updatable. In particular, the following is not updatable:
subscriptionId (although you must include it in updateContextAvailabilitySubscription to refer to the subscription)

- reference

Thus, the following is updatable:

- entityIdList (in fact, it is mandatory in every updateContextAvailabilitySubscription).
- attributeList
- duration

### 2.3.4.8 Adding and removing attributes with APPEND and DELETE in updateContext

We have seen how to use updateContext with APPEND action type to create new entities. In addition, APPEND can be used to add a new attribute after entity creation. Let's illustrate this with an example.

We start creating a simple entity 'E1' with one attribute named 'A':

```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>A</name>
          <type>TA</type>
          <contextValue>1</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

```
Now, in order to append a new attribute (let’s name it 'B') we use updateContext APPEND with an entityId matching 'E1':

```json
{$contextElements: [ {
    "type": "T",
    "isPattern": "false",
    "id": "E1",
    "attributes": [ {
        "name": "A",
        "type": "TA",
        "value": "1"
    } ]
  },
  "updateAction": "APPEND"
]}
EOF
```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>B</name>
          <type>TB</type>
          <contextValue>2</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

(JSON (since release 0.9.0))

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
Now we can check with a query to that entity that both attributes A and B are there:

```xml
curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

**JSON** *(since release 0.10.0)*

```bash
curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -m json.tool
```
<?xml version="1.0"?>
<contextElementResponse>
<contextElement>
    <entityId type="" isPattern="false">
        <id>E1</id>
    </entityId>
    <contextAttributeList>
        <contextAttribute>
            <name>A</name>
            <type>TA</type>
            <contextValue>1</contextValue>
        </contextAttribute>
        <contextAttribute>
            <name>B</name>
            <type>TB</type>
            <contextValue>2</contextValue>
        </contextAttribute>
    </contextAttributeList>
    <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
    </statusCode>
</contextElement>
</contextElementResponse>

**JSON (since release 0.9.0)**
We can also remove attributes in a similar way, using the DELETE action type. For example, to remove attribute 'A' we will use (note the empty contextValue element):

```
XML
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
```
Future Internet Core Platform

```xml
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>A</name>
          <type>TA</type>
          <contextValue/>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>DELETE</updateAction>
</updateContextRequest>
EOF
```

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
    "contextElements": [
        {
            "type": "T",
            "isPattern": "false",
```
Now, a query to the entity shows attribute B:

XML

curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/xml' | xmllint --format -

JSON (since release 0.10.0)

curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' | python -m json.tool

XML

<?xml version="1.0"?>
<contextElementResponse>
<contextElement>
  <entityId type="" isPattern="false">
    <id>E1</id>
  </entityId>
  <contextAttributeList>
    <contextAttribute>
      <name>B</name>
      <type>TB</type>
      <contextValue>2</contextValue>
    </contextAttribute>
  </contextAttributeList>
</contextElement>

<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>

JSON (since release 0.9.0)

```json
{
  "contextElement": {
    "attributes": [
      {
        "name": "B",
        "type": "TB",
        "value": "2"
      }
    ]
  }
}
```
You can also use convenience operations with POST and DELETE verbs to add and delete attributes. Try the following:

Add a new attribute 'C' and 'D':

```
XML
(curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/xml' -X POST -d @- | xmllint --format -) << EOF
<?xml version="1.0" encoding="UTF-8"?>
<appendContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>C</name>
      <type>TC</type>
      <contextValue>3</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>D</name>
      <type>TD</type>
      <contextValue>4</contextValue>
    </contextAttribute>
  </contextAttributeList>
</appendContextElementRequest>
```
```
</appendContextElementRequest>

EOF

JSON (since release 0.10.0)

(curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF

{
    "attributes": [
    {
        "name": "C",
        "type": "TC",
        "value": "3"
    },
    {
        "name": "D",
        "type": "TD",
        "value": "4"
    }
    ]
}
EOF

Remove attribute 'B':

XML

curl localhost:1026/NGSI10/contextEntities/E1/attributes/B -s -S --header 'Content-Type: application/xml' -X DELETE | xmllint --format -
### JSON (since release 0.10.0)

```bash
curl localhost:1026/NGSI10/contextEntities/E1/attribute/B -s -S --header 'Content-Type: application/json' -X DELETE --header 'Accept: application/json' | python -mjson.tool
```

Query entity (should see 'C' and 'D', but not 'B'):

```xml
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    <entityId type="" isPattern="false">
      <id>E1</id>
    </entityId>
    <contextAttributeList>
      <contextAttribute>
        <name>C</name>
        <type>TC</type>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
</contextElementResponse>
```

### JSON (since release 0.10.0)

```bash
curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

```xml
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    <entityId type="" isPattern="false">
      <id>E1</id>
    </entityId>
    <contextAttributeList>
      <contextAttribute>
        <name>C</name>
        <type>TC</type>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
</contextElementResponse>
```
<contextValue>3</contextValue>
</contextAttribute>
<contextAttribute>
  <name>D</name>
  <type>TD</type>
  <contextValue>4</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>

---

**JSON (since release 0.10.0)**

```json
{
  "contextElement": {
    "attributes": [
      {
        "name": "C",
        "type": "TC",
        "value": "3"
      },
      {
        "name": "D",
        "type": "TD",
        "value": "4"
      }
    ]
  }
}
```
2.3.4.9  Deleting entities

New since version 0.11.0: for older versions you need to use a database procedure to delete entities. Apart from deleting individual attributes from a given entity (see previous section on that topic), you can also delete an entire entity, including all its attributes with their corresponding metadata. In order to do so, the updateContext operation is used, with DELETE as actionType and with an empty attributeList, as in the following example:

XML

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T" isPattern="false">
        <id>E1</id>
      </entityId>
    </contextElement>
  </contextElementList>
</updateContextRequest>
<contextElement>
</contextElementList>
<updateAction>DELETE</updateAction>
</updateContextRequest>
EOF

JSON (since release 0.9.0)

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": {
    "contextElement": {
      "type": "T",
      "isPattern": "false",
      "id": "E1"
    }
  },
  "updateAction": "DELETE"
}
EOF

You can also use the following equivalent convenience operation:

XML

curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/xml' -X DELETE

JSON (since release 0.9.0)
curl localhost:1026/NGSI10/contextEntities/E1 -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -X DELETE
<type>string</type>
<value>ground</value>
</contextMetadata>
</metadata>
</contextAttribute>
<contextAttribute>
 <name>temperature</name>
 <type>centigrade</type>
 <contextValue>23.8</contextValue>
 <metadata>
   <contextMetadata>
     <name>ID</name>
     <type>string</type>
     <value>wall</value>
   </contextMetadata>
 </metadata>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

---

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{


"contextElements": [
{
"type": "Room",
"isPattern": "false",
"id": "Room1",
"attributes": [
{
"name": "temperature",
"type": "centigrade",
"value": "23.5",
"metadatas": [
{
"name": "ID",
"type": "string",
"value": "ground"
}
]
}]
},
{
"name": "temperature",
"type": "centigrade",
"value": "23.8",
"metadatas": [
{
"name": "ID",
"type": "string",
"value": "wall"
}
]}
]
Now, we can query for temperature to get both instances:

**XML**

```
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
</queryContextRequest>
EOF
```

**JSON (since release 0.9.0)**

```
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
```
We can update an specific instance (e.g. ground), letting the other untouched:

```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room1</id>
      </entityId>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

EOF
<contextValue>30</contextValue>
<metadata>
  <contextMetadata>
    <name>ID</name>
    <type>string</type>
    <value>ground</value>
  </contextMetadata>
</metadata>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>UPDATE</updateAction>
</updateContextRequest>
EOF

**JSON (since release 0.9.0)**

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1",
    "attributes": [
    {
      "name": "temperature",
    
```
"type": "centigrade",
"value": "30",
"metadatas": [
{

"name": "ID",
"type": "string",
"value": "ground"
}
]
}
]

"updateAction": "UPDATE"
}
EOF

Check it using again queryContext (ground has changed to 30ºC but wall has its initial value of 23.8º C).

To avoid ambiguities, you cannot mix the same attribute with and without ID. The following entity creation will fail:

XML

(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room2</id>
      </entityId>
    </contextElement>
  </contextElementList>
</updateContextRequest>
EOF
<entityId>
<contextAttributeList>
<contextAttribute>
  <name>temperature</name>
  <type>centigrade</type>
  <contextValue>23.5</contextValue>
  <metadata>
    <contextMetadata>
      <name>ID</name>
      <type>string</type>
      <value>ground</value>
    </contextMetadata>
  </metadata>
</contextAttribute>
<contextAttribute>
  <name>temperature</name>
  <type>centigrade</type>
  <contextValue>23.8</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

JSON (since release 0.9.0)
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
    "contextElements": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room1",
            "attributes": [
                {
                    "name": "temperature",
                    "type": "centigrade",
                    "value": "23.5",
                    "metadatas": [
                        {
                            "name": "ID",
                            "type": "string",
                            "value": "ground"
                        }
                    ]
                },
                {
                    "name": "temperature",
                    "type": "centigrade",
                    "value": "23.8"
                }
            ]
        }
    ],
}
```
Finally, you can use also the following convenience operations with attributes using ID metadata:

- GET /ngsi10/contextEntities/Room1/attributes/temperature/ground: to get an specific attribute identified by ID
- PUT /ngsi10/contextEntities/Room1/attributes/temperature/ground (using as payload updateContextElementRequest, as described in a previous section).
- DELETE /ngsi10/contextEntities/Room1/attributes/temperature/ground: to remove an specific attribute identified by ID (see DELETE attribute semantics described in a previous section).

2.3.4.11 Using empty types

You can use empty types in entities and attributes in NGSI9 registerContext. In fact, convenience operations implicitly use empty types in this way.

Moreover, you can use empty entity types in discover context availability or query context operations. In this case, the absence of type is interpreted as "any type".

For example, let's consider having the following context in Orion Context Broker:

- Entity 1:
  - ID: Room1
  - Type: Room
- Entity 2:
  - ID: Room1
  - Type: Space

A discoveryContextAvailability/querycontext using:

```xml
...<entityIdList>
  <entityId type="" isPattern="false">
    <id>Room1</id>
  </entityId>
</entityIdList>
...```

will match both Entity 1 and Entity 2.

Note that type is not used by definition for attributes in discoverContextAvailability and queryContext. The elements in attributeList are always simple strings (matching names), and type is not taken into account. So if you have two attributes with the same name but different types (e.g. temperature-centigrade and temperature-Fahrenheit) both attributes match.
Regarding the updateContext operation, the absence of type in attributes will update all the attributes, which usually is not the desired behaviour. For example, in the above case of temperature-centigrade and temperature-Fahrenheit (for Room8), consider the following updateContext request with empty type for temperature:

```bash
(curl localhost:1026/ngsi10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="Room" isPattern="false">
        <id>Room8</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>temperature</name>
          <type/>
          <contextValue>30</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>Update</updateAction>
</updateContextRequest>
EOF
```

That request will update both temperature-centigrade and temperature-Fahrenheit to "30", as can be checked with a queryContext:

```bash
(curl localhost:1026/ngsi10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?
```
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room4</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
</queryContextRequest>

<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room8</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>30</contextValue>
          </contextAttribute>
          <contextAttribute>
            <name>temperature</name>
            <type>Farenheit</type>
            <contextValue>30</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>

EOF
However, this is probably not the desired result (as 30ºC corresponds to 80ºF or 30ºF corresponds to -1.11ºC, both cannot have equal value at the same time!). Thus, in general, we recommend to always use types, when possible.

2.3.4.12 Extending duration

We have seen that registrations and subscriptions (both context and context availability ones) have a duration. The expiration date is calculated using the following formula:

- \( \text{expiration} = \text{current-time} + \text{duration} \)

The behavior of the broker regarding expired elements is the following:

- For registrations: an expired registration is not taken into account in discoverContextAvailability request processing, but it is still updatable.

- For subscriptions: an expired subscription is not taken into account to send new notifications based on it, but it is still updatable (using updateContextSubscription/updateContextSubscriptionAvailability) and it can be canceled (using unsubscribeContext/unsubscribeContextAvailability).

Note that Orion Context Broker doesn't remove expired elements from the database, but that they can be easily deleted as described in the administration manual.

Finally, take into account that the expiration is \textit{recalculated} on updates, not \textit{expanded}. Let's clarify this with an example. Let's suppose that at 18:30 you do a subscription with PT1H duration (i.e. one hour). Thus, it will expire at 19:30. Next, at 19:00 you do an update using again PT1H as duration. So, that hour period is not added to 19:30 (the previous expiration limit) but added to the 19:00 (the current time). Thus, the new expiration time is 20:00.
2.3.4.13  *Only-type entity registrations using convenience operations*

**Since release 0.9.0 (FI-WARE 3.2.3):** new in this release.

You can use the NGSI9 "contextEntityTypes" convenience operations to register entity types without an specific ID (whatever it means in the context of your context-based application :). Let's illustrate with an example.

Let's register the "Funny" entity type (note that we are not specifying any entity ID):

```bash
(curl localhost:1026/ngsi9/contextEntityTypes/Funny -s -S --header
 'Content-Type: application/xml' -d @- | xmllint --format - ) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>
  <providingApplication>http://mysensors.com/Funny</providingApplication>
</registerProviderRequest>
EOF
```

Now, let's discover on that type:

```bash
curl localhost:1026/ngsi9/contextEntityTypes/Funny -s -S --header
 'Content-Type: application/xml' | xmllint --format -

<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Funny" isPattern="false">
            <id/>
          </entityId>
        </entityIdList>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
```
As you can see, the ID element is empty (it makes sense, as we didn't specify any ID at registration).

Moreover, you can register attributes in these registrations, e.g:

```
(curl localhost:1026/ngsi9/contextEntityTypes/MoreFunny/attributes/ATT
 -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) << EOF
<?xml version="1.0"?>
<registerProviderRequest>
  <duration>P1M</duration>

<providingApplication>http://mysensors.com/Funny</providingApplication>
</registerProviderRequest>
EOF

curl localhost:1026/ngsi9/contextEntityTypes/MoreFunny -s -S --header 'Content-Type: application/xml' | xmllint --format -

<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="MoreFunny" isPattern="false">
```
2.3.4.14 **Structured attribute values**

New since version 0.11.0.

Apart from simple strings such as "22.5" or "yellow", you can use complex structures as attribute values. In particular, an attribute can be set to a vector or to a key-value map (usually referred to as an "object") using `updateContext` (or equivalent convenience operation). These values are retrieved with a `queryContext` on that attribute (or equivalent convenience operation) and notifyContext notifications sent as a consequence of a NGSI10 subscriptions.

Vector or key-map values correspond directly to JSON vectors and JSON objects, respectively. Thus, the following `updateContext` request sets the value of attribute "A" to a vector and the value of attribute B to a key-map object (we use UPDATE as actionType, but this can be used at initial entity or attribute creation with APPEND).

```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @python -mjson.tool) <<EOF
{
```
"contextElements": [
    {
        "type": "T1",
        "isPattern": "false",
        "id": "E1",
        "attributes": [
            {
                "name": "A",
                "type": "T",
                "value": [ "22",
                           {
                               "x": [ "x1", "x2" ],
                               "y": "3"
                            }
                       ],
            {
                "name": "B",
                "type": "T",
                "value": {
                    "x": { "x1": "a", "x2": "b" },
                    "y": [ "y1", "y2" ]
                }
            }
        ]
    },
    {
        "name": "C",
        "type": "T",
        "value": {
            "x": [ "x1", "x2" ],
            "y": [ "y1", "y2" ]
        }
    }
]
In the case of XML, the structure is a bit less "natural" than in JSON, but equivalent:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T1" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>A</name>
          <type>T</type>
          <contextValue type="vector">
            <item>22</item>
            <x type="vector">
              <item>x1</item>
              <item>x2</item>
            </x>
            <y>3</y>
          </contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
```

```bash
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T1" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>A</name>
          <type>T</type>
          <contextValue type="vector">
            <item>22</item>
            <x type="vector">
              <item>x1</item>
              <item>x2</item>
            </x>
            <y>3</y>
          </contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
</updateContextRequest>
EOF
```
The particular rules applied to XML format are the following ones:

- **Tags not using** the "type" attribute equal to "vector" represent key-map object, taking into account the following:
  - Each child tag is considered a key-map pair of the object, whose key is the tag name and the value is the content of the tag. The value can be either a string or a tag (which can represent either a vector or a key-map object).
It is not allowed to have 2 child tags with the same name (a parse error is generated in that case).

- Tags **using** the "type" attribute equal to "vector" represent a vector, taking into account the following:
  - Each child tag is considered a vector element, whose value is the content of the tag. The value can be either a string or a tag (which can represent either a vector or a key-map object). The name of the tag is not taken into account (vector elements have no name, otherwise the vector wouldn't be an actual vector, but a key-map).
  - In updateContext, you can use whatever name you want for child tags. However, all the child tags must have the same name (otherwise a parse error is generated).

- Except for "type", XML attributes are not allowed within the sub-tree of an structured value (a parse error is generated as a consequence).

The value of the attribute is stored internally by Orion Context Broker in a format-independent representation. Thus, you can updateContext a structured attribute using JSON, then queryContext that attribute using XML (or vice-versa) and you get the equivalent result. The internal format-independent representation ignores the tag names for vector items set using XML (as described above), so the queryContextResponse/notifyContextRequest in XML uses always a predefined tag name for that: <item> (that may not match the tag name used in the updateContext operation setting that value; in fact, the updateContext operation could have been done with JSON, in which case the tag name for vector items has no sense at all).

Note that in order to align XML/JSON representations, the final "leaf" elements of the structured attribute values after traversing all vectors and key-maps are always considered as strings. String is the "natural" type for elements in XML, but not in JSON (that allow other types, such as integer). Thus, take into account that although you can use for instance a JSON integer as a field value in updates (such as {"x": 3}), you will retrieve a string in queries and notifications (i.e. {"x": "3"}).

### 2.3.4.15 Geolocation capabilities

**New since version 0.11.0.**

Orion Context Broker has several capabilities related to geolocation that are described in this section. It is strictly required to use MongoDB 2.4 or higher in order to use geolocation features (see **requirements** section in the installation manual).

#### 2.3.4.15.1 Defining location attribute

Entities can have a location, specified by one of its attributes. In order to state which attribute (among all the ones belonging to the entity) defines the location, the "location" metadata is used. For example, the following updateContext request creates the entity "Madrid" (of type "City") with attribute "position" defined as location.

```xml
<item>
  <position>
    <value>Madrid</value>
  </position>
</item>
```
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<updateContextRequest>
 <contextElementList>
  <contextElement>
   <entityId type="City" isPattern="false">
    <id>Madrid</id>
   </entityId>
   <contextAttributeList>
    <contextAttribute>
     <name>position</name>
     <type>coords</type>
     <contextValue>40.418889, -3.691944</contextValue>
     <metadata>
      <contextMetadata>
       <name>location</name>
       <type>string</type>
       <value>WSG84</value>
      </contextMetadata>
     </metadata>
    </contextAttribute>
   </contextAttributeList>
  </contextElement>
 </contextElementList>
<updateAction>APPEND</updateAction>
</updateContextRequest>
EOF

JSON
(curl localhost:1026/NGSI10/updateContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
  {
    "type": "City",
    "isPattern": "false",
    "id": "Madrid",
    "attributes": [
      {
        "name": "position",
        "type": "coords",
        "value": "40.418889, -3.691944",
        "metadatas": [
          {
            "name": "location",
            "type": "string",
            "value": "WSG84"
          }
        ]
      }
    ]
  },
  "updateAction": "APPEND"
}
EOF

Additional comments:
Note that you can use different attributes to specify the location in different entities, e.g. entity "Car1" could be using "position" attribute, while entity "Phone22" could use attribute "coordinates".

In order to avoid inconsistencies, only one attribute at a time can be defined as location. If you want to redefine the attribute of an entity used for location, first you have to DELETE it, then APPEND the new one (check the section about adding and removing attributes dynamically).

The value of the location metadata is the coordinates system used. Current version only support **WSG84** (which is the one used internally by the MongoDB database) but other systems may be added in future versions.

The value of the location attribute is a string with two numbers separated by a comma (",": the first number is the latitude and the second is the longitude. Only decimal notation is allowed (e.g. "40.418889"), degree-minute-second notation is not allowed (e.g. "40°44'55"N").

### 2.3.4.15.2 Geo-located queries

Entities location can be used in `queryContext` (or equivalent convenience operations). To do so, we use the scope element, using "FIWARE_Location" as scopeType and an area specification as scopeValue. The query result includes only the entities located in that area, i.e. context elements belonging to entities not included in the area are not taken into account. Regarding area specification, Orion Context Broker allows the following possibilities:

- Area internal to a circumference, given its centre and radius.
- Area external to a circumference, given its centre and radius.
- Area internal to a polygon (e.g. a terrain zone, a city district, etc.), given its vertices.
- Area external to a polygon (e.g. a terrain zone, a city district, etc.), given its vertices.
- Area unions or intersections (e.g. the intersection of a circle and a polygon) are not supported in the current version.

In order to illustrate geo-located queries with polygons, let's consider the following scenario: three entities (A, B and C, of type "Point") have been created in Orion Context Broker, each one in the coordinates shown in the following picture.
Let's consider a query whose scope is the internal area to the square defined by coordinates (0, 0), (0, 6), (6, 6) and (6, 0).

To define a polygon, we use the polygon element which, in sequence, include a vertexList. A vertexList is composed by a list of vertex elements, each one containing a couple of elements (latitude and longitude) that provide the coordinates of the vertex. The result of the query would be A and B.

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Point" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
</queryContextRequest>

D.6.4.3: Fi-WARE Users and Programmers Guide 213
<entityId>
</entityIdList>
<attributeList>
</attributeList>
<restriction>
<scope>
<operationScope>
<scopeType>FIWARE_Location</scopeType>
<scopeValue>
<polygon>
<vertexList>
<vertex>
<latitude>0</latitude>
<longitude>0</longitude>
</vertex>
<vertex>
<latitude>0</latitude>
<longitude>6</longitude>
</vertex>
<vertex>
<latitude>6</latitude>
<longitude>6</longitude>
</vertex>
<vertex>
<latitude>6</latitude>
<longitude>0</longitude>
</vertex>
<vertex>
<latitude>0</latitude>
<longitude>0</longitude>
</vertex>
</vertexList>
</polygon>
</scopeValue>
</scope>
</restriction>
</attributeList>
</entityList>
```json
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d '@-' | python -mjson.tool) <<EOF
{
   "entities": [
   {
      "type": "Point",
      "isPattern": "true",
      "id": ".*"
   }
   ],
   "restriction": {
      "scopes": [
         {
            "type": "FIWARE_Location",
            "value": {
               "polygon": {
                  "vertices": [
                     {
                        "latitude": "0",
                        "longitude": "0"
                     },
                     {
                        "latitude": "0",
                        "longitude": "0"
                     }
                  ]
               }
            }
         }
      ]
   }
EOF
```
Let's consider a query whose scope is the internal area to the rectangle defined by coordinates (3, 3), (3, 8), (11, 8) and (11, 3).
The result of the query would be B and C.

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Point" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
  <restriction>
    <scope>
      <operationScope>
        <scopeType>FIWARE_Location</scopeType>
        <scopeValue>
          <polygon>
            <vertexList>
              <vertex>
              </vertex>
            </vertexList>
          </polygon>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
<vertex>
  <latitude>3</latitude>
  <longitude>3</longitude>
</vertex>
<vertex>
  <latitude>3</latitude>
  <longitude>8</longitude>
</vertex>
<vertex>
  <latitude>11</latitude>
  <longitude>8</longitude>
</vertex>
<vertex>
  <latitude>11</latitude>
  <longitude>3</longitude>
</vertex>
</vertexList>
</scope>
</restriction>
</queryContextRequest>
EOF

JSON

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{

"entities": [
{
 "type": "Point",
 "isPattern": "true",
 "id": ".*"
}
],
"restriction": {
 "scopes": [
 {
 "type": "FIWARE_Location",
 "value": {
 "polygon": {
 "vertices": [
 {
 "latitude": "3",
 "longitude": "3"
 },
 {
 "latitude": "3",
 "longitude": "8"
 },
 {
 "latitude": "11",
 "longitude": "8"
 },
 {
 "latitude": "11",
 "longitude": "3"
 }
 ]
 } 
]
However, if we consider the query to the external area to that rectangle, the result of the query would be A. To specify that we refer to the area external to the polygon we include the inverted element set to "true".

```xml
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Point" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
  <restriction>
    <scope>
      <operationScope>
        <scopeType>FIWARE_Location</scopeType>
      </operationScope>
      <scopeValue>
```
<polygon>
  <vertexList>
    <vertex>
      <latitude>3</latitude>
      <longitude>3</longitude>
    </vertex>
    <vertex>
      <latitude>3</latitude>
      <longitude>8</longitude>
    </vertex>
    <vertex>
      <latitude>11</latitude>
      <longitude>8</longitude>
    </vertex>
    <vertex>
      <latitude>11</latitude>
      <longitude>3</longitude>
    </vertex>
  </vertexList>
  <inverted>true</inverted>
</polygon>

EOF
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "Point",
      "isPattern": "true",
      "id": ".*"
    }
  ],
  "restriction": {
    "scopes": [
      {
        "type": "FIWARE_Location",
        "value": {
          "polygon": {
            "vertices": [
              {
                "latitude": "3",
                "longitude": "3"
              },
              {
                "latitude": "3",
                "longitude": "8"
              },
              {
                "latitude": "11",
                "longitude": "8"
              }
            ]
          }
        }
      }
    ]
  }
}EOF
Let's consider a query whose scope is the internal area to the triangle defined by coordinates (0, 0), (0, 6) and (6, 0).

The result of the query would be A.

```
Let's consider a query whose scope is the internal area to the triangle defined by coordinates (0, 0), (0, 6) and (6, 0).

The result of the query would be A.
```

```
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
```
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Point" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
  <restriction>
    <scope>
      <operationScope>
        <scopeType>FIWARE_Location</scopeType>
        <scopeValue>
          <polygon>
            <vertexList>
              <vertex>
                <latitude>0</latitude>
                <longitude>0</longitude>
              </vertex>
              <vertex>
                <latitude>0</latitude>
                <longitude>6</longitude>
              </vertex>
              <vertex>
                <latitude>6</latitude>
                <longitude>0</longitude>
              </vertex>
            </vertexList>
          </polygon>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "entities": [ 
    {
      "type": "Point",
      "isPattern": "true",
      "id": ".*"
    }
  ],
  "restriction": {
    "scopes": [ 
      {
        "type": "FIWARE_Location",
        "value": {
          "polygon": {
            "vertices": [ 
              {
                "latitude": "0",
                "longitude": "0"
              }
            ]
          }
        }
      }
    ]
  }
}
EOF
However, if we consider the query to the external area to that triangle (using the inverted element set to "true"), the result of the query would be B and C.

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF

<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Point" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
</queryContextRequest>

EOF
<entityIdList>
<attributeList>
</attributeList>
<restriction>
<scope>
<operationScope>
<scopeType>FIWARE_Location</scopeType>
<scopeValue>
<polygon>
<vertexList>
<vertex>
<latitude>0</latitude>
<longitude>0</longitude>
</vertex>
<vertex>
<latitude>0</latitude>
<longitude>6</longitude>
</vertex>
<vertex>
<latitude>6</latitude>
<longitude>0</longitude>
</vertex>
</vertexList>
</polygon>
</scopeValue>
</operationScope>
</scope>
</restriction>
</queryContextRequest>
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
  ]
},
"restriction": {
  "scopes": [
  ]
},
"vertices": [
},
{ "vertices": [
  ],
{ "vertices": [
  ],
{ "vertices": [
  ],
{ "vertices": [
  ],
{ "vertices": [
  ]
}
EOF
Now, in order to illustrate circle areas, let's consider the following scenario: three entities (representing the cities of Madrid, Alcobendas and Leganes) have been created in Orion Context Broker. The coordinates for Madrid are (40.418889, -3.691944); the coordinates for Alcobendas are (40.533333, -3.633333) and the coordinates for Leganes are (40.316667, -3.75). The distance between Madrid and Alcobendas is around 14.5 km, and the distance between Madrid and Leganes is 13.5 km.

Let's consider a query whose scope is inside a radius of 14 km (14000 meters) centred in Madrid.
To define a circle, we use the circle element which, in sequence, include a three elements: centerLatitude (the latitude of the circle center), centerLongitude (the longitude of the circle center) and radius (in meters). The result of the query would be Madrid and Leganes.

**XML**

```xml
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="City" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
</queryContextRequest>
EOF
```
<restriction>
  <scope>
    <operationScope>
      <scopeType>FIWARE_Location</scopeType>
      <scopeValue>
        <circle>
          <centerLatitude>40.418889</centerLatitude>
          <centerLongitude>-3.691944</centerLongitude>
          <radius>14000</radius>
        </circle>
      </scopeValue>
    </operationScope>
  </scope>
</restriction>
</queryContextRequest>
EOF

JSON

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
    {
      "type": "City",
      "isPattern": "true",
      "id": ".*"
    }
  ],
EOF
Let's consider a query whose scope is inside a radius of 15 km (15000 meters) centred in Madrid.
The result of the query would be Madrid, Leganes and Alcobendas.

```xml
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="City" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
  <restriction>
    <scope>
    </scope>
  </restriction>
</queryContextRequest>
```

D.6.4.3: FI-WARE Users and Programmers Guide 233
<operationScope>
  <scopeType>FIWARE_Location</scopeType>
  <scopeValue>
    <circle>
      <centerLatitude>40.418889</centerLatitude>
      <centerLongitude>-3.691944</centerLongitude>
      <radius>15000</radius>
    </circle>
    </scopeValue>
  </operationScope>
</scope>
</restriction>
</queryContextRequest>
EOF

```
JSON
(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m jjson.tool) <<EOF
{
  "entities": [
  {
    "type": "City",
    "isPattern": "true",
    "id": ".*"
  }]
  "restriction": {
    "scopes": [
```
Let's consider a query whose scope is outside a radius of 14 km (14000 meters) centred in Madrid.
We use the inverted element set to "true". The result of the query would be Alcobendas.

XML

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="City" isPattern="true">
      <id>.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
  </attributeList>
  <restriction>
    <scope>
<operationScope>
  <scopeType>FIWARE_Location</scopeType>
  <scopeValue>
    <circle>
      <centerLatitude>40.418889</centerLatitude>
      <centerLongitude>-3.691944</centerLongitude>
      <radius>14000</radius>
      <inverted>true</inverted>
    </circle>
  </scopeValue>
</operationScope>
</scope>
</restriction>
</queryContextRequest>
EOF

JSON

(curl localhost:1026/NGSI10/queryContext -s -S --header 'Content-Type: application/json' --header 'Accept: application/json' -d @- | python -m json.tool) <<EOF
{
  "entities": [
    {
      "type": "City",
      "isPattern": "true",
      "id": ".*"
    }
  ],
  "restriction": {
...

D.6.4.3: FIWARE Users and Programmers Guide 237
2.3.4.16 **Configuration Manager role**

The Orion Context Broker can run in "Configuration Management Role", which is a work-mode specially thought for scenarios in which the broker plays the role of Configuration Manager GE in the FIWARE platform (as you would probably have guessed :).

This is done using the "-ngsi9", "-fwdHost" and "-fwdPort" command line options at start time (see the administration manual to know how to configure the broker at start time) as described below:

- Using "-ngsi9" disables NGSI10, i.e. any attempt to use NGSI10 operations will result in an error.
- Register Context requests are forwarded to the host and port specified by "-fwdHost" and "-fwdPort". In other words, each time the Orion Context Broker receives a registerContext request (for creating new registrations or updating existing ones) a registerContext (with the same registration information) is sent to fwdHost/fwdPort (where usually another instance of Orion Context Broker runs, configured in normal mode). Note that the registrationID in Orion "ConfMan" and in Orion "normal" will be different, but it is not a problem as Orion "ConfMan" keeps an ID mapping. Using "-fwdPort 0" (or not using the option at all) disables registerContext forwarding.
Note that both "-ngsi9" and "-fwdHost/-fwdPort" are independent, e.g. you can enable registerContext forwarding using "-fwdHost/-fwdPort" without disabling NGSI10 with "-ngsi9". However, to get the proper behaviour of a Configuration Manager GE you need to use all three options.

### 2.3.4.17 Associations

Associations allow to configure relationship between context elements. For example, let's suppose that you have completed the steps of the tutorial about registerContext operation and that you want to create an association between adjacent room temperatures, so you can specify that the temperature of a given room is the temperature of your neighbour in right direction. Let's name this association "right_neighbour". The knowledge of this relationship is useful for building an application to detect "heat waves" that propagates along the rooms of a building.
Associations are created using registerContext operations, but instead of a "normal" contextRegistration element (with entityIdList and contextRegistrationAttributeList) we use a registrationMetadata element:

```
(curl localhost:1026/ngsi9/registerContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0"?>
  <registerContextRequest>
    <contextRegistrationList>
      <contextRegistration>
        <registrationMetadata>
          <contextMetadata>
            <name>right_neighbour</name>
            <type>Association</type>
            <value>
              <entityAssociation>
```

D.6.4.3: FI-WARE Users and Programmers Guide 240
From version 0.13.0, the XML tags for associations (attributeAssociationList and attributeAssociation) start with lowercase 'a'.

The elements within registrationMetadata are the following:

- A name for the association ("right_neighbour" in our case)
- A type, whose value must be "Association"
- The entityAssociation element, that specifies the source and target of the association at entity level. In this case, the target is Room2 and the source is Room1. In our example, both target and
source have the same type ("Room") but you can also associate entities of different types, e.g. a Car associated to a garage Room.

- The attributeAssociationList which defines the attribute associations. In this case, we are using a "temperature" (of source Room) to "temperature" (of target Room) association. Although in the example the associated attributes have the same name, this is not mandatory.

Note that although it doesn't make sense to specify a providing application for associations (as they are links between context elements, but no context elements themselves) we have to include one in the request to conform with NGSI specification. We use http://www.fi-ware.eu/NGSI/association by convention.

Associations are like normal registrations, e.g. they have an expiration and a registration ID that is returned in the response:

```
<registerContextResponse>
  <registrationId>51c45dafe955db1d6e22c729</registrationId>
  <duration>P1M</duration>
</registerContextResponse>
```

In order to navigate associations we use the usual discoverContextAvailability operation, but adding a scope restriction to specify that we are looking for associated targets and source. For example, if we want to discover the sources of Room2 temperature we will use the following request:

```
(curl localhost:1026/ngsi9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @ - | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room2</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <restriction>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </restriction>
</discoverContextAvailabilityRequest>
```
The response is as follows:

```xml
<?xml version="1.0"?>
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room1</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Raws</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
```
</contextRegistration>
</contextRegistrationResponse>
<contextRegistrationResponse>
<contextRegistration>
 <registrationMetadata>
  <contextMetadata>
   <name>right_neighbour</name>
   <type>Association</type>
   <value>
    <entityAssociation>
     <sourceEntityId type="Room" isPattern="false">
      <id>Room1</id>
     </sourceEntityId>
     <targetEntityId type="Room" isPattern="false">
      <id>Room2</id>
     </targetEntityId>
    </entityAssociation>
    <attributeAssociationList>
     <attributeAssociation>
      <sourceAttribute>temperature</sourceAttribute>
      <targetAttribute>temperature</targetAttribute>
     </attributeAssociation>
    </attributeAssociationList>
   </value>
  </contextMetadata>
 </registrationMetadata>
 <providingApplication>http://www.fi-ware.eu/NGSI/association</providingApplication>
</contextRegistration>
</contextRegistrationResponse>
The response includes the source of the association and the association itself (in its corresponding registrationMetadata element).

If we want to discover in the opposite way, i.e. the targets of Room1 temperature we use the following request:

```
(curl localhost:1026/ngsi9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <restriction>
    <scope>
      <operationScope>
        <scopeType>IncludeAssociations</scopeType>
        <scopeValue>TARGETS</scopeValue>
      </operationScope>
    </scope>
  </restriction>
</discoverContextAvailabilityRequest>
EOF
```

and in this case we will get the target of the association and the association itself (in its corresponding registrationMetadata element).
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room2</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Rooms</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
Just to compare, have a look at what happens with the `discoverContextAvailability` request but in a "conventional" way, i.e. without the scope element:

```
(curl localhost:1026/ngsi9/discoverContextAvailability -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format - ) <<EOF
<?xml version="1.0"?>
<discoverContextAvailabilityRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
    <entityId type="Room" isPattern="false">
      <id>Room2</id>
    </entityId>
  </entityIdList>
</discoverContextAvailabilityRequest>
```
In this case, the response provides the Room1 information and no association metadata is included (this is the usual behaviour, as described previously in this document):

```xml
<discoverContextAvailabilityResponse>
  <contextRegistrationResponseList>
    <contextRegistrationResponse>
      <contextRegistration>
        <entityIdList>
          <entityId type="Room" isPattern="false">
            <id>Room1</id>
          </entityId>
        </entityIdList>
        <contextRegistrationAttributeList>
          <contextRegistrationAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <isDomain>false</isDomain>
          </contextRegistrationAttribute>
        </contextRegistrationAttributeList>
        <providingApplication>http://mysensors.com/Rooms</providingApplication>
      </contextRegistration>
    </contextRegistrationResponse>
  </contextRegistrationResponseList>
</discoverContextAvailabilityResponse>
```
An important use case for associations in the FI-WARE platform is the one in which the IoT Broker GE works in combination with Orion Context Broker acting in Configuration Manager role. In this case, the IoT Broker uses associations to create things-device associations in the Orion Context Broker (e.g. things are Rooms and devices are sensors in that rooms). Have a look at the association extension definition for more detail on this.

**Release Note (0.5.0 FIWARE 2.3.3):** associations are a beta feature introduced in this version. Thus, it is limited and doesn't provide all the functionality described in the association extension definition. In particular:

- Entity-Entity "pure" associations (i.e. without specifying particular attribute-attribute associations) has not been tested.
- More than one element in the attributeAssociationList in registerContext has not been tested.
- Recursive associations have not been tested.
- We have tested association on entities/attributes registered using standard operations. We haven't tested using convenience operations.

### 2.3.4.18 Context Broker Federation

**Release Note (0.7.0 FIWARE 3.1.2):** new since this release.

Apart from processing updateContext and registerContext (usually issued by a client application) Orion Context Broker can process notifyContextRequest and notifyContextAvailabilityRequest with the same semantics. This opens the door to interesting federation scenarios (one example is the FI-LAB context management platform).

Consider the following setup: three context broker instances running in the same machine (of course, this is not a requirement but makes things simpler to test this feature), in ports 1030, 1031 and 1032 respectively and using different databases (named A, B and C to be brief). Let's start each instance (run each command in a separate terminal):

```
contextBroker -fg -port 1030 -db orion1030
```
contextBroker -fg -port 1031 -db orion1031
contextBroker -fg -port 1032 -db orion1032

Next, let's send a subscribeContext to A (to make B subscribe to updates made in A). Note that the URL used in the reference has to be "/ngsi10/notifyContext":

(curl localhost:1030/NGSI10/subscribeContext -s -S --header 'Content-Type: application/xml' -d @ | xmllint --format |) <<EOF
<?xml version="1.0"?>
<subscribeContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <duration>P1M</duration>
  <notifyConditions>
    <notifyCondition>
      <type>ONCHANGE</type>
      <condValueList>
        <condValue>temperature</condValue>
      </condValueList>
    </notifyCondition>
  </notifyConditions>
  <throttling>PT5S</throttling>
</subscribeContextRequest>
EOF

Next, let's send a subscribeContext to B (to make C subscribe to updates made in B). The subscription is basically the same, only the port in the curl line and reference elements are different.
Future Internet Core Platform

```
(curl localhost:1031/NGSI10/subscribeContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0"?>
<subscribeContextRequest>
    <entityIdList>
        <entityId type="Room" isPattern="false">
            <id>Room1</id>
        </entityId>
    </entityIdList>
    <duration>P1M</duration>
    <notifyConditions>
        <notifyCondition>
            <type>ONCHANGE</type>
            <condValueList>
                <condValue>temperature</condValue>
            </condValueList>
        </notifyCondition>
    </notifyConditions>
    <throttling>PT5S</throttling>
</subscribeContextRequest>
EOF
```

Now, let's create an entity in context broker A.

```
(curl localhost:1030/NGSI10/updateContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
    <contextElementList>
        <contextElement>
            <entityId type="Room" isPattern="false">
```
Given the subscriptions in place, a notifyContextRequest is automatically sent from A to B. That event at B causes in sequence a notifyContextRequest to be sent to C. So, at the end, the creation of an entity in A causes the creation of the same entity (with the same attribute values) in C. You can check it by doing a queryContext to C:

```
(curl localhost:1032/NGSI10/queryContext -s -S --header 'Content-Type: application/xml' -d @<entityIdList>
<entityId type="Room" isPattern="false">
  <id>Room1</id>
</entityId>
</entityIdList>
<attributeList/>
</queryContextRequest>
EOF
```
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Room" isPattern="false">
          <id>Room1</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>23</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
    <statusCode>
      <code>200</code>
      <reasonPhrase>OK</reasonPhrase>
    </statusCode>
  </contextResponseList>
</queryContextResponse>

In the current context broker version, the semantics of notifyContextRequest are the same that updateContext APPEND or, if the context element already exist, the semantics of updateContext UPDATE. Thus, federation doesn't provide exact mirroring: a updateContext DELETE to one context broker will not produce the same effect in the federated context broker.

This mechanism works similarly with registerContext and subscribeContextAvailability. In this case, the URL for the reference element is "/ngsi9/notifyContextAvailability".
2.3.4.19 Mixing JSON and XML requests

Release Note (0.9.0 FIWARE 3.2.2): new since this release (as was the one in which JSON was introduced).

You can use XML and JSON at the same time. For example, you can do a registerContext in XML for a given entity, the discoverContextAvailability in JSON to retrieve it.

2.3.4.20 Cross-format notifications

New since 0.13.0.

Default behavior for NGSI10 notifications (/ngsi10/notifyContext) is to use the same format (XML or JSON) that was used at creation time of the subscription (/ngsi10/subscribeContext). If another format is desired, the URL attribute "notifyFormat" must be used at subscription time:

```
(curl 'localhost:1026/NGSI10/subscribeContext?notifyFormat=json' -s -S
 --header 'Content-Type: application/xml' -d @ | xmllint --format -)
```

The above example shows a subscribeContext done using XML, that will receive notifications in JSON. Currently, cross-format notifications are not supported in NGSI9 subscribeContextAvailability.

2.3.4.21 HTTP and NGSI response codes

Two independent response codes are being considered in the API responses: one "internal" at NGSI level (i.e. encoded in the REST HTTP response payload) and other "external" at HTTP level (the HTTP response code itself). Note that this manual focuses on the NGSI aspects of the API, thus we always assume in this documentation (unless otherwise noted) that HTTP code is "200 OK".

In order to illustrate the existence of both codes and their independence, let's consider a queryContext operation on a non-existing entity (e.g. "foo"). Note the -v flag in the curl command, in order to print the HTTP response codes and headers:

```
# curl localhost:1026/ngsi10/contextEntities/foo -s -S --header
'Content-Type: application/xml' -v | xmllint --format -
```

* About to connect() to localhost port 1026 (#0)
* Trying ::1... connected
* Connected to localhost (::1) port 1026 (#0)
> GET /ngsi10/contextEntities/foo HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7
    NSS/3.13.1.0 zlib/1.2.3 libidn/1.18 libssh2/1.2.2
> Host: localhost:1026
> Accept: */*
> Content-Type: application/xml
>
< HTTP/1.1 200 OK
< Content-Length: 316
< Content-Type: application/xml
< Date: Mon, 31 Mar 2014 10:13:45 GMT
<
{ [data not shown]
* Connection #0 to host localhost left intact
* Closing connection #0
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    <entityId type="" isPattern="false">
      <id>foo</id>
    </entityId>
  </contextElement>
</contextElementResponse>
<span style="color: #666; font-size: 80%;">StatusCode
  <code>404</code>
  <reasonPhrase>No context element found</reasonPhrase>
  <details>Entity id: 'foo'</details>
</statusCode>
</contextElementResponse>
Note that in this case the NGSI response code is "404 No context element found" while the HTTP is "200 OK". Thus, in other words, the communication at HTTP level was ok, although an error condition (the entity doesn't exist in Orion Context Broker database) happened at the NGSI level.

The following example shows a case of an HTTP level problem, due to a client attempting to get the response in a MIME type not supported by Orion (in this case "text/plain"). In this case, an HTTP response code "406 Not Acceptable" is generated.

```bash
# curl localhost:1026/ngsi10/contextEntities/foo -s -S --header
'Accept: text/plain' -v | xmllint --format -
* About to connect() to localhost port 1026 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 1026 (#0)
> GET /ngsi10/contextEntities/foo HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7
> Host: localhost:1026
> Accept: text/plain
>
< HTTP/1.1 406 Not Acceptable
< Content-Length: 196
< Content-Type: application/xml
< Date: Mon, 31 Mar 2014 10:16:16 GMT
<
{ [data not shown]
* Connection #0 to host localhost left intact
* Closing connection #0
<?xml version="1.0"?>
<orionError>
  <code>406</code>
  <reasonPhrase>Not Acceptable</reasonPhrase>
  <details>acceptable types: 'application/xml' but Accept header in request was: 'text/plain'</details>
```
2.3.4.22 **Security considerations**

Orion Context Broker doesn't implement any access control mechanism, i.e. any application can issue any request on it as long as it has connectivity to the host and port where the broker is running. However, access control are provided by other mechanisms, as the access control framework provided by FI-WARE GEs.

**Since 0.12.0:** Orion Context Broker supports HTTPS, using the `-https` options (which in addition needs the `-key` and `-cert` options, to specify the files containing the private key and certificates for the server, respectively). Check the command line options section in the administration manual for details. Note that current Orion version cannot run in both HTTP and HTTPS at the same time, i.e. using `-https` disables HTTP.

**Since 0.13.0:** Apart from using HTTPS in the API server exported by Orion, you can also use HTTPS in notifications. In order to do so:

- You have to use the "https" protocol schema in URL in your reference element in `subscribeContext` or `subscribeContextAvailability` subscriptions, e.g.

  ```xml
  ...<reference>https://mymachime.example.com:1028/notify</reference>...
  ```

- You have to use Rush as relayer (as the HTTPS encoding is implemented in Rush). See how to run Orion using Rush for additional information on this topic.

2.3.4.23 **Known limitations**

2.3.4.23.1 **Attribute values**

Although NGSI specifies that the value for attributes is `xsd:any` (which basically means "anything"), the current Orion Context Broker version only allows plain strings. E.g:

```xml
...<contextValue>This is a nice string</contextValue>...
```

That use to be the case in almost any situation, so you probably don't have to worry about this. However, take into account that you should avoid using the following attribute values in updateContext:

- Strings that could be interpreted as XML (either well-formed or wrong-formed), e.g:
Using the above patterns for attribute values may lead to unexpected behaviour in the broker.

2.3.4.23.2  *Request maximum size*

The current maximum request size in Orion Context Broker is 1 MB. This limit should suffice the most of the use cases and, at the same time, avoids denial of service due to too large requests. If you don't take this limitation into account, you will get messages such the following ones:

```xml
<?xml version="1.0"?>
<queryContextResponse>
    <errorCode>
        <code>413</code>
        <reasonPhrase>Payload Too Large</reasonPhrase>
        <details>payload size: 1500748</details>
    </errorCode>
</queryContextResponse>
```

Or, if you are sending a huge request, this one:

```html
<html>
    <head><title>Internal server error</title></head>
```
Some programmer needs to study the manual more carefully.

If you find this 1MB limit too coarse, send us an email so we can consider your feedback in future releases.

2.3.4.23.3 Notification maximum size
Notification maximum size is set to 8MB in Orion Context Broker version 0.9.1. Larger notifications will not be sent by context broker and you will get the following trace in the log file:

HTTP request to send is too large: N bytes

where N is the number of bytes of the too large notification.

2.3.4.23.4 Cross-format notifications
This limitation was overcome for NGSI10 notifications in version 0.13.0. It is still being a limitation for NGSI9 notifications.
Orion Context Broker 0.9.0 (FI-WARE 3.2.3) introduced JSON, an alternative to XML for requests and responses. However, note that if you do a subscribeContext/subscribeContextAvailability in a given format, the notifyContextRequest/notifyContextAvailabilityRequest messages sent as a consequence of that subscription are sent in the same format. For example, if you do subscribeContext using XML, you will get notifyContextRequest in XML. Currently, it is no possible to receive notifications in the opposite format (in this case, JSON).

2.3.4.23.5 Content-Length header is required
Orion Context Broker expects always a Content-Length header in all client requests, otherwise the client will receive a "411 Length Required" response. This is due to the way the underlying HTTP library (microhttpd) works, see details in this email thread in the microhttpd mailing list.

2.3.5 FI-LAB context management platform
FI-LAB implements a context information management platform based on Orion Context Broker to which FI-LAB users can access. This platform allows access to real time sensor information and open data datasets, integrating Orion with other FI-WARE GEs, such as the Backend Device Management and Cosmos. This section describes the details of this platform, shown in the figure.
The different elements are described following, from bottom to top:

- Different sensor sources are generating data to feed the platform. In the moment of writing this, the sensor information from Las Llamas park in Santander (Spain) used in the FI-WARE LiveDemo along with information coming from other smart cities (Santander, Seville, etc.) is being collected, but we are enriching our sensor base in a regularly basis.

- In the Data Source Layer, there are several instances of the Backend Device Management GEi that aggregate the information coming from sensors. That information is progressed to a Orion Context Broker instance. Note that, at this level, there is one context broker per Backend Device Management GEi. The FI-LAB user don't access directly to Data Source Layer.

- The context broker instance at Data Source Layer is federated with the context broker instance at Aggregation Layer. Thus, this context broker, located at orion.lab.fi-ware.eu:1026, receives all the context information from all data sources. This instance is accessible to FI-LAB user (through OAuth2 FI-WARE framework), but note that is a shared among all FI-LAB users, e.g. if you register your own entities there, others users can access to them and even modify them.
  - An example of Orion Context Broker client authenticating using OAuth can be found in https://github.com/fgalan/oauth2-example-orion-client. You can use either the full client in Node.js or the simple token_script.sh to get a valid X-Auth-Token (however, this token would expire from time to time, so you are encourage to develop a full fledged OAuth authentication system in your client).
- FI-LAB users can create their own instances of context broker in the FI-LAB cloud. That way, you can deploy and use a non-shared instance of the broker. You can federate your broker with the shared one in orion.lab.fi-ware.eu using federation, to get the information you need in your application.

- Finally, note that all information received by the Aggregation layer broker is also stored persistently in Cosmos and you can use that historical information for your applications. More information on this can be found in this wiki page

Currently, this is the information that you will find available in orion.lab.fi-ware.eu:

<table>
<thead>
<tr>
<th>Entity ID (pattern)</th>
<th>Entity type</th>
<th>Attributes (1)</th>
</tr>
</thead>
</table>
| OUTSMART.NODE.<id_node> | Node        | • Timelast
          |             | • Latitud
          |             | • Longitud
          |             | • Presence
          |             | • batteryCharge
          |             | • Illuminance |
| OUTSMART.RG.<id_regulator> | Regulator   | • Timelast
          |             | • Latitud
          |             | • Longitud
          |             | • ActivePower
          |             | • ReactivePower
          |             | • electricPotential
          |             | • electricCurrent |
| urn:smartsantander:testbed:<id_sensor> (2) santander:lux | | • Timelast
          |             | • temperature
          |             | • luminousFlux
          |             | • batteryCharge
          |             | • acceleration
          |             | • Latitud
<pre><code>      |             | • Longitud |
</code></pre>
<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>urn:smartsantander:testbed:&lt;id_sensor&gt; (2)</td>
<td>santander:soundacc</td>
</tr>
<tr>
<td></td>
<td>• Timelastnent</td>
</tr>
<tr>
<td></td>
<td>• sound</td>
</tr>
<tr>
<td></td>
<td>• batteryCharge</td>
</tr>
<tr>
<td></td>
<td>• acceleration</td>
</tr>
<tr>
<td></td>
<td>• Latitud</td>
</tr>
<tr>
<td></td>
<td>• Longitud</td>
</tr>
<tr>
<td>urn:smartsantander:testbed:&lt;id_sensor&gt; (2)</td>
<td>santander:traffic</td>
</tr>
<tr>
<td></td>
<td>• Timelastnent</td>
</tr>
<tr>
<td></td>
<td>• trafficIntensity</td>
</tr>
<tr>
<td></td>
<td>• median_speed</td>
</tr>
<tr>
<td></td>
<td>• average_speed</td>
</tr>
<tr>
<td></td>
<td>• Latitud</td>
</tr>
<tr>
<td></td>
<td>• Longitud</td>
</tr>
<tr>
<td>urn:smartsantander:testbed:&lt;id_sensor&gt; (2)</td>
<td>santander:sound</td>
</tr>
<tr>
<td></td>
<td>• Timelastnent</td>
</tr>
<tr>
<td></td>
<td>• sound</td>
</tr>
<tr>
<td></td>
<td>• batteryCharge</td>
</tr>
<tr>
<td></td>
<td>• Latitud</td>
</tr>
<tr>
<td></td>
<td>• Longitud</td>
</tr>
<tr>
<td>Sevilla:&lt;id_sensor&gt;</td>
<td>sevilla:water</td>
</tr>
<tr>
<td></td>
<td>• Timelastnent</td>
</tr>
<tr>
<td></td>
<td>• volume</td>
</tr>
</tbody>
</table>

Notes:

- (1) In order to know the meaning of the different attributes shown in this column have a look to the FIWARE open dataset central.
- (2) The fourth types of SmartSantander sensor share the same ID pattern.

### 2.3.5.1 Orion Context Broker version deployed in orion.lab.fi-ware.eu

This section provides historical data regarding the particular Orion Context Broker version installed in FI-LAB in a given moment, starting in 0.8.1 FI-WARE (3.2.1).

- 0.8.1 FI-WARE 3.2.1: since October 30th, 2013 to December 16th, 2013
- 0.9.0 FI-WARE 3.2.3: since December 16th, 2013 to January 17th, 2014
• 0.9.1 FI-WARE 3.3.1: since January 17th, 2014 to February 26th, 2014
  o In the period from January 17th to January 20th a 0.9.1 release candidate was running
• 0.10.1 FI-WARE 3.3.2: since February 26th, 2014 to April 11th, 2014
  o 0.10.0 was never installed in FI-LAB platform
• 0.11.0 FI-WARE 3.3.3: since April 11th, 2014 to May 5th, 2014
• 0.12.0 FI-WARE 3.4.1: since May 5th, 2014 to June, 3rd, 2014
• 0.13.0 FI-WARE 3.4.2: since June 3rd, 2014 to July 2nd, 2014
• 0.14.0 FI-WARE 3.4.3: since July 2nd, 2014

2.3.6 Experimental features
This section includes several features that hasn’t been stabilized yet. For example, the parameter names may change, etc. More information about why a given feature is considered experimental is provide in the corresponding subsection.

Once stabilized, the functionality documentation is moved to the regular sections in the manual.

2.3.6.1 Multi service tenancy
New since 0.13.0. This is an experimental feature so it may suffer some changes in the near future (in particular, we haven't decided yet if we will use the term "service" or "tenant" in the CLI option and header)

The Orion Context Broker implements a simple multitenant/multiservice model based on API namespace separation (either URL-based or HTTP header based) and logical database separation, to ease service/tenant based authorization policies provided by other FI-WARE components or third party software, e.g. the ones in the FI-WARE security framework (PEP proxy, IDM and Access Control). This functionality is activated when the -multiservice command line option is used. When used, the following modifications apply to the Orion behavior described in this manual.

The -multiservice option accepts two values: "url" or "header" (actually, "off" is also accepted, being the default value for this command line argument and meaning that no tenants/services are allowed). When "-multiservice url" is used, the CB should pay attention to the following URL patterns in the NGSI API:

• /ngsi10/<operation> and /ngsi9/<operation> (the only URLs allowed in mono-service/tenant case). In the multi-service/tenant scenario, these URL correspond to the "default service/tenant"
• /<service_or_tenant>/ngsi10/<operation> and /<service_or_tenant>/ngsi9/<operation>, e.g. "/fermin/ngsi10/updateContext" will belong to the service/tenant "fermin".

When "-multiservice header" is used, the API URL patterns are the usual ones and Orion uses the "Fiware-Service " HTTP header in the request to identify the service/tenant. If the header is not present in the HTTP request, the default service/tenant is used.
Multitenant/multiservice ensures that the entities/attributes/subscriptions of one service/tenant are "invisible" to other services/tenants. For example, queryContext on tenantA space will never return entities/attributes from tenantB space. This isolation is based on database separation, which details are described in the Installation and Administration manual.

In addition, note that when "-multiservice" is used (either the option value is "url" or "header") Orion includes the "Fiware-Service" header in the notifyContextRequest and notifyContextAvailability request messages associated to subscriptions in the given tenant/service (except for the default service/tenant, in which case the header is not present), e.g.:

```
POST http://127.0.0.1:9977/notify
Content-Length: 725
User-Agent: orion/0.13.0
Host: 127.0.0.1:9977
Accept: application/xml, application/json
Fiware-Service: t_02
Content-Type: application/xml

<notifyContextRequest>

<notifyContextRequest>
```

Regarding service/tenant name syntax, it must be a string of alphanumeric characters (and the "_" symbol). Maximum length is 20 characters, which should be enough for most use cases. In addition, note that the following strings are reserved keywords that cannot be used as service/tenant names:

- ngsi9
- ngsi10
- log
- version
- statistics
- leak
- exit
2.3.6.2  **Entity service paths**

New since 0.14.0. This is an experimental feature so it may suffer some changes in the near future (in particular, we haven’t decided yet the header name will be "ServicePath")

Orion Context Broker supports hierarchical scopes, so entities can be assigned to a scope at creation time with `updateContext` (or related convenience operation). Then, `queryContext` (and related convenience operations) can be also scoped to locate entities in the corresponding scopes.

For example, consider an Orion-based application using the following scopes (shown in the figure):

- Madrid, as first level scope
- Gardens and Districts, as second-level scope (children of Madrid)
- ParqueNorte, ParqueOeste and ParqueSur (children of Gardens) and Fuencarral and Latina (children of Districts)
- Parterre1 and Parterre2 (children of ParqueNorte)

The scope to use is specified using the “ServicePath” HTTP in update/query request. For example, to create the entity “Tree1” of type "Tree" in “Parterre1” the following ServicePath will be used:

```
ServicePath: Madrid/Gardens/ParqueNorte/Parterre1
```

In order to search for “Tree1” in that scope, the same ServicePath will be used.
Scopes are hierarchical. Thus, a queryContext with pattern entity id ".*" of type “Tree” in Madrid/Gardens/ParqueNorte will return all the trees in ParqueNorte, no matter to which parterre they belong (or even if they belong to the root Madrid/Gardens/ParqueNorte scope).

Finally, you can query for disjoint scopes, using a comma-separated list in the ServicePath header. For example, to get all trees in both ParqueNorte and ParqueOeste (but not ParqueSur) the following ServicePath would be used in queryContext request:

```
ServicePath: Madrid/Gardens/ParqueNorte, Madrid/Gardens/ParqueOeste
```

Some additional remarks:

- Limitations:
  - 10 maximum scope levels in a path
  - 10 maximum characters in each level, only alphanum and underscore allowed
  - 10 maximum disjoint scopes paths in a comma separated list in query ServicePath header (no more than 1 scope path in update ServicePath header)

- ServicePath is an optional header. It is assumed that all the entities created without using ServicePath belongs to a “null” scope (without any child or parent). All the queries without using ServicePath are resolved in the “null” scope. This behavior ensures backward compatibility to pre-0.14.0 versions.

- It is possible to have an entity with the same ID and type in different Scopes. E.g. we can create entity ID "Tree1" of type "Tree" in Madrid/Gardens/ParqueNorte/Parterre1 and another entity with ID "Tree1" of type "Tree" in Madrid/Gardens/ParqueOeste without getting any error. However, queryContext can be weird in this scenario (e.g. a queryContext in ServicePath Madrid/Gardens will returns two entities with the same ID and type in the queryContextResponse, making hard to distinguish to which scope belongs each one)

- Entities belongs to one (and only one) scope.

- The scopes entities can be combined orthogonally with the multi-service/multi-tenant functionality. In that case, each “scope tree” lives in a different service/tenant and they can use even the same names with complete database-based isolation. See figure below.
Current version doesn’t allow to change the scope to which an entity belongs through the API (a workaround is to modify the _id.servicePath field in the entities collection directly).

2.3.7 Additional information and resources

This document provides a comprehensive description of Orion Context Broker usage scenarios and capabilities. Thus, it provides all the information you need to build context-based applications using Orion Context Broker. However, if you want to expand your knowledge on the matter you can check the following resources:

- Subscribe to the Orion Context Broker support mailing list. It is a low-traffic list that will keep you updated on bug reports, new releases and general discussion on Orion Context Broker.
- **Installation and Administration Manual.** It describes administration aspects of Orion Context Broker, as well as sanity check and diagnosis procedures.
- Orion Context Broker webinar materials. You can access the last webinar presentation and video recording. They provide an introduction to basic context management context, the NGSI API and the role of Orion Context Broker in the FI-WARE architecture. In addition, the video includes a live demonstration of the broker in real operation.
- Pages for Orion-based Publish/Subscribe Context Broker GE and Configuration Manager GE in the FI-WARE catalogue. These pages provide links to the documentation and software packages, along with information about the instances installed in the FI-WARE testbed for each Use Case project and the terms and conditions under which the GEis can be used.

- The specification of the FI-WARE NGSI10 and NGSI9 RESTful API specifications. These are the "formal" NGSI API specifications used by all NGSI-based GEis in FI-WARE (as the Orion Context Broker). In addition, you can have a look at the NGSI OMA specification in which FI-WARE specifications are based. The FI-WARE NGSI binding XSD release 1.0 can be found in this file.

2.3.7.1 Sample code

This section contains code snippets (most of them contributed by external developers) that can be used as examples for programming with Orion Context Broker in different technologies. Note that Orion Context Broker evolves with time so it could happen that the code examples get obsolete in some moment (the publication date is provides as reference).

2.3.7.1.1 Java


Thanks to Alejandro Villamarin (published around October 2013):

```java
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;

//Test the Orion Broker, query for Room
String XML = "<?xml version="1.0" encoding="UTF-8"?>" +
    "<queryContextRequest>" +
    "<entityIdList>" +
    "<entityId type="Room" isPattern="true">" +
    "<id>Room.*</id>" +
    "</entityId>" +
    "</entityIdList>" +
    "<attributeList>" +
```

D.6.4.3: FI-WARE Users and Programmers Guide 268
"<attribute>temperature</attribute>" +
"</attributeList>
"</queryContextRequest>";

try {
    Client client = Client.create();
    WebResource webResource = client.resource(urlString);
    ClientResponse response = 
    webResource.type("application/xml").header("X-Auth-Token", 
    "your_auth_token").post(ClientResponse.class, XML);

    if (response.getStatus() != 200) {
        System.err.println(response.getEntity(String.class));
        throw new RuntimeException("Failed : HTTP error code : " + 
response.getStatus());
    }

    System.out.println("Output from Server .... \n");
    String output = response.getEntity(String.class);
    System.out.println(output);

} catch (Exception e) {
    System.err.println("Failed. Reason is " + e.getMessage());
}

2.3.7.1.2 JavaScript
Using JQuery AJAX, thanks to Marco Vereda Manchego (original post) (published around October 2013):

function capture_sensor_data(){
  var contentTypeRequest = $.ajax({
      data: '<?xml version="1.0" encoding="UTF-8"?>'
<queryContextRequest>
  <entityIdList>
    <entityId type = "Sensor" isPattern="true">
      <id>urn:smartsantander:testbed:*</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>Latitud</attribute>
    <attribute>Longitud</attribute>
  </attributeList>
</queryContextRequest>

{ type: 'POST',
  dataType: 'xml',
  contentType: 'application/xml',
  headers: { 'X-Auth-Token': 'you_auth_token' }
};

contentTypeRequest.done(function(xml){
  var latitud = new Array();
  var longitud = new Array();
  var i = 0;
  var len = $(xml).find("contextAttribute").children().size();
  $(xml).find('contextAttribute').each(function(){
    if ( $(this).find('type').text() == "urn:x-ogc:def:phenomenon:IDAS:1.0:latitude" && $(this).find('contextValue').text() != "" )
      {
        latitud[i] = $(this).find('contextValue').text(); i=i+1;
      }
  });
});
2.3.7.1.3 Arduino

Thanks to Enrique Hernandez Zurita, using Orion 0.11.0 and 0.12.0 (published around June 2014):

```cpp
#include <SPI.h>
#include <WiFi.h>
#include <WString.h>
char ssid[] = "SSID";
char pass[] = "CONTRASEÑA";

int status = WL_IDLE_STATUS;
IPAddress server(130,206,82,115);
WiFiClient client;
```
String line="";
int value=0;
int led=9;

void setup() {
    pinMode(led,OUTPUT);
    Serial.begin(9600);
    while (!Serial) {
    }
    if (WiFi.status() == WL_NO_SHIELD) {
        Serial.println("WiFi shield not present");
        while(true);
    }
    while (status != WL_CONNECTED) {
        Serial.print("Attempting to connect to SSID: ");
        Serial.println(ssid);
        status = WiFi.begin(ssid, pass);
        delay(5000);
    }
    Serial.println("Connected to wifi");
    Serial.println("\nStarting connection to server...");
    if (client.connect(server, 1026)) {
        Serial.println("connected to server");
        client.println("POST /NGSI10/queryContext HTTP/1.1");
        client.println("Host: 130.206.82.115:1026");
        client.println("User-Agent: Arduino/1.1");
        client.println("Connection: close");
client.println("Content-Type: application/xml");
client.println("Content-Length: ");
client.println("227");
client.println();
client.println("<?xml version="1.0" encoding="UFT-8"?>");
client.println("<queryContextRequest>");
client.println("<entityIdList>");
client.println("<entityId type="UPCT:ACTUATOR" isPattern="false">");
client.println("<id>UPCT:ACTUATOR:1</id>");
client.println("</entityId>");
client.println("</entityIdList>");
client.println("<attributeList/>");
client.println("</queryContextRequest>");
client.println();
Serial.println("XML Sent");

}else{Serial.println("Impossible to connect");}
}

void loop() {
  if(value==1){digitalWrite(led, HIGH);}

  while (client.available()) {
    char c = client.read();
    Serial.write(c);
    line=line + c;
    if(c==10){
      value=searchValue(line,value);
    }
  }
}
if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting from server.");
    Serial.println(value);
    client.stop();
    while(true);
}

int searchValue(String s, int i) {
    int beginning, ending;
    String val;
    beginning = s.indexOf('>');
    ending = s.indexOf('<', beginning + 1);
    if (s.startsWith("contextValue", s.indexOf('<') + 1)) {
        val = s.substring(beginning + 1, ending);
        return val.toInt();
    } else {
        return i;
    }
}
3 Publish/Subscribe GE - Context Awareness Platform - User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

3.1 Introduction

Context Awareness Platform (referred as "CAP" in the following) implementing Publish/Subscribe Context Broker is able to manage context data by means of both ContextML/CQL and OMA NGSI interfaces. This document describes both CAP interfaces with invocation examples.

3.1.1.1 Background and Details


3.2 User guide

CAP core engine supporting ContextML/CQL is a back-end component, so no user guide is provided.

3.3 Programmer's guide

3.3.1 OAuth2 authentication

In FI-WARE testbed reference installation, OAuth2 authentication has been enabled from R3.2.3. This leads to the following important modifications:

- Every endpoint in the following paragraphs is exposed on ssl standard port 443, therefore URLs should be modified from http to https
- Every request should include an HTTP header as follows:
  - header name: "Authorization"
  - header value: "Bearer [oauth_token_value]"

The OAuth token [oauth_token_value] should be obtained from the FI-WARE Identity Management platform, as described in http://forge.fi-ware.org/plugins/mediawiki/wiki/fiware/index.php/Identity_Management_-_KeyRock_-_User_and_Programmers_Guide#User_Guide Standard access on port 80 without OAuth authentication is temporarily available for backward compatibility but is deprecated.
3.3.2 ContextML/CQL API


3.3.2.1 The ContextML language for context representation

ContextML is an XML-based language defined as a common representation for exchanging context data. Context information could refer to different domains, for this reason context information has been divided into context scopes, group of context attributes related to the same information category.

For example, the context scope named position groups the following attributes:

- latitude and longitude (in degrees)
- accuracy (in meters)
- locMode (ex. GPS), it is the technology used to evaluate the position.

The ContextML schema is composed by:

- ctxEls: contain the context data;
- ctxResp: contain a generic response from a platform module

A ContextML document can contain one or more context elements (ctxEl). Each context elements contains the following information:

- contextProvider: a unique identifier for the provider of the data;
- entity: the identifier and the type of the entity to which the data is related;
- scope: the scope to which the context data belongs;
- timestamp and expires: respectively, the time in which the response was created, and the expiration time of the data part;
- dataPart: is the part of the document which contains actual context data. The information is represented by a list of context parameters through the <par> element. They can be grouped through the <parS> element (“parameter struct”) and/or <parA> element (“parameter array”) if necessary.

3.3.2.2 ContextUpdate: how to send a context element within CAP

This method allows the upload of new context data into the CAP. The platform checks that the updated scopes are defined and stores the context values into the cache.
An HTTP POST data request should be sent to:

http://pubsub.testbed.fi-ware.org/CB/ContextBroker/contextUpdate

The POST body should contain a ContextML message containing the context elements to be updated. The Content-Type header must be set to "text/xml" and the http content length must be set accordingly. Since the ContextML information is contained in the POST body, no request parameters are needed. Here is an example of body, for a context update of "position" for a device:

```
<?xml version="1.0" encoding="UTF-8"?>
<contextML xmlns="http://ContextML/1.7"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ContextML/1.7 ../ContextML-1.7.xsd">
    <ctxEls>
        <ctxEl>
            <contextProvider id="MyClient" v="1.2.1"/>
            <entity id="123456789123" type="imei"/>
            <scope>position</scope>
            <timestamp>2012-05-20T11:12:19+01:00</timestamp>
            <expires>2012-05-20T11:21:22+01:00</expires>
            <dataPart>
                <par n="latitude">45.11045277777778</par>
                <par n="longitude">7.67525194444445</par>
                <par n="accuracy">50</par>
                <par n="locMode">GPS</par>
            </dataPart>
        </ctxEl>
    </ctxEls>
</contextML>
```

Normally the CAP response is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```
3.3.2.3  GetContext: how to retrieve a context element within CAP

This method allows the retrieval of context elements within the CAP. The platform searches for valid context elements in cache, otherwise tries to update them with the help of context providers. Updated context information is stored into the cache.

An HTTP GET data request should be sent to the server, containing the entity and the comma separated list of required scopes (scopeList parameter).
Here is an example to require the scope “position” of a device:


The CAP answer with the required context element, if available, as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contextML xmlns="http://ContextML/1.7"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ContextML/1.7 ../ContextML-1.7.xsd">
  <ctxEls>
    <ctxEl>
      <contextProvider id="MyClient" v="1.2.1"/>
      <entity id="123456789123" type="imei"/>
      <scope>position</scope>
      <timestamp>2012-05-20T11:12:19+01:00</timestamp>
      <expires>2012-05-20T11:21:22+01:00</expires>
      <dataPart>
        <par n="latitude">45.11045277777778</par>
        <par n="longitude">7.675251944444445</par>
        <par n="accuracy">50</par>
        <par n="locMode">GPS</par>
      </dataPart>
    </ctxEl>
  </ctxEls>
</contextML>
```

If the required context element is not available, the response will be similar to:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```
3.3.2.4 *GetContextQL: how to subscribe to context notification within CAP*

This method allows the subscription to notifications of context elements within the CAP. The platform forwards every context elements matching the required notification criteria to the URL specified by the client.

An HTTP POST request should be sent to the CAP endpoint (with Content-Type="application/x-www-form-urlencoded"):

http://pubsub.testbed.fi-ware.org/CB/ContextBroker/getContextQL

containing the following mandatory url-encoded parameters:

- `cqlReq`: containing the CQL message (for details on CQL language see *CQL_API*),
- `callbackUrl`: containing the url which the client should receive the notification messages on.

An example of body with parameters (which should be url-encoded):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contextQL>
  <ctxQuery>
    <action type="SUBSCRIBE" />
    <entity>imei|123456789123</entity>
    <scope>cell</scope>
  </ctxQuery>
</contextQL>
```
The CAP answers with a message as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contextML xmlns="http://ContextML/1.7"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ContextML/1.7 .../ContextML-1.7.xsd">
  <ctxResp>
    <contextProvider id="CB" v="1.4.3"/>
    <timestamp>2012-11-13T16:35:35+01:00</timestamp>
    <method>getContextQL</method>
    <resp status="OK" code="200"/>
    <validity>180</validity>
    <subId>62</subId>
  </ctxResp>
</contextML>
```

The subscriber should check the validity, which could be shorter than the required one.

### 3.3.2.4.1 Notifications to the client
Notification to the callback URL will be HTTP POST (Content-Type="application/x-www-form-urlencoded"), the body is a single parameter ctxData containing the ContextML message, as follows (ex.):

```xml
ctxData=<?xml version="1.0" encoding="UTF-8"?>
<contextML>
  <ctxEls>
    <ctxEl subId="62">
      <contextProvider id="provName" v="1.13"/>
    </ctxEl>
  </ctxEls>
</contextML>
```
The client should answer with an acknowledge message to every notification, as the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contextML>
    <ctxResp>
        <resp status="OK" code="200"/>
    </ctxResp>
</contextML>
```

Notice that after a predefined number of missing acknowledge messages from the subscriber, the CAP deactivates the related subscription.

### 3.3.2.4.2 How to renew CQL subscriptions

Subscriptions may be renewed by means of the subId returned in the original CAP subscription response, sending a CQL message as the following example (for subId=62):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contextQL>
    <ctxQuery>
        <action type="SUBSCRIBE"/>
        <validity>180</validity>
        <subId>62</subId>
    </ctxQuery>
</contextQL>
```
3.3.3 NGSI API

A partial implementation of NGSI interface, described in OMA standard document http://www.openmobilealliance.org/Technical/release_program/docs/NGSI/V1_0-20101207-C/OMA-TS-NGSI_Context_Management-V1_0-20100803-C.pdf, has been realized by means of a wrapper on top of ContextML/CQL core engine component. NGSI interface is available by means of both standard methods and convenience functions described in FIWARE open specification FIWARE_NGSI_Open_RESTful_API_Specification_(PRELIMINARY). NGSI data structure have been modeled in XML or JSON language, compliant to supplied xsd schema and NGSI specification. In the following examples it is assumed XML as input/output format, unless otherwise specified.

It should be noted that since NGSI and ContextML/CQL interfaces have a slightly different approach to context management, this implementation of NGSI functions undergoes some limitations which will be described in details in the paragraph #Limitation_of_current_NGSI_implementation. Here is a list of the most important ones related to context model:

- AttributeDomain and related metadata are not supported
- In EntityId data structure the Pattern parameter is not supported
- In ContextAttribute data structure the metadata Timestamp, Expires and Source are optional in input as required by specifications, but default values are introduced when context is retrieved.

- The attributeID is not supported, thus every operation involving it (e.g. updateContext APPEND) is not available; it means that, for a specified entityId and context attribute name, only one context attribute at a time could be retained in CAP cache and a new context attribute will overwrite the old one with the same attribute name.

In next releases some limitations may be overcome. Furthermore, some assumption has been made, in order to match NGSI data structures with related ContextML ones, as described in the next paragraph.

3.3.3.1 **NGSI to ContextML: mapping**

Knowing mapping assumptions is useful if context data are updated with NGSI method of the wrapper and retrieved by means of the ContextML/CQL methods or viceversa. The following applies:

- The mapping between NGSI EntityId and ContextML Entity is as follows:

```
<entityId type="urn:[ctxML_entity_type]">
  <id>urn:[ctxML_entity_type]:[ctxML_entity_id]</id>
</entityId>
```

Ex.: (NGSI entityId.id) "urn:username:sergio" -->(ContextML entity) "username|sergio"

- The ContextAttribute name is exactly the name of ContextML "scope".

3.3.3.2 **Specifications on contextValue format**

The contextValue data structure, defined in OMA NGSI specification as a generic xs:anyType object, is supported only with the following formats:

- Native datatype: ex. simple strings

```
<contextValue>222-1-61101-7065</contextValue>
```

- Set of simple parameters: ex.

```
<contextValue>
  <val1>222-1-61101-7065</val1>
  <val2>222-1-61101-7066</val2>
</contextValue>
```
- **Data structures:** ex.

```xml
<contextValue>
  <cgi>
    <val1>222-1-61101-7065</val1>
    <val2>222-1-61101-7066</val2>
  </cgi>
</contextValue>
```

- **Array of simple parameters:** ex.

```xml
<contextValue>
  <cgi>
    <cgi>222-1-61101-7065</cgi>
    <cgi>222-1-61101-7066</cgi>
    <cgi>222-1-61101-7067</cgi>
  </cgi>
</contextValue>
```

- **Array of data structures:** ex.

```xml
<contextValue>
  <cgi>
    <cgi>
      <val1>222-1-61101-7065</val1>
      <val2>222-1-61101-7066</val2>
    </cgi>
    <cgi>
      <val1>222-1-61101-7067</val1>
      <val2>222-1-61101-7068</val2>
    </cgi>
  </cgi>
</contextValue>
```
Note that array elements must have the same name of the array itself (in the example: "cgi"). Since the parsing of contextValue data structure is made without an available xsd schema, it is important to not insert white spaces between tags (except for text nodes), which would be interpreted as XML nodes leading to errors. For the same reason, if the serialization is made using commons libraries, auto indent options should be avoided. For instance, the last example should be sent as follows:

```
<contextValue><cgi><cgi><val1>222-1-61101-7065</val1><val2>pippo</val2></cgi></contextValue>
```

### 3.3.3.3 Exchanged messages: some general information

The communication between applications or clients and CAP is done with HTTP requests. Since the platform is able to process both JSON and XML format, every request should specify the body content type ("Content-Type" HTTP header) and the desired output format by means of "Accept" HTTP header, set to "application/xml;charset=UTF-8" or "application/json;charset=UTF-8" (charset information is optional). If Accept header is not present, CAP returns a response with the same content-type of the request.

For standard OMA NGSI methods, requests are always HTTP POST, while convenience functions use POST, PUT, DELETE or GET method depending on the specific operation to be performed, according to specifications based on RESTful paradigm.

### 3.3.3.4 NGSI by means of XML

In the following some examples of requests/responses for main available functions are described. The [FIWARE_NGSI_Open_RESTful_API_Specification_(PRELIMINARY)](http://www.fi-ware.org/oma-xml) and related OMA NGSI specification should be kept as a reference for further details. For XML format, client should specify correct HTTP headers:

- Content-Type="application/xml"
- Accept="application/xml" (optionally, if not present the CAP will answer in the same format of the request)

#### 3.3.3.4.1 How to retrieve context: queryContext method

The queryContext method allows to retrieve context information available on the CAP. Almost every functionality is supported, included restrictions on context values or metadata. Entity pattern are not supported at the time being, as well as the "scope" optional parameter inside Restriction.

An HTTP POST data request should be sent to:

```
http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI10/queryContext
```

The POST body should contain an XML QueryContextRequest message containing the context attributes to be queried. The request’s contentType must be set to "application/xml;charset=UTF-8" and the http
content length must be set accordingly. Since the information is contained in the POST body, no request parameters are needed.

Example of request message:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId>
      <id>urn:username:cristinaF</id>
    </entityId>
    <entityId>
      <id>urn:username:sergio</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>cell</attribute>
  </attributeList>
  <restriction>
    <attributeExpression>//contextValue[starts-with(cgi,'222')]</attributeExpression>
  </restriction>
</queryContextRequest>
```

Example of response:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="urn:username">
```
<id>urn:username:cristinaF</id>
</entityId>
<contextAttributeList>
<contextAttribute>
<name>cell</name>
<contextValue>
<cgi>222-1-61101-7065</cgi>
</contextValue>
</contextAttribute>
<contextMetadata>
<name>Timestamp</name>
<type>xsd:dateTime</type>
<value>2012-10-17T12:09:31+02:00</value>
</contextMetadata>
<contextMetadata>
<name>Expires</name>
<type>xsd:dateTime</type>
<value>2012-10-17T12:11:31+02:00</value>
</contextMetadata>
<contextMetadata>
<name>Source</name>
<value>teamlife_1.0</value>
</contextMetadata>
</metadata>
</contextAttribute>
<contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>Ok</reasonPhrase>
3.3.3.4.2 How to update or delete context: updateContext method

The updateContext method is used to send updated context to the CAP. In this implementation UPDATE and DELETE update actions are available (APPEND is not supported).

An HTTP POST data request should be sent to:

```
http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI10/updateContext
```

The POST body should contain an XML UpdateContextRequest message containing the context elements to be updated. The request’s contentType must be set to "application/xml;charset=UTF-8" and the http content length must be set accordingly. Since the information is contained in the POST body, no request parameters are needed.

**UPDATE action**

Here is an example of UPDATE action, for "cell" context attribute, related to user "sergio":

```xml
<contextElementResponse>
  <contextElement>
    <entityId>
      <id>urn:username:sergio</id>
    </entityId>
    <contextAttributeList>
      <contextAttribute>
        <name>cell</name>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
  <statusCode>
    <code>404</code>
    <reasonPhrase>ContextElement not found</reasonPhrase>
  </statusCode>
</contextElementResponse>
```
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
    <contextElementList>
        <contextElement>
            <entityId>
                <id>urn:username:sergio</id>
            </entityId>
            <contextAttributeList>
                <contextAttribute>
                    <name>cell</name>
                    <contextValue>
                        <cgi>222-1-61101-7066</cgi>
                    </contextValue>
                    <metadata>
                        <contextMetadata>
                            <name>Timestamp</name>
                            <value>2012-06-13T15:04:52+01:00</value>
                        </contextMetadata>
                        <contextMetadata>
                            <name>Expires</name>
                            <value>2012-06-13T15:06:52+01:00</value>
                        </contextMetadata>
                        <contextMetadata>
                            <name>Source</name>
                            <value>teamlife_1.0</value>
                        </contextMetadata>
                    </metadata>
                </contextAttribute>
            </contextAttributeList>
        </contextElement>
    </contextElementList>
</updateContextRequest>
Normally, the CAP will respond as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<updateContextResponse>
    <contextResponseList>
        <contextElementResponse>
            <contextElement>
                <entityId>
                    <id>urn:username:sergio</id>
                </entityId>
                <contextAttributeList>
                    <contextAttribute>
                        <name>cell</name>
                        <contextValue>
                            <cgi>222-1-61101-7066</cgi>
                        </contextValue>
                        <metadata>
                            <contextMetadata>
                                <name>Timestamp</name>
                                <value>2012-06-13T15:04:52+01:00</value>
                            </contextMetadata>
                            <contextMetadata>
                                <name>Expires</name>
                                <value>2012-06-13T15:06:52+01:00</value>
                            </contextMetadata>
                        </metadata>
                    </contextAttribute>
                </contextAttributeList>
            </contextElement>
        </contextElementResponse>
    </contextResponseList>
</updateContextResponse>
```
In case of error in one of the updated context elements, the response will be of the following type:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<updateContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId>
          <id>urn:username:sergio</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>cell</name>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```
**DELETE action**

Here is an example of DELETE action, for "cell" context attribute, related to user "sergio":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId>
        <id>urn:username:sergio</id>
      </entityId>
      <contextAttributeList>
        <contextAttribute>
          <name>cell</name>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>DELETE</updateAction>
</updateContextRequest>
```

Normally, the server will respond as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<updateContextResponse>  
```

```xml```
In case of error, the response will be similar to UPDATE action case.

3.3.3.4.3 HOW TO RECEIVE CONTEXT NOTIFICATIONS

CAP allows applications to subscribe to context changes or updates, in order to receive notification of context elements at every updates, at specified time intervals or when a condition is verified. Current implementation support subscriptions without condition (i.e. every related context element received by the CAP is forwarded to subscribed applications), ONTIMEINTERVAL and ONCHANGE ones.

subscribeContext method
This method allows the subscription to context data provisioning within the CAP.

An HTTP POST data request should be sent to:

http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI10/subscribeContext

or using the convenience function:
The POST body should contain an XML SubscribeContextRequest message containing the context entities and attributes to be notified when new values are received by the CAP, with or without a condition field. The request’s contentType must be set to "application/xml;charset=UTF-8" and the http content length must be set accordingly. Since the information is contained in the POST body, no request parameters are needed.

Here is some examples of subscription requests:

- **without condition**, to be notified for new values of "cell" context attribute of user "cristinaF"
  (also an example of restriction is included):

  ```xml
  <subscribeContextRequest>
  <entityIdList>
    <entityId type="urn:username" isPattern="false">
      <id>urn:username:cristinaF</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>cell</attribute>
  </attributeList>
  <reference>http://127.0.0.1/ngsicbapi/notifyContext</reference>
  <duration>PT300S</duration>
  <restriction>
    <attributeExpression>//contextAttribute/metadata/contextMetadata[name='Timestamp']/value</attributeExpression>
  </restriction>
  </subscribeContextRequest>
  ```

- **ONTIMEINTERVAL**, to be notified for values of "cell" context attribute of user "cristinaF" at a specified time interval:

  ```xml
  <subscribeContextRequest>
  <entityIdList>
    <entityId type="urn:username" isPattern="false">
      <id>urn:username:cristinaF</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>cell</attribute>
  </attributeList>
  <reference>http://127.0.0.1/ngsicbapi/notifyContext</reference>
  <duration>PT300S</duration>
  <restriction>
    <attributeExpression>//contextAttribute/metadata/contextMetadata[name='Timestamp']/value</attributeExpression>
  </restriction>
  </subscribeContextRequest>
  ```
ONCHANGE, to be notified for values of "cell" context attribute of user "cristinaF" when either "cell" or "position" context attribute is updated with a different context value. The ONCHANGE subscription is correctly processed only for context attributes with a single context value (base type). If needed, ContextQL subscriptions allow finer specification on changes related to a specific context value.
• **ONVALUE**, to be notified for values of "position" context attribute (if available) of user "cristinaF" when "cell" context attribute has (or contains a parameter with) a specified value. The XPath restriction should always contain the context attribute name triggering the notification (in the example: "/contextAttribute[name='cell']..." ), which can be the same one to be notified (e.g notify "cell" when "cell" has a context value of X). Multiple trigger context attributes are not supported in the current version.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<subscribeContextRequest>
  <entityIdList>
    <entityId type="urn:username" isPattern="false">
      <id>urn:username:cristinaF</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>position</attribute>
  </attributeList>
</subscribeContextRequest>
```
For complex context attribute values containing parameters, an example of ONVALUE restriction could be (context value containing a "cgi" parameter whose value starts with "222"):

```xml
<restriction>//contextAttribute[name='cell'][contextValue[starts-with('cgi', '222')]]</restriction>
```

If everything is OK, the server returns a message similar to the following:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<subscribeContextResponse>
  <subscribeResponse>
    <subscriptionId>a51bddaf-e2c4-481a-ace8-d45e1d725f44</subscriptionId>
    <duration>PT300S</duration>
  </subscribeResponse>
</subscribeContextResponse>
```

Here is an example of message returned in case of error:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<subscribeContextResponse>
```
<subscribeError>
  <errorCode>
    <code>472</code>
    <reasonPhrase>Invalid parameter</reasonPhrase>
  </errorCode>
</subscribeError>
</subscribeContextResponse>

CAP will notify every related context update with an HTTP POST to the URL specified in "reference" field, with a body as the following:

<notifyContextRequest>
  <subscriptionId>a51bddaf-e2c4-481a-ace8-d45e1d725f44</subscriptionId>
  <originator>NGSIPubSub</originator>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId>
          <id>urn:username:cristinaF</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>cell</name>
            <contextValue>
              <cgi>222-1-61101-7066</cgi>
            </contextValue>
            <metadata>
              <contextMetadata>
                <name>Timestamp</name>
                <type>xsd:dateTime</type>
                <value>2012-06-13T15:04:52+01:00</value>
              </contextMetadata>
            </metadata>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
    </contextElementResponse>
  </contextResponseList>
</notifyContextRequest>
The notified application is expected to answer with an acknowledge message as the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<notifyContextResponse>
  <responseCode>
    <code>200</code>
    <reasonPhrase>Ok</reasonPhrase>
    <details>a</details>
  </responseCode>
</notifyContextResponse>
```
After a number of notification failures, the subscription is deactivated by the CAP, in order to avoid excessive overload.

**updateContextSubscription method**

A subscription may be renewed or updated sending an HTTP POST message to:

```
http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI10/updateContextSubscription
```

with the following body:

```
<?xml version="1.0" encoding="UTF-8"?>
<updateContextSubscriptionRequest>
  <duration>PT5M</duration>
  <restriction>
    <attributeExpression>//contextMetadata[name='Timestamp'][starts-with(value,'2012-06-21')]</attributeExpression>
  </restriction>
  <subscriptionId>9212ce4b-0c21-4479-be42-9e2b7ad43c53</subscriptionId>
</updateContextSubscriptionRequest>
```

If the "restriction" field is not present, the former restriction still applies. CAP responds with a message as:

```
<?xml version="1.0" encoding="UTF-8"?>
<subscribeContextResponse>
  <subscribeResponse>
    <subscriptionId>9212ce4b-0c21-4479-be42-9e2b7ad43c53</subscriptionId>
    <duration>PT5M</duration>
  </subscribeResponse>
</subscribeContextResponse>
```

As for subscribeContext response, the duration may be shorter than the one required by the application. Note that if a subscription is already expired it cannot be renewed.
3.3.3.4.4 How a provider can register context availability: registerContext method

By means of NGSI-9 registerContext method, a provider declares to the CAP that is capable of providing a list of context attributes related to one or more entities. When the CAP receives a request of a context element, it searches in cache and, if the required element is not present, it looks for providers able to return it, and invokes them (by means of a queryContext message) in order to get the required context element. The registerContext could be done on all context attributes, omitting the ContextAttributeList parameter. It could be done on a simple entity pattern as "urn:[entityType]:*", but this kind of NGSI registration is correctly managed for retrieval of context during the processing of a subscription or queryContext related to a single entity (not "urn:[entityType]:*"), for intrinsic limitation of ContextML CAP, which is not able to forward such "generic" request to registered providers. A registration may be performed sending an HTTP POST message to:

http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI9/registerContext

Making a new registration

In order to make a new registration, the body of the request is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId>
          <id>urn:username:sergio</id>
        </entityId>
        <entityId>
          <id>urn:imei:cristina</id>
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>cell</name>
        </contextRegistrationAttribute>
        <contextRegistrationAttribute>
          <name>position</name>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
    </contextRegistration>
  </contextRegistrationList>
</registerContextRequest>
```
The message announces to the CAP that:

- the first provider is able to return context attributes "cell" and "position" of entities "urn:username:sergio" and "urn:username:cristina"
- the second provider is able to return every context attributes related to entities "urn:username:sergio1" and "urn:username:cristina1"

The providerApplication field contains the URI to which the provider accepts the NGSI-10 queryContext request by the CAP. When needed, in order to retrieve context from provider, the CAP will send an HTTP POST method to endpoint [providerApplication_value]/queryContext, with a queryContextRequest body (as seen in queryContext paragraph).

The CAP responds with a message containing the assigned registrationId and the confirmed duration:
Renewing/updating an already existing registration

The provider may renew the registration simply adding the registrationId to the request, as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<registerContextRequest>
    <contextRegistrationList>
        <contextRegistration>
            <entityIdList>
                <entityId>
                    <id>urn:username:sergio</id>
                </entityId>
                <entityId>
                    <id>urn:imei:cristina</id>
                </entityId>
            </entityIdList>
            <contextRegistrationAttributeList>
                <contextRegistrationAttribute>
                    <name>cell</name>
                </contextRegistrationAttribute>
                <contextRegistrationAttribute>
                    <name>position</name>
                </contextRegistrationAttribute>
            </contextRegistrationAttributeList>
        </contextRegistration>
    </contextRegistrationList>
</registerContextRequest>
```
If the registrationId matches a current registration, the CAP updates the registration data with the new provided ones. If the registrationId is unknown, an error is returned to the provider.

3.3.3.4.5 **Convenience functions: some examples**

Besides the standard OMA NGSI methods, a set of convenience functions is available. They are RESTful methods built on top of standard OMA NGSI methods, and allow a subset of operations in a easier way.

In some cases, as already seen in [subscribeContext method](#subscribeContext_method) paragraph, exchanged messages are the same as for corresponding standard OMA NGSI method. Here below some further examples are reported, necessarily not exhaustive. In the following, unless otherwise specified, it is assumed the base URL:

http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI10

**How to retrieve context**

The HTTP GET method is used to retrieve context:
Future Internet Core Platform

- `/contextEntities/{entityId}` retrieves all available context attributes related to the specified entityId, ex:
  - `/contextEntities/urn:username:sergio`

- `/contextEntities/{entityId}/attributes/{attributeName}` retrieves specified context attribute related to the specified entityId, ex:
  - `/contextEntities/urn:username:sergio/attributes/cell`

- `/contextEntityTypes/{typeName}` retrieves all available context attributes related to the entities with the specified entity type name, ex:
  - `/contextEntities/urn:username`

- `/contextEntityTypes/{typeName}/attributes/{attributeName}` retrieves specified context attribute related to the entities with the specified entity type name, ex:
  - `/contextEntities/urn:username/attributes/cell`

How to update context

The HTTP PUT method is used to update context. The body of the message contains the data structure with the context attributes to be updated.

- `/contextEntities/{entityId}` updates one or more context attributes related to the specified entityId, ex:
  - `/contextEntities/urn:username:sergio`, an example of body is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<updateContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>cell</name>
      <contextValue>10</contextValue>
      <metadata>
        <contextMetadata>
          <name>Timestamp</name>
          <type>xsd:dateTime</type>
        </contextMetadata>
      </metadata>
    </contextAttribute>
  </contextAttributeList>
</updateContextElementRequest>
```
How to delete context

The HTTP DELETE method is used to remove context attributes from CAP cache.

- /contextEntities/{entityId}
  removes all context attributes related to the specified entityId, ex.:
    - /contextEntities/urn:username:sergio

- /contextEntities/{entityId}/attributes/{attributeName}
  removes a specified context attribute related to the specified entityId, ex.:
    - /contextEntities/urn:username:sergio/attributes/cell

How to renew or delete a context subscription

The HTTP PUT method is used to renew a context subscription. The body of the message contains the data structure with the context subscription data to be updated, the specified subId should match with the value in the data structure, otherwise CAP returns an error.

- /contextSubscriptions/{subId}
  updates the specified context subscription, ex.:
    - /contextSubscriptions/9212ce4b-0c21-4479-be42-9e2b7ad43c53, an example of body is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<updateContextSubscriptionRequest>
  <duration>PT5M</duration>
  <restriction>
    <attributeExpression>//contextMetadata[name='Timestamp'][starts-with(value,'2012-06-21')]</attributeExpression>
  </restriction>
  <subscriptionId>9212ce4b-0c21-4479-be42-9e2b7ad43c53</subscriptionId>
</updateContextSubscriptionRequest>
```
The HTTP DELETE method is used to remove an active subscription before its natural expiration time (no body is required).

- /contextSubscriptions/{subId}

  removes the specified context subscription, ex.:
  - /contextSubscriptions/9212ce4b-0c21-4479-be42-9e2b7ad43c53

**How to register context availability**

NGSI-9 registration may be done also by means of convenience functions, sending an HTTP POST message to the REST endpoint depending on the entity/contextAttribute couple. In this case, since method belongs to NGSI-9 interface, the base URL is:

http://pubsub.testbed.fi-ware.org/ngsicbapi/NGSI9

- /contextEntities/{entityId}/attributes/{attributeName}
  - /contextEntities/urn:username:sergio/attributes/position

with the body:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<registerProviderRequest>
  <duration>PT3000S</duration>
  <providingApplication>http://localhost:8080/ngsicbapi/testProvider</providingApplication>
</registerProviderRequest>
```

The CAP responds with a registerContextResponse message as seen for standard NGSI-9 registerContext.

Similarly, renewal by means of convenience functions is made with an HTTP POST request to the same endpoint, as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<registerProviderRequest>
  <duration>PT3000S</duration>
  <providingApplication>http://localhost:8080/ngsicbapi/testProvider</providingApplication>
  <registrationId>b56a91aa-a552-487a-ac8b-1ed025969c4e</registrationId>
</registerProviderRequest>
```
The CAP response will be the same as for new registrations.

3.3.3.5 **NGSI by means of JSON**

Here are reported two simple examples of requests made using JSON format (JSON messages used for further methods may be easily obtained from above XML examples). Client should specify correct HTTP headers:

- Content-Type="application/json"
- Accept="application/json" (optionally, if not present the CAP will answer in the same format of the request)

3.3.3.5.1 **queryContext method**

The queryContext method allows to retrieve context information available on the CAP. Every functionality is supported, included restrictions on context values or metadata (fatto salvo quanto detto sui pattern nell’entityId). The "scope" optional parameter inside Restriction is not supported.

Example of request message:

```json
{
  "queryContextRequest": {
    "entityIdList": {
      "entityId": [
        {
          "type": "urn:username",
          "id": "urn:username:sergio"
        }
      ],
      "attributeList": {
        "attribute": [
          "cell",
          "cell1"
        ]
      }
    }
  }
}
```
Example of response:

```json
{
  "queryContextResponse": {
    "contextResponseList": {
      "contextElementResponse": [
        {
          "statusCode": {
            "code": 200,
            "reasonPhrase": "Ok"
          },
          "contextElement": {
            "entityId": {
              "id": "urn:username:sergio",
              "type": "urn:username"
            },
            "contextAttributeList": {
              "contextAttribute": [
                {
                  "name": "cell",
                  "metadata": {
                    "contextMetadata": [
                      {
                        "name": "Timestamp",
                        "value": "2012-07-16T10:06:04+02:00",
                      }
                    ]
                  }
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```
The updateContext method is used to send context to be updated to the CAP. In this implementation UPDATE and DELETE update actions are available (APPEND is not supported).

Example of request message:

```json
{
  "updateContextRequest": {
    "contextElementList": {
      "contextElement": [
        {
          "entityId": {
            "id": "urn:username:sergio",
            "type": "urn:username"
          },
          "contextAttributeList": {
            "contextAttribute": [
```

{
    "name": "cell",
    "metadata": {
        "contextMetadata": [
            {
                "name": "Timestamp",
                "value": "2012-06-13T15:04:52+01:00"
            },
            {
                "name": "Expires",
                "value": "2012-06-13T17:05:52+01:00"
            },
            {
                "name": "Source",
                "value": "TeamLife_1.13"
            }
        ]
    },
    "contextValue": {
        "cgi": "222-1-61101-7066"
    }
}
}
Example of response:

```json
{
    "updateContextResponse": {
        "contextResponseList": {
            "contextElementResponse": [
                {
                    "statusCode": {
                        "code": 200,
                        "reasonPhrase": "Ok"
                    },
                    "contextElement": {
                        "entityId": {
                            "id": "urn:username:sergio",
                            "type": "urn:username"
                        },
                        "contextAttributeList": {
                            "contextAttribute": [
                                {
                                    "name": "cell"
                                }
                            ]
                        }
                    }
                }
            ]
        }
    }
}
```
3.3.3.6 **Limitation of current NGSI implementation**

The current (R3.3) reference implementation of the NSGI interface has the following limitations:

- the AttributeDomain and relative meta-data are not supported as well as the convenience functions using these data;
- the OperationScope e.g. restriction over a particular group of sensors to be interrogated on geographic basis, optional as restriction, is not supported;
- the Pattern, including Simple, of the entityId is not supported, but the convenience functions of /contextEntityTypes using entity of entityType presented also in CQL, are available. Thus it is possible to execute queryContext of all the context attributes of an entityId of the same entityType;
- the context meta-data Timestamp and Expires are mandatory. If not present in the ctxUpdate, the default values of \[ts=[current\_time]\] and \[exp=[ts+1day]\] will be inserted;
- the updateContext method is available only for operations UPDATE and DELETE. The operation APPEND is not supported. The UPDATE operation overwrites any previously stored context attribute value, by default;
- the convenience function GET /contextEntityTypes/[entityType]/attributes is not supported, while the same function with a specified attribute name is supported. i.e. GET /contextEntityTypes/[entityType]/attributes/[attrName];
- the subscription Duration field is mandatory and could be limited by the CAP;
- the ONCHANGE condition subscription is supported on context attributes with simple context values only;
- the updateSubscription does not update the condition itself;
- the error codes are supported but to be improved yet;
- the contextValue tag shall not contain blank spaces, indents or end-lines (this should not be a problem if using common XML libraries for the client development). It supports the structures of simple types such as of simple parameters, of structures of simple parameters and arrays of structures of simple parameters, as specified in the former section #Specifications_on_contextValue_format.
- the registerContext related to an entityType (ex. urn:username) does not work properly for invocation of provider due to every requests (subscriptions, queryContext request, convenience function GET /contextEntityTypes/... ) involving an entire entityType.

3.3.3.7 **New NGSI features of R2.3**

Main new NGSI features available in CAP R2.3 are:

- Support of NGSI-9 registerContext, and related invocation of "pull" providers
• Subscription with ONCHANGE condition

• Notification could be made in JSON or XML format, according to the Accept header of the related subscription request (CAP R1 could notify context in XML only).

3.3.3.8  **New NGSI features of R3.2.3**

• Support for JSON provider invocation (previous releases supported XML only). While XML remains the default language, a JSON provider should register with NGSI-9 RegisterContext, specifying HTTP header "Accept": "application/json; charset=UTF-8"
4 Publish/Subscribe Semantic Extension - User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

4.1 Introduction

This chapter explains how to use the PubSub Semantic Extension. We describe the SPARQL API and give some examples on how to use it.

For more information about SPARQL, the SPARQL API Specification document provides an introduction to the language and explains how the user can benefit from such a SPARQL API.

4.1.1 Background and Detail

This User and Programmers Guide relates to the Publish/Subscribe Semantic Extension GE which is part of the Data/Context Management chapter. Please find more information about this Generic Enabler in the following Semantic Context Open Specification.

4.2 User Guide

The PubSub Semantic Extension runs as a backend service. Thus, it doesn't provide any Graphical User Interface (GUI). It is accessed through its REST API, described in the Programmers Guide section.

4.3 Programmer Guide

4.3.1 SPARQL API usage

The PubSub semantic extension provides a REST API. It can be invoked by using the GET method with the unique and mandatory "query" parameter, which value should be a SPARQL query.

So the URL should be:

http://{SemanticExtensionIPAddress}:{SemExtPort}/queryservice/sparql?query={SPARQLQuery}

Where:

{SemanticExtensionIPAddress}

is the IP address of the SemanticExtension service. In the testbed, the global instance of this service can be used for testing purposes. Its IP address is: 130.206.82.176.

{SemExtPort}
is the portnumber where the SemanticExtension service will listen to its clients. The default portnumber is 9000.

{SPARQLQuery}

is the SPARQL query.

The following section details how to invoke the API in three alternative ways:

- using curl
- using a REST client
- a Java program

Henceforth, we will use the global instance deployed at the IP address: 130.206.82.176 on the default port number: 9000. We will use the following SPARQL query:

```
```

When this string should be URL encoded, we will use its URL format which is:

```
select+?s+?p+?o+where+%7B?s+?p+?o+%7D
```

### 4.3.1.1 Using curl

```
curl -v --proxy1.0 http://p-goodway.rd.francetelecom.fr:8080 --url http://130.206.82.176:9000/queryservice/sparql?query=select+%3Fs+%3Fp+%3Fo+where+%7B%3Fs+%3Fp+%3Fo+%7D
```

### 4.3.1.2 Using a Internet browser REST Client

The following image is the screen dump of Firefox REST client pluging.
The "GET" Method has been selected. The "URL" dialog box has been filled with the value introduced above. Once the "SEND" button has been pressed, the result can be displayed in the "Response Body (Raw)" window tab.

4.3.1.3 Using Java with a REST-WS API

This should be the most common option, for Java based application development. The Java code shown below is the simplest Java PubSub Semantic Extension client implementation.

```java
import java.net.URI;

import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;

import org.apache.commons.codec.EncoderException;
```
import org.apache.commons.codec.net.URLCodec;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;

public class SimpleJerseyClient {


    public static void main(String[] args) {

        System.getProperties().put("proxyHost", "p-goodway.rd.francetelecom.fr");
        System.getProperties().put("proxyPort", "8080");

        String baseURI = "http://130.206.82.176:9000/queryservice";

        URLCodec uc = new URLCodec();
        String sparqlQueryUri = null;

        try {
            sparqlQueryUri = uc.encode(SPARQLQUERYSTR);
            System.out.println("SimpleJerseyClient(): uri = "+ sparqlQueryUri);
        } catch (EncoderException e) {
            // TODO Auto-generated catch block
            System.err.println(e.toString());
            e.printStackTrace();
        }
    }
}
Future Internet Core Platform

```java
ClientConfig config = new DefaultClientConfig();
Client client = Client.create(config);
WebResource service =
    client.resource(UriBuilder.fromUri(baseURI).build());

// Get XML

System.out.println(service.path("sparql")
    .queryParam("query", SPARQLQUERYSTR)
    .accept(MediaType.TEXT_XML)
    .get(String.class));
```

In this code we use the Jersey JAX-RS API implementation, as suggested by the

```java
import com.sun.jersey.api.client.*;
...

"import" instructions.

The SPARQL query is stored in the SPARQLQUERYSTR static member in full text.

```java
private static String SPARQLQUERYSTR = "select ?s ?p ?o where
    {?s ?p ?o }";
```

It will later be uuencoded using an URLencoder provided by the Apache commons-codec java library:

```java
URLCodec uc = new URLCodec();
String sparqlQueryUri = null;
...

sparqlQueryUri = uc.encode(SPARQLQUERYSTR);
```

The first part of the URL, that specifies the server IP address the Semantic Extension service port number
and service name is stored in the baseURI variable.
String baseURI = "http://130.206.82.176:9000/querieservice";

The rest of the code creates a client and binds the client to the service located at the baseURI URL, completes the baseURI with the missing "sparql" path token, "query" parameter specification with its corresponding "SPARQLQUERYSTR" value. The last instruction performs this completion, at the same time as it calls the "get" method and print the result onto the standard output:

```java
ClientConfig config = new DefaultClientConfig();
Client client = Client.create(config);
WebResource service =
    client.resource(UriBuilder.fromUri(baseURI).build());
    // Get XML

System.out.println(service.path("sparql").queryParam("query", SPARQLQUERYSTR).accept(MediaType.TEXT_XML).get(String.class));
```

4.3.1.4 Results analysis

Whatever the method used to invoke the PubSub semantic extension, the result of this invocation looks like:

```xml
<sparql xmlns="http://www.w3.org/2005/sparql-results#
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:swd="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rfc1/result2.xsd">
    <head>
        <variable name="s"/>
        <variable name="p"/>
        <variable name="o"/>
    </head>
    <results distinct="false" ordered="true">
        <result>
```
As can be seen, the result is a list of bindings for the variables of the query. In our example there are three variables "?s", "?p" and "?o". In the first binding, we have

```xml
<s><uri>http://www.fi-ware.eu/PubSubOntology.owl</uri></s>
<p><uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri></p>
<o><uri>http://www.w3.org/2002/07/owl#Ontology</uri></o>
```

```xml
<s><uri>http://www.fi-ware.eu/PubSubOntology.owl#Position123456789123</uri></s>
<p><uri>http://www.fi-ware.eu/PubSubOntology.owl#hasLongitude</uri></p>
<o><literal datatype="http://www.w3.org/2001/XMLSchema#float">7.675251960754395</literal></o>
```

As can be seen, the result is a list of bindings for the variables of the query. In our example there are three variables "?s", "?p" and "?o". In the first binding, we have

```xml
s = http://www.fi-ware.eu/PubSubOntology.owl
p = http://www.w3.org/1999/02/22-rdf-syntax-ns#type
o = http://www.w3.org/2002/07/owl#Ontology
```
Which means that the graph fragment specified in the WHERE block of the query can be matched against the RDF data, when the three variables "?s", "?p" and "?o" are assigned the values given above.

4.3.2 examples
We give here some more examples of SPARQL queries and the corresponding results.

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?instance ?class
WHERE
{ ?instance rdf:type ?class }
```

Queries the classes declared in the RDF data
The reply will look like:

```xml
<sparql xmlns="http://www.w3.org/2005/sparql-results#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rf1/result2.xsd">
  <head>
    <variable name="instance"/>
    <variable name="class"/>
  </head>
  <results distinct="false" ordered="true">
    <result>
      <binding name="instance"><uri>http://www.fi-ware.eu/PubSubOntology.owl</uri></binding>
      <binding name="class"><uri>http://www.w3.org/2002/07/owl#Ontology</uri></binding>
    </result>
    <result>
      <binding name="instance"><uri>http://www.fi-ware.eu/PubSubOntology.owl#username</uri></binding>
    </result>
  </results>
</sparql>
```
<binding name="class"><uri>http://www.fi-ware.eu/PubSubOntology.owl#User</uri></binding>

</result>

<result>
  <binding name="instance"><uri>http://www.fi-ware.eu/PubSubOntology.owl#CivilAddress1</uri></binding>
  <binding name="class"><uri>http://www.fi-ware.eu/PubSubOntology.owl#CivilAddress</uri></binding>
</result>

<result>
  <binding name="instance"><uri>http://www.fi-ware.eu/PubSubOntology.owl#123456789123</uri></binding>
  <binding name="class"><uri>http://www.fi-ware.eu/PubSubOntology.owl#ImeiEntity</uri></binding>
</result>

<result>
  <binding name="instance"><uri>http://www.fi-ware.eu/PubSubOntology.owl#Position123456789123</uri></binding>
  <binding name="class"><uri>http://www.fi-ware.eu/PubSubOntology.owl#Position</uri></binding>
</result>

</results>
</sparql>

PREFIX ps: <http://www.fi-ware.eu/PubSubOntology.owl#>

SELECT ?imei
WHERE
{ ?imei rdf:type ps:ImeiEntity }

Queries the instances of the class ps:ImeiEntity

The reply will look like:
Future Internet Core Platform

```xml
<sparql xmlns="http://www.w3.org/2005/sparql-results#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rf1/result2.xsd">
  <head>
    <variable name="imei"/>
  </head>
  <results distinct="false" ordered="true">
    <result>
      <binding name="imei"><uri>http://www.fi-ware.eu/PubSubOntology.owl#123456789123</uri></binding>
    </result>
  </results>
</sparql>
```
5 CEP GE - IBM Proactive Technology Online User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

5.1 Introduction

The CEP GE is implemented by IBM Proactive Technology Online (a.k.a Proton). IBM Proactive Technology Online is a scalable integrated platform to support the development, deployment, and maintenance of event-driven applications. While standard reactive applications are based on reactions to single events, Proton reacts to situations rather than to single events. A situation is a condition that is based on a series of events that have occurred within a dynamic time window called a context. Situations include composite events (e.g., sequence), counting operators on events (e.g., aggregation) and absence operators. The Proton engine receives information on the occurrence of events from event producers, detects situations, and reports the detected situations to external consumers.

5.1.1.1 Background and Details

This User and Programmers Guide relates to the CEP GE which is part of the Data/Context Management chapter. Please find more information about this Generic Enabler in the following CEP Open Specification and CEP Architecture.

5.2 User Guide

The IBM Proactive Technology Online User Guide explains how to create a complex event processing (CEP) application, also known as an Event Processing Network. It describes the language constructs that are used to build CEP applications and includes a tutorial that explains how this can be done using the IBM Proactive Technology Online authoring tool. The User Guide can be found in the following link:

IBM Proactive Technology Online User Guide

5.3 Programmer Guide

The Programmer Guide describes the processing engine of IBM Proactive Technology Online. The programmer guide includes the high level architecture of the engine, the various supported adapters that can be used by the event producers to send events to the engine and the various supported adapters that can be used to send derived events from the engine to the consumers. It describes how a built-in adapter type can be used and how a new adapter type can be added. Configuration information and running instructions are also included. The Programmer Guide can be found in the following link:

IBM Proactive Technology Online Programmer Guide
6 BigData Analysis - User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

6.1 Introduction

This guide covers the user and development aspects of the Cosmos platform, the implementation base of the Big Data GE.

The Cosmos platform has been designed for the storage and processing of large amounts of data, in batch mode, distributing tasks in a medium-size cluster of computers, following the MapReduce paradigm [1]. The platform provides the necessary framework to allow a developer to focus on solving analytical problems without needing to know how to deploy a Big Data infrastructure, distribute jobs and data, and their synchronization.

Therefore, the user will learn about how to create his/her own computing cluster of private machines, and how the input data stored in the Infinity permanent storage cluster (based on Hadoop Distributed File System) can be analyzed with a wide variety of tools, ranging from querying systems such as Hive (simple but effective, mainly addressed for beginners) to complex workflows designed with Oozie (which combine the power of MapReduce with scripts, custom programs, even Hive tasks, in a chain of analysis tasks).

For a more detailed view of the architecture, refer to FIWARE.ArchitectureDescription.Data.BigData.

You can also refer to BigData_Analysis_Open_RESTful_API_Specification for a complete specification of the different Cosmos's APIs.

6.2 User guide

This user guide has been divided in three main sections:

• How to use the GE in order to create private computing clusters.
• How to use the Infinity permanent storage cluster in order to store and retrieve data.
• How to use a computing cluster in order to analyze the data stored in Infinity and create new output data.

In addition, this user guide shows you how to use Cygnus, the piece of software connecting Orion Context Broker and Cosmos in order to persists in an historic fashion the context data managed by Orion.

6.2.1 Using the Platform

The platform is used for cluster management purposes, i.e. creation, deletion, showing details, etc. This section has nothing to say with respect to how one of those clusters is used; see next section for that.
Cosmos CLI is the tool you need in order to manage the clusters in the Cosmos platform. This is a python script that allows you to perform all the operations you may need from the command line. A Cosmos user for the platform is needed.

### 6.2.1.1 Registering a new user

The Cosmos platform provides a web portal for users setup. It runs in the Master Node of the installation and can be used by typing the Master Node's IP address or FQDN in any browser (https access on default port 443):

![cosmos_portal.png]

A new user registration leads to the following

### 6.2.1.2 Installing Cosmos CLI

System requirements are python 2.7 with setuptools. In order to install the CLI, just type:

```
$ easy_install https://cosmos.hi.inet:8000/eggs/cosmos-py2.7.egg
```

Once it is installed, it requires minimum configuration:

```
$ cosmos configure
```

Certain information will be asked:

- Use [https://cosmos.hi.inet/cosmos/v1](https://cosmos.hi.inet/cosmos/v1) as the API base URL.
- Provide your API credentials (please, don't share with anyone).
- Type the SSH command.

This configuration file is placed at `~/.cosmosrc` in a Linux/OS X box and at `%USERPROFILE%\Application Data\cosmosrc.yaml` on Windows boxes.

### 6.2.1.3 Managing clusters

As already said, Cosmos CLI allows you to create, list and, work in general, with Cosmos clusters from the command line.

Cluster creation:

```
$ cosmos create --name "Number cruncher" --size 4
Provisioning new cluster dfbf0ab91099442f9014c3d4e2605b63
```

List clusters:

```
$ cosmos list
```
Available clusters:
  * dfbf0ab91099442f9014c3d4e2605b63

Cluster details:

```
$ cosmos show dfbf0ab91099442f9014c3d4e2605b63
{
  "href": "https://cosmos.hi.inet/cosmos/v1/cluster/dfbf0ab91099442f9014c3d4e2605b63",
  "id": "dfbf0ab91099442f9014c3d4e2605b63",
  "name": "Number cruncher",
  "size": 4,
  "state": "provisioning",
  "stateDescription": "Cluster is acquiring and configuring resources"
}
```

Accessing the cluster:

```
$ cosmos ssh dfbf0ab91099442f9014c3d4e2605b63
Warning: Permanently added 'xxx.xxx.xxx.xxx' (DSA) to the list of known hosts.
Enter passphrase for key '/user/.ssh/id_rsa':
Last login: Thu Nov 21 05:50:04 2013 from xxxxxxxxxx
[user@compute00 ~]$
```

Please notice your private key file must be named .ssh/id_rsa, otherwise it will not work.

Cluster termination:

```
$ cosmos terminate dfbf0ab91099442f9014c3d4e2605b63
Terminating cluster dfbf0ab91099442f9014c3d4e2605b63
```
6.2.1.4 À-la-carte cluster provisioning

Clusters are created with Hadoop HDFS and Mapred by default but can be customized to have any of the supported services such as Hive or Pig. The platform team is working on supporting more and more useful services so check the currently supported services by means of the list-services command.

```
$ cosmos list-services
Optional services:
  HIVE
  OOZIE
  PIG
  SQOOP
```

Then, you can add one or more optional services when running the create command.

```
$ cosmos create --name "Custom" --size 4 --services HIVE PIG
Provisioning new cluster dfbf0ab91099442f9014c3d4e2605b63
```

6.2.1.5 Persistent storage

All Cosmos clusters have an HDFS filesystem used for job execution. As the lifecycle of the in-cluster HDFS is as transient as the whole cluster, long-term results need to be retrieved and stored on a persistent storage. In the Cosmos environment such storage is called Infinity and can be managed from the command line.

Access the persistent storage:

- List files `cosmos ls <path>`
- Download files `cosmos get <remote_path> <local_path>`
- Upload files `cosmos put <local_path> <remote_path>`
- Remove files `cosmos rm [-r] <path>`

Note that the persistent storage is visible from your clusters as a non-local HDFS system. When you upload a dataset to `/data/file.txt` with `cosmos put` the information will be available at `hdfs://orion-infinity.hi.inet/user/frbattid/data/file.txt`
6.2.2 Using the Infinity cluster (persistent storage)

6.2.2.1 Data management
The Infinity cluster is exclusively used for storage purposes, permanently saving the user data, both the input and output data involved in the Hive, MapReduce or Oozie analysis.

In order to upload or download data to/from the Infinity cluster, which is based on Hadoop Distributed File System, several options are given to the user. Please observe the usage of Hadoop commands from the Infinity Head Node is not allowed, since this access is restricted to normal users.

6.2.2.1.1 Using the CLI
Storing/retrieving data in Infinity is as easy as creating a cluster: a single command line is all you need.

```
$ cosmos put /path/to/my_local_file /my_hdfs_bucket
$ cosmos get /my_hdfs_bucket /path/to/my_local_file
```

It is natively compatible with Hadoop, so you can feed data to your clusters directly from Infinity without the need of adapters or any kind of special setup.

6.2.2.1.2 Using a custom client
You can access the Infinity's HDFS by developing a custom client for local or remote execution. The complete Hadoop API is documented in the official documentation.

Additionally, the HTTP REST API supports the complete FileSystem/FileContext interface for HDFS. The operations and the corresponding FileSystem/FileContext methods are shown in the official documentation.

The programming section of this document will explain you how to create an application taking advantage both of the Hadoop API and WebHDFS.

6.2.3 Using a private cluster (temporal storage and computing)

6.2.3.1 User administration
Private clusters can also be shared among several users, e.g. users from an organization owning the cluster.

User management within a computing cluster is different from platform users management. In this case, everything inherits from Hadoop, which is user-oriented.

Within a fresh new private cluster there is only a default superuser called hdfs, which has access to the whole HDFS. This user is just for administration purposes and should never be used by common users of the cluster. In fact, common users cannot use it unless they know the hdfs password (if exists).

Regarding common users, access control and data protection among them are ensured due to users log into the master node by using their local OS user; then, they are only authorized, both by Hadoop
commands and SFTP, to access their home directory in HDFS, which is identified by their local OS user as well. Thus, users must created at three levels:

- Head Node user administration.
- HDFS user administration.

### 6.2.3.1.1 User administration at Head Node level
First of all, a user must be created in the local OS of the Head Node in order to be able to access it. This is done as usual:

```sh
$ useradd <user>
$ passwd <user>
```

Note that a new user can only be created by a sudoer user (such as `root`).

### 6.2.3.1.2 User administration at HDFS level
Once the user has been created at local level, the user may be created in HDFS. Please observe this is not mandatory since, as seen above, the user at HDFS level is created the first time a write operation is performed. Nevertheless, if you want to create it by yourself, this is done by typing the following command:

```sh
$ sudo -u hdfs hadoop fs -mkdir /user/<user>
$ sudo -u hdfs hadoop fs -chown -R <user>:hue /user/<user>
```

The above commands must be performed by the superuser of HDFS, which is `hdfs`. New users home directory is located at `/user/<user>` as you can see.

### 6.2.3.2 Data management
The private clusters have an internal HDFS for temporarily intermediate data storage. This HDFS can be managed through the Hadoop API and WebHDFS, the same than the Infinity cluster. Nevertheless, in this case the Cosmos CLI cannot be used; as a counterpart, the Hadoop commands can be used by logging into the Head Node of the computing cluster.

#### 6.2.3.2.1 Using the Hadoop commands
You can upload and download data to/from HDFS by using simple Hadoop commands. In order to do this, the data must be located in the local file system of the Head Node, which in final term limits the amount of data that can be managed, due the local file system capacity of the Head Node will be always less than the whole capacity of the entire private cluster. Thus, this limitation must be taken into account when deciding the uploading/downloading approach.

You can upload data by using the `put` or `copyFromLocal` Hadoop commands:

```
$ hadoop fs -put <local_file> <hdfs_location>
$ hadoop fs -copyFromLocal <local_file> <hdfs_location>
```

Please observe if `<local_file>` is `'-'` in the `put` command then the stdin is read:
You can download data by using the `get` or `copyToLocal` Hadoop commands:

```bash
$ hadoop fs -get <hdfs_file> <local_location>
$ hadoop fs -copyToLocal <hdfs_file> <local_location>
```

### 6.2.3.2.2 Using a custom client

You can access the private cluster's HDFS by developing a custom client for local or remote execution. The complete Hadoop API is documented in the [official documentation](#).

Additionally, the HTTP REST API supports the complete FileSystem/FileContext interface for HDFS. The operations and the corresponding FileSystem/FileContext methods are shown in the [official documentation](#).

The programming section of this document will explain you how to create an application taking advantage both of the Hadoop API and WebHDFS.

### 6.2.3.3 Doing analysis

#### 6.2.3.3.1 Querying Hive

Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc SQL-like queries, and the analysis of large datasets stored in Hadoop compatible file systems [4]. Using Hive no MapReduce programming is needed, since all the MapReduce stuff is automatically done by Hive.

The way Hive works is by loading all the data in SQL-like tables and then allowing for internal (using the Hive cli) or external (using a Java-based Hive client) SQL-like queries written in HiveQL [2]. Then, as already said, almost each time a query is performed a predefined Hive MapReduce job is run in order to select, filter, join, group, etc. the required data. We say *almost each time* because if a simple `select * from table` is performed, no MapReduce job is run (all the data within the table is returned).

**Hive CLI (local operation)**

Hive CLI [6] must be considered only for testing purposes, or remote executions through Oozie. It can be used by sshing the Head Node using your credentials and by typing `hive` in a shell:

```bash
$ hive
$ Hive history
file=/tmp/myuser/hive_job_log_opendata_201310030912_2107722657.txt
$ hive>select column1,column2,otherColumns from mytable where column1='whatever' and columns2 like '%whatever%';
```

Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
set hive.exec.reducers.bytes.per.reducer=<number>

In order to limit the maximum number of reducers:
set hive.exec.reducers.max=<number>

In order to set a constant number of reducers:
set mapred.reduce.tasks=<number>


Kill Command = /usr/lib/hadoop/bin/hadoop job -Dmapred.job.tracker=cosmosmaster-gi:8021 -kill job_201308280930_0953

2013-10-03 09:15:34,519 Stage-1 map = 0%, reduce = 0%
2013-10-03 09:15:36,545 Stage-1 map = 67%, reduce = 0%
2013-10-03 09:15:37,554 Stage-1 map = 100%, reduce = 0%
2013-10-03 09:15:44,609 Stage-1 map = 100%, reduce = 33%
2013-10-03 09:15:45,631 Stage-1 map = 100%, reduce = 100%

Ended Job = job_201308280930_0953
OK

the result set...

Custom Hive client (remote operation)

A Hive server is running in the master node. Connections from your Hive clients will be served in the TCP/1000 port as usual. Please, explore next sections in order to learn how to write your own Hive client in Java. Consider this link [3] as another useful entry point to write your own Hive clients as well.

6.2.3.3.2 Uploading and running custom MapReduce jobs

Hadoop MapReduce jobs are written in Java and packaged as jar files; the next section will explain how to create them. This section will explain how to execute these jobs by specifying the folder containing the input data files and the location where the results are expected to be available. Both folders will be placed under the /user/myuser folder; the input folder must exist before the execution, but the output folder is created once the job has finished:

$ hadoop fs -ls /user/myuser
Found 2 items

drwxr-xr-x - myuser cosmos 0 2013-06-21 09:41
/user/myuser/input
Once the Hadoop job has been uploaded, it can be run by typing the following command in a shell:

```
$ hadoop jar MyWordCount.jar es.tid.MyWordCount input output
```

```
13/06/21 09:55:12 WARN snappy.LoadSnappy: Snappy native library is available
13/06/21 09:55:12 INFO util.NativeCodeLoader: Loaded the native-hadoop library
13/06/21 09:55:12 INFO snappy.LoadSnappy: Snappy native library loaded
13/06/21 09:55:12 INFO mapred.FileInputFormat: Total input paths to process : 2
13/06/21 09:55:12 INFO mapred.JobClient: Running job: job_201306121632_0005
13/06/21 09:55:13 INFO mapred.JobClient: map 0% reduce 0%
13/06/21 09:55:32 INFO mapred.JobClient: map 66% reduce 19%
13/06/21 09:56:00 INFO mapred.JobClient: map 66% reduce 22%
13/06/21 09:56:01 INFO mapred.JobClient: map 100% reduce 22%
13/06/21 09:56:13 INFO mapred.JobClient: map 100% reduce 23%
```
13/06/21 09:56:14 INFO mapred.JobClient:  map 100% reduce 39%
13/06/21 09:56:15 INFO mapred.JobClient:  map 100% reduce 82%
13/06/21 09:56:16 INFO mapred.JobClient:  map 100% reduce 100%
13/06/21 09:56:16 INFO mapred.JobClient:  Job complete: job_201306121632_0005
13/06/21 09:56:16 INFO mapred.JobClient:  Counters: 27
13/06/21 09:56:16 INFO mapred.JobClient:  Job Counters
13/06/21 09:56:16 INFO mapred.JobClient:  Launched reduce tasks=9
13/06/21 09:56:16 INFO mapred.JobClient:  SLOTS_MILLIS_MAPS=14985
13/06/21 09:56:16 INFO mapred.JobClient:  Total time spent by all reduces waiting after reserving slots (ms)=0
13/06/21 09:56:16 INFO mapred.JobClient:  Total time spent by all maps waiting after reserving slots (ms)=0
13/06/21 09:56:16 INFO mapred.JobClient:  Launched map tasks=6
13/06/21 09:56:16 INFO mapred.JobClient:  Data-local map tasks=6
13/06/21 09:56:16 INFO mapred.JobClient:  SLOTS_MILLIS_REduces=496204
13/06/21 09:56:16 INFO mapred.JobClient:  FileSystemCounters
13/06/21 09:56:16 INFO mapred.JobClient:  FILE_BYTES_READ=126
13/06/21 09:56:16 INFO mapred.JobClient:  HDFS_BYTES_READ=442
13/06/21 09:56:16 INFO mapred.JobClient:  FILE_BYTES_WRITTEN=603288
13/06/21 09:56:16 INFO mapred.JobClient:  HDFS_BYTES_WRITTEN=17
13/06/21 09:56:16 INFO mapred.JobClient:  Map-Reduce Framework
13/06/21 09:56:16 INFO mapred.JobClient:  Map input records=7
13/06/21 09:56:16 INFO mapred.JobClient:  Reduce shuffle bytes=234
13/06/21 09:56:16 INFO mapred.JobClient:  Spilled Records=12
13/06/21 09:56:16 INFO mapred.JobClient:  Map output bytes=190
13/06/21 09:56:16 INFO mapred.JobClient:  CPU time spent (ms)=22780
13/06/21 09:56:16 INFO mapred.JobClient:  Total committed heap usage (bytes)=1641349120
13/06/21 09:56:16 INFO mapred.JobClient:  Map input bytes=114
Notice how the command is structured:

```
hadoop jar <jar_file> <main_class> <existing_input_folder>
<non_existing_output_folder>
```

Once the job has finished (a real job may take several hours or days to complete its task), the results can be found in the specified output folder:

```
$ hadoop fs -ls /user/myuser
Found 4 items
  drwx------   - myuser cosmos         0 2013-06-21 09:56
  /user/myuser/.staging
  drwxr-xr-x   - myuser cosmos         0 2013-06-21 09:41
  /user/myuser/input
  drwxr-xr-x   - myuser cosmos         0 2013-06-21 09:56
  /user/myuser/output
  drwxr-xr-x   - myuser cosmos         0 2013-06-21 08:26
  /user/myuser/tests

$ hadoop fs -ls /user/myuser/output
Found 11 items
  -rw-r--r--   3 myuser cosmos         0 2013-06-21 09:56
  /user/myuser/output/_SUCCESS
  drwxr-xr-x   - myuser cosmos         0 2013-06-21 09:55
  /user/myuser/output/_logs
  -rw-r--r--   3 myuser cosmos         0 2013-06-21 09:56
  /user/myuser/output/part-00000
```
There are nine part-0000X folder under /user/myuser/output because that is the configured number of reducer tasks is conf/mapred-site.xml configuration file; since the the hello_world file only contained two different words (hello and world), only two reducers (the ones receiving the counts of the hello and the world words) have produced a result. A merged result can be downloaded to the Head Node file system by using this command:

```bash
$ hadoop fs -getmerge /user/myuser/output/ local_file
```

Hadoop is distributed with several MapReduce examples within the hadoop-examples.jar file. The following command shows which are the available MapReduce applications:

```bash
$ hadoop jar /usr/lib/hadoop-0.20/hadoop-examples.jar

An example program must be given as the first argument.

Valid program names are:

- aggregatewordcount: An Aggregate based map/reduce program that counts the words in the input files.
- aggregatewordhist: An Aggregate based map/reduce program that computes the histogram of the words in the input files.
```
dbcount: An example job that counts the pageview counts from a database.

grep: A map/reduce program that counts the matches of a regex in the input.

join: A job that effects a join over sorted, equally partitioned datasets.

multifilewc: A job that counts words from several files.

pentomino: A map/reduce tile laying program to find solutions to pentomino problems.

pi: A map/reduce program that estimates Pi using monte-carlo method.

randomtextwriter: A map/reduce program that writes 10GB of random textual data per node.

randomwriter: A map/reduce program that writes 10GB of random data per node.

secondariesort: An example defining a secondary sort to the reduce.

sleep: A job that sleeps at each map and reduce task.

sort: A map/reduce program that sorts the data written by the random writer.

sudoku: A sudoku solver.

teragen: Generate data for the terasort.

terasort: Run the terasort.

teravalidate: Checking results of terasort.

wordcount: A map/reduce program that counts the words in the input files.

6.2.3.3 Using Oozie

Oozie is a workflow scheduler system for Apache Hadoop jobs. It allows designing Oozie Workflows, i.e. Directed Acyclical Graphs (DAGs) of actions, which in the end coordinate the execution of the jobs.

An action can be a MapReduce job, a Pig application, a file system task, or a Java application. Flow control in the DAGs is performed by node elements providing a certain logic based on the input of the preceding task in the graph (e.g. forks, joins, decision nodes), or when an event (time, whatever) triggers. Conditions have been met.

An example of DAG is the following one:
Oozie workflows definitions are written in hPDL (a XML Process Definition Language similar to JBOSS JBPM jPDL). Oozie workflows can be parameterized (using variables like ${inputDir} within the workflow definition). When submitting a workflow job values for the parameters must be provided. If properly parameterized (i.e. using different output directories) several identical workflow jobs can concurrently.

Oozie can be used in three ways, command line, Java client API and API REST.

**Oozie CLI (local operation)**

In order to use Oozie through commands, it is necessary to install the Oozie client in a remote machine. This client will be able to talk with the Oozie server already installed in the cluster (TCP/11000 port). This [official guidelines](#) will help you to install it.

Once the client has been setup, you can schedule your jobs by following this other [guidelines](#). Basically, you have to type a command like this one:

```
$ oozie job -oozie http://<master-node>:11000/oozie -config examples/apps/map-reduce/job.properties -run
```

If everything goes well, a job identifier is printed. As can be seen, you must specify the Master Node IP address or hostname, and the path to the Oozie application you want to run (in the above command, we are using the examples given with the Oozie distribution).

The oozie command can provide per job status information as well:

```
$ oozie job -oozie http://<master-node>:11000/oozie -info <job-identifier>

Job ID : 0000014-140116081225611-oozie-oozi-W
```

**Figure 1 - Example of Oozie DAG**

![Diagram of Oozie DAG](image)
Workflow Name: map-reduce-wf
App Path: hdfs://<name-node>:8020/user/<user>/examples/apps/map-reduce/workflow.xml
Status: RUNNING
Run: 0
User: <user>
Group: users
Created: 2014-01-16 16:53
Started: 2014-01-16 16:53
Last Modified: 2014-01-16 16:53
Ended: -

Actions
---------------------------------------------------------------------
ID Status Ext ID Ext Status Err Code
---------------------------------------------------------------------
0000014-140116081225611 oozie-oozimr-node RUNNING job_201401151554_0032 RUNNING -   
---------------------------------------------------------------------
---------------------------------------------------------------------

Oozie applications consists of a well-known structured directories containing:

- **lib/**, a directory containing the MR job jar, needed libraries, etc.
- **workflow.xml**, the workflow definition written in hPDL.
- **job.properties**, the values for the parameters used in the workflow definition.

The programming guide will provide guidelines about how to create your own Oozie applications.

**Custom Oozie client (remote operation)**

Oozie provides a Java API for custom Java clients development. There is a step-by-step example in this [official link](#).

Additionally, the [API REST](#) for Oozie allows you for submitting and runnint workflows in a REST fashion.

The programming section of this document will explain you how to create an application taking advantage both of the Oozie API and the Oozie REST API.
6.2.4 Using Cygnus

Cygnus persists in an HDFS-based storage cluster, such as Infinity, context data managed by Orion Context Broker.

By design, Orion only stores the last value for the entity's attributes it manages, avoiding all possibility of querying for the historical evolution of such attributes. This is solved by connecting Orion and Cosmos through Cygnus, a piece of software that subscribes to Orion for certain entities and attributes, and waits for notifications sent by Orion containing the last updated value for the subscribed context data. Once received, Cygnus automatically persists the data in the HDFS backend.

6.2.4.1 Json notification example

Let's consider the following notification in Json format coming from an Orion Context Broker instance:

```json
POST http://localhost:1028/accumulate
Content-Length: 492
User-Agent: orion/0.9.0
Host: localhost:1028
Accept: application/xml, application/json
Content-Type: application/json

{
  "subscriptionId" : "51c0ac9ed714fb3b37d7d5a8",
  "originator" : "localhost",
  "contextResponses" : [
    {
      "contextElement" : {
        "attributes" : [
          {
            "name" : "temperature",
            "type" : "centigrade",
            "value" : "26.5"
          }
        ],
        "type" : "Room",
        "attributes" : [
          {
            "name" : "humidity",
            "type" : "percent",
            "value" : "30.5"
          }
        ],
        "type" : "Sensor"
      }
    }
  ]
}
```
Such a notification is sent by Orion to the default Flume HTTP source, which relies on the developed OrionRestHandler for checking its validity (it is a POST request, the target is "notify" and the headers are OK), detecting the content type (it is in Json format), extracting the data (the Json part) and creating an event to be put in the channel:

```
    event={body={the_json_part...},headers=Template:"content-type","application/json")
```

The channel is a simple MemoryChannel behaving as a FIFO queue, and from where the OrionHDFSSink extracts the events.

The developed sink persists the data according to the original ngsi2cosmos specification, i.e. for each (entity,attribute) pair, create/append to a file named

```
<entity_name>-<entity_type>-<attribute_name>-<attribute_type>.txt
```

data lines in the form

```
<ts>|<ts_ms>|<entity_name>|<entity_type>|<attribute_name>|<attribute_type>|<value>
```

Thus, the file named "Room1-Room-temperature-centigrade.txt" (it is created if not existing) will contain a new line such as "2014-02-27 14_46_21|13453464536|Room1|Room|temperature|centigrade|26.5".

6.2.4.2  **XML notification example**

The injector also works with XML-based notifications sent to the injector (it can be seen [here](#)). The only difference is the event is created by specifying the content type is XML, and the notification parsing is done in a different way:
event={body={the_json_part...},headers=Template:"content-type","application/json"}

The key point is the behaviour remains the same than in the Json example: the same file will be created, and the same data line will be persisted within it.

6.2.4.3 **Orion subscription**

Once the connector is running, it is necessary to tell Orion Context Broker about it, in order Orion can send context data notifications to the connector. This can be done on behalf of the connector by performing the following curl command:

```bash
(curl localhost:1026/NGSI10/subscribeContext -s -S --header 'Content-Type: application/xml' -d @- | xmllint --format -) <<EOF
</?xml version="1.0"?>
<subscribeContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="false">
      <id>Room1</id>
    </entityId>
  </entityIdList>
  <attributeList>
    <attribute>temperature</attribute>
  </attributeList>
  <reference>http://host_running_cygnus:5050/notify</reference>
  <duration>P1M</duration>
  <notifyConditions>
    <notifyCondition>
      <type>ONCHANGE</type>
      <condValueList>
        <condValue>pressure</condValue>
      </condValueList>
    </notifyCondition>
  </notifyConditions>
  <throttling>PT5S</throttling>
</subscribeContextRequest>
```

EOF
6.3 Programmer guide

6.3.1 Before starting programming

If you are an expert on Hadoop and Hadoop ecosystem, you do not need to read this. Otherwise, please read carefully the following guidelines in order to experience an incremental learning curve with respect to the Big Data GE.

First of all, you must identify if the simple CLI is enough for you when dealing with data uploading and downloading from the Infinity cluster. If not, please consider creating your own HDFS client either by using WebHDFS or the Hadoop Java API.

Then, you should wonder if you need simple querying capabilities or more complex data processing. Maybe, by using HiveQL queries you are done with this GE. If your needs go far beyond Hive, or if you want to exercise with MapReduce, then is the moment to learn about how to program a custom job.

Once you are able to differentiate between Hive querying and MapReduce, perhaps you are interested in Oozie workflows, which allow you to orchestrate several MapReduce jobs and Hive tasks, achieving complex data processing.

Finally, you may want to learn how to submit and run those Oozie workflows in a remote way, either by using the Java API or the REST API.

6.3.2 Programming a HDFS client

6.3.2.1 Using the Hadoop API

This part of the Hadoop API addresses the HDFS management in a programmatically way. As you probably know, there is a CLI in charge of data uploading and downloading to/from Infinity, in addition to the Hadoop commands that can executed directly in the Head Node of your private clusters. This is only for those users needing a custom client.

The following code can be used as a basic snippet guiding you on how to write your own Hadoop clients:

```java
package a.package.of.your.choice;

import java.io.XXX;

import org.apache.hadoop.conf.Configuration;
```
import org.apache.hadoop.fs.XXX;
import org.apache.hadoop.hdfs.XXX;

public class MyHDFSClient{

    public static void main(String args[]) {
        // create a basic client
        HDFSClient client = new HDFSClient();

        // create a Configuration object pointing to all the Hadoop configuration files
        Configuration conf = new Configuration();
        conf.addResource(new Path("/home/hadoop/hadoop/conf/core-site.xml"));
        conf.addResource(new Path("/home/hadoop/hadoop/conf/hdfs-site.xml"));
        conf.addResource(new Path("/home/hadoop/hadoop/conf/mapred-site.xml"));

        // obtain a FileSystem object
        FileSystem fileSystem = FileSystem.get(conf);

        // use the FileSystem object in order to check if a file exists in HDFS
        if (!(fileSystem.exists(hdfsFile))) {
            System.out.println("No such file in HDFS: " + dstPath);
        } // if

        // use the FileSystem object in order to rename an existing file in HDFS
        fileSystem.rename(oldName, newName);
    }
}
// use the FileSystem object in order to delete an existing file in HDFS
fileSystem.delete(new Path(hdfsFile), true);

// use the FileSystem object in order to upload a new file to HDFS
fileSystem.copyFromLocalFile(srcPath, dstPath);

...
} // main

} // MyHDFSClient

You can explore all the available operations regarding the FileSystem object in the Apache Hadoop API Javadoc.

6.3.2.2 Using WebHDFS

WebHDFS is an alternative to the Hadoop API when writing custom HDFS clients. In this case, everything is built by invoking a REST API which provides full functionality regarding the HDFS.

Due the API is REST, you are not chained to any programming language when using WebHDFS, and the only thing you need is a Http Client API or library available in your favourite language. Here, a Java client snippet will be show through the use og Apache HttpComponents.

package a.package.of.your.choice;

import ...

public class MyWebHDFSClient {

    // convenience class for http clients creation given a pool
    public class HttpClientFactory {

        private static PoolingClientConnectionManager connectionsManager;
    }
}
public HttpClientFactory() {
    connectionsManager = new PoolingClientConnectionManager();
    connectionsManager.setMaxTotal(500);
    connectionsManager.setDefaultMaxPerRoute(100);
} // HttpClientFactory

public DefaultHttpClient getHttpClient() {
    return new DefaultHttpClient(connectionsManager);
} // getHttpClient

} // HttpClientFactory

public static void main(String args[]) {
    // create a http clients factory and get a http client
    HttpClientFactory factory = new HttpClientFactory();
    DefaultHttpClient client = factory.getHttpClient();

    // create a GET message in order to list a HDFS directory
    HttpGet httpGet = new HttpPost("http://head-node:50070/webhdfs/v1/path/to/my/file?op=LISTSTATUS");
    HttpResponse response = httpClient.execute(httpGet);

    // read the content of the response
    BufferedReader reader = new BufferedReader(new InputStreamReader(response.getEntity().getContent()));
    reader.readLine();
    ...

    reader.close();
}
// create a PUT message in order to create a new file in HDFS... this is 2-step operation, the first

// one step tell the Head Node about the new file


HttpRespons e response = httpClient.execute(httpPut);

// read the content of the response, within it there is the redirection location url (the real

// Datanode where the data must be copied

BufferedReader reader = new BufferedReader(new InputStreamReader(response.getEntity().getContent()));
    reader.readLine();
...
    redirectionURL = getRedirectionUTL(res);

// create a second PUT message in order to really copy the data

HttpPut httpPut = new HttpPut("http://data
node:50075/webhdfs/v1/path/to/my/file?op=CREATE?data=true");
    httpPut.setHeader("Content-Type", "application/octet-stream");
    httpPut.setEntity(new StringEntity("file content"));
    HttpResponse response = httpClient.execute(httpPut);

// read the content of the response, within it there is the redirection location url (the real

// Datanode where the data must be copied

BufferedReader reader = new BufferedReader(new InputStreamReader(response.getEntity().getContent()));
    reader.readLine();
...
    reader.close();
} // main
6.3.3 Programming a Hive client

6.3.3.1 Java
In order to create a Java-based Hive client, you will need a few lines of code and to include some libraries.

6.3.3.1.1 Needed libraries
Add the following lines to your Maven based pom.xml in order to solve the dependencies:

```xml
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>0.20.0</version>
</dependency>

<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>0.7.1</version>
</dependency>

<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-jdbc</artifactId>
    <version>0.7.1</version>
</dependency>
```

6.3.3.1.2 Minimum Java client code
```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;
```
public class HiveClient {
    // JDBC driver required for Hive connections
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";
    private static Connection con;

    private static Connection getConnection(String ip, String port, String user, String password) {
        try {
            // dynamically load the Hive JDBC driver
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            System.out.println(e.getMessage());
            return null;
        } // try catch

        try {
            // return a connection based on the Hive JDBC driver
            return DriverManager.getConnection("jdbc:hive://" + ip + ":" + port
                + "/default?user=" + user + "&password=" + password);
        } catch (SQLException e) {
            System.out.println(e.getMessage());
            return null;
        } // try catch

    } // getConnection

    private static void doQuery() {
        try {
            // from here on, everything is SQL!
            Statement stmt = con.createStatement();
        } // try
    } // doQuery
} // HiveClient
```java
ResultSet res = stmt.executeQuery("select
column1,column2,otherColumns "
    + "from mytable where column1='whatever' and columns2 
like '%whatever%'");

// iterate on the result
while (res.next()) {
    String column1 = res.getString(1);
    Integer column2 = res.getInt(2);

    // whatever you want to do with this row, here
}

// close everything
res.close();
stmt.close();
con.close();
}

// try catch
}

public static void main(String[] args) {

    // get a connection to the Hive server running on the specified 
    // IP address, listening on 10000/TCP port
    // authenticate using my credentials
    con = getConnection("130.206.80.46", "10000", "myuser", "mypasswd");

    // do a query, querying the Hive server will automatically 
    // imply the execution of one or more MapReduce jobs
    doQuery();
```
6.3.3.2 **Python**

The following code implements a basic Hive client using Python:

```python
#!/usr/bin/env python
#
# -*- coding: latin-1 -*-

import sys

from hive_service import ThriftHive
from hive_service.ttypes import HiveServerException
from thrift import Thrift
from thrift.transport import TSocket
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol

try:
    transport = TSocket.TSocket('130.206.80.46', 10000)
    transport = TTransport.TBufferedTransport(transport)
    protocol = TBinaryProtocol.TBinaryProtocol(transport)

    client = ThriftHive.Client(protocol)
    transport.open()
    client.execute("select column1,column2,otherColumns 
                    + "from mytable where column1='whatever' and columns2 like '
                    'whatever'"")

    while (1):
        row = client.fetchOne()
```
if (row == None):
    break

    # whatever you want to do with this row, here
    transport.close()
except Thrift.TException, tx:
    print '%s' % (tx.message)

Please notice that Thrift for Hive must be installed:

$ pip install hive-thrift-py

6.3.3.3  Other languages
Please, visit this URL for reference [4].

6.3.4  Programming a MapReduce job

6.3.4.1  Hadoop's MapReduce internals
6.3.4.1.1  Input Management
Hadoop, basically, works as a box receiving a very large input data and generating a relatively short output data. The key, as already known, is how to process the input data in an efficient way, even before the data arrives to the mappers and reducers. Hadoop achieves this by using a couple of processing modules called InputSplit and RecordReader:

- The InputSplits split the input file in blocks of fixed size. Blocks are specified by the path of the file and the offset in the data.
- The RecordReaders extract key-value (being the keys whatever: lines, words, characters...) pairs from the splits and send them to the Mappers. This component ensures each key-value pair is processed only once; if a key is shared between two splits, only one will handle it.

The next figure shows the whole input management:
Both InputSplits and RecordReaders work transparently, and the programmer only needs to specify a standard input format from the following ones:

- FileInputFormat
- TextInputFormat
- KeyValueTextInputFormat
- SequenceFileInputFormat
- SequenceFileAsTextInputFormat

### 6.3.4.1.2 Output Management

The Mappers produce key-value pairs as well, which are sent to the Reducers, producing new output pairs which in the end must be serialized. The output processing works in a similar way than the input processing, where a RecordWriter interfaces between the Reducer and the output file:
The programmer only needs to specify a standard output format from a set of available ones in order to deal with the RecordWriter (FileOutputFormat, etc). Other serialization formats can be used apart from Hadoop ones:

- Thrift
- Protocol Buffers
- Avro

In order to use them, set the property `io.serializations` to a Hadoop-compatible serialization framework:

```java
configuration.set("io.serializations", "org.apache.hadoop.io.serializer.JavaSerialization");
```

6.3.4.1.3  More details on the key-value pairs

The key-value pairs exchanged all along the MapReduce process have the following properties:

- Values implement the Writable interface.
- Keys implement WritableComparable.
- Some out-of-the-box Hadoop classes: IntWritable, LongWritable, FloatWritable, DoubleWritable...

6.3.4.2  Creating your own job

A MapReduce job in Hadoop/Cosmos consists of:

- A driver, a piece of software where to define inputs, outputs, formats, etc. and the entry point for launching the job.
- A set of Mappers, given by a piece of software defining its behaviour.
- A set of Reducers, given by a piece of software defining its behaviour.
6.3.4.2.1  Driver development
The following code shows the basic skeleton of a MapReduce driver.

The first decision a programmer should take is which MapReduce interface to use: org.apache.mapred
or org.apache.mapreduce; there is not much differences among them (key-value pairs are always
pushed in mapred, while you can choose if they are pushed or pulled in mapreduce), but the former is
deprecated from Hadoop 0.21.

Related to the above, the programmer must decide whether to use the JobConf or the Job class: the
second one is preferred over the first one, but essentially their purpose is the same, i.e. to create a new
MapReduce job by specifying the job name, input and output formats, which are the mapper and
reducer class and which are the input and output files.

```java
/* org.apache.mapred example */
package my.org

import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {
    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(MapClass.class);
        conf.setCombinerClass(ReduceClass.class);
        conf.setReducerClass(ReduceClass.class);
        conf.setInputFormat(TextInputFormat.class);
    }
}
```
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
JobClient.runJob(conf);
}

/* org.apache.mapreduce example */
package org.myorg;

import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
    }
}
job.setOutputFormatClass(TextOutputFormat.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.waitForCompletion(true);
}
}

6.3.4.2.2  Mappers development

/* org.apache.mapred example */
public static class MapClass extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}

/* org.apache.mapreduce example */
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
/* org.apache.mapreduce example */
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);

while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    context.write(word, one);
}

6.3.4.2.3 Reducers development

/* org.apache.mapred example */
public static class ReduceClass extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;

        while (values.hasNext()) {
            sum += values.next().get();
        }

        output.collect(key, new IntWritable(sum));
    }
}

/* org.apache.mapreduce example */
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;

        while (values.hasNext()) {
            sum += values.next().get();
        }

        context.write(key, new IntWritable(sum));
    }
}
6.3.4.2.4 Compilation
Once the above classes have been coded, just compile them and create a jar file. Supposing the source files are in a src folder, the destination for the compiled classes is classes and the destination for the jar file is dist, the following commands should be enough to create everything:

```bash
$ rm dist/*
$ rm classes/es/tid/*
$ javac -classpath /usr/lib/hadoop-0.20/hadoop-0.20.2-cdh3u6-core.jar -d classes/ src/*.java
$ jar -cvf dist/word-count.jar -C classes/ .
```

The created jar should be automatically distributed by the JobTracker to all the cluster nodes when invoked (see the Uploading and running jobs section in this wiki). You can also copy it to /usr/lib/hadoop-0.20/lib by yourself:

```bash
$ cp dist/word-count.jar /usr/lib/hadoop-0.20/lib
```

6.3.4.3 Some notes on setting the number of mappers and reducers
Picking the appropriate size for the tasks for your job can radically change the performance of Hadoop. Increasing the number of tasks increases the framework overhead, but increases load balancing and lowers the cost of failures. At one extreme is the 1 map/1 reduce case where nothing is distributed. The other extreme is to have 1,000,000 maps/1,000,000 reduces where the framework runs out of resources for the overhead.

Remember the number of mappers and reducers is configured in the conf/mapred-site.xml file.

6.3.4.3.1 Number of maps
The number of maps is usually driven by the number of DFS blocks in the input files. Although that causes people to adjust their DFS block size to adjust the number of maps. The right level of parallelism for maps seems to be around 10-100 maps/node, although we have taken it up to 300 or so for very cpu-light map tasks. Task setup takes awhile, so it is best if the maps take at least a minute to execute.

Actually controlling the number of maps is subtle. The mapred.map.tasks parameter is just a hint to the InputFormat for the number of maps. The default InputFormat behavior is to split the total number of
bytes into the right number of fragments. However, in the default case the DFS block size of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via mapred.min.split.size. Thus, if you expect 10TB of input data and have 128MB DFS blocks, you'll end up with 82k maps, unless your mapred.map.tasks is even larger. Ultimately the InputFormat determines the number of maps.

The number of map tasks can also be increased manually using the JobConf's conf.setNumMapTasks(int num). This can be used to increase the number of map tasks, but will not set the number below that which Hadoop determines via splitting the input data.

6.3.4.3.2  Number of reducers
The right number of reduces seems to be 0.95 or 1.75 * (nodes * mapred.tasktracker.tasks.maximum). At 0.95 all of the reduces can launch immediately and start transfering map outputs as the maps finish. At 1.75 the faster nodes will finish their first round of reduces and launch a second round of reduces doing a much better job of load balancing.

Currently the number of reduces is limited to roughly 1000 by the buffer size for the output files (io.buffer.size * 2 * numReduces << heapSize). This will be fixed at some point, but until it is it provides a pretty firm upper bound.

The number of reduces also controls the number of output files in the output directory, but usually that is not important because the next map/reduce step will split them into even smaller splits for the maps.

The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumReduceTasks(int num).

6.3.5  Programming an Oozie application

6.3.5.1  Application structure
An Oozie application consists of a folder containing the following:

- **lib/**, a subdirectory containing the MR job jar, other needed jar libraries, etc.
- **workflow.xml**, the workflow definition written in hPDL.
- **job.properties**, the values for the parameters used in the workflow definition.

Once the Oozie application has been created, it must be uploaded to the HDFS user space.

6.3.5.2  Workflows
Writing Oozie workflows can be a hard task, but in a few words it is necessary to generate a XML document as the one below, containing a list of flow control nodes (e.g. start, kill, end) and actions to be executed. Flow control nodes and actions may be parameterized in the **job.properties** file. This file must be called workflow.xml and must be included in the application folder.

```xml
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
</workflow-app>
```
<action name='wordcount'>
  <map-reduce>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
    <configuration>
      <property>
        <name>mapred.mapper.class</name>
        <value>org.myorg.WordCount.Map</value>
      </property>
      <property>
        <name>mapred.reducer.class</name>
        <value>org.myorg.WordCount.Reduce</value>
      </property>
      <property>
        <name>mapred.input.dir</name>
        <value>${inputDir}</value>
      </property>
      <property>
        <name>mapred.output.dir</name>
        <value>${outputDir}</value>
      </property>
    </configuration>
    <ok to='end'/>
    <error to='end'/>
  </map-reduce>
  <kill name='kill'>
    <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
  </kill>
</action>
All the detailed documentation can be found [here](https://example.com).

Once a workflow is written, it is ready to be executed with the Oozie client as described in the *Using Oozie* section. Only if you need a custom mechanism to run the workflows, the Java client API and the API REST are recommended (see next sections).

### 6.3.5.3 Java client API

Oozie provides a Java API for custom Java clients development. There is a step-by-step example in this [official link](https://example.com) which is summarized here.

**6.3.5.3.1 Needed libraries**

Add the following lines to your Maven based pom.xml in order to solve the dependencies:

```xml
<dependencies>
  ...
  <dependency>
    <groupId>com.yahoo.oozie</groupId>
    <artifactId>oozie-client</artifactId>
    <version>2.3.2-cdh3u6</version>
  </dependency>
</dependencies>
```

**6.3.5.3.2 Minimum Java client code**

```java
package com.mycompany.oozieclienttest;

import java.util.logging.Level;
```
import java.util.logging.Logger;
import org.apache.oozie.client.OozieClient;
import org.apache.oozie.client.OozieClientException;
import org.apache.oozie.client.WorkflowJob;
import java.util.Properties;

/**
 * Oozie client test.
 * 
 */
public final class OozieClientTest {
  /**
   * @param args 
   */
  private OozieClientTest() {
  } // OozieClientTest

  /**
   * @param args 
   */
  public static void main(String[] args) {
    // get a OozieClient for local Oozie
    OozieClient client = new OozieClient("http://130.206.80.46:11000/oozie/");

    // create a workflow job configuration and set the workflow application path
    Properties conf = client.createConfiguration();
    conf.setProperty(OozieClient.APP_PATH, "hdfs://cosmosmaster-gi:8020/user/frb/examples/apps/map-reduce");
// setting workflow parameters
conf.setProperty("nameNode", "hdfs://cosmosmaster-gi:8020");
conf.setProperty("jobTracker", "cosmosmaster-gi:8021");
conf.setProperty("outputDir", "output-data");
conf.setProperty("examplesRoot", "examples");
conf.setProperty("queueName", "default");

// submit and start the workflow job
String jobId = null;

try {
    jobId = client.run(conf);
} catch (OozieClientException ex) {
    Logger.getLogger(OozieClientTest.class.getName()).log(Level.SEVERE, null, ex);
}

System.out.println("Workflow job submitted");

try {
    // wait until the workflow job finishes printing the status every 10 seconds
    while (client.getJobInfo(jobId).getStatus() == WorkflowJob.Status.RUNNING) {
        System.out.println("Workflow job running ...");
        Thread.sleep(10 * 1000);
    }
} catch (OozieClientException ex) {
Logger.getLogger(OozieClientTest.class.getName()).log(Level.SEVERE, null, ex);

} catch (java.lang.InterruptedException ex) {

Logger.getLogger(OozieClientTest.class.getName()).log(Level.SEVERE, null, ex);

} // try catch catch

// print the final status of the workflow job
System.out.println("Workflow job completed ...");

try {
    System.out.println(client.getJobInfo(jobId));
} catch (OozieClientException ex) {

Logger.getLogger(OozieClientTest.class.getName()).log(Level.SEVERE, null, ex);

} // try catch

} // main

} // OozieClientTest

6.3.5.4 **API REST**

Finally, the **API REST** for Oozie allows you for submitting and running workflows in a REST fashion.

When using the API rest, the workflows must be created as usual, nevertheless the job properties must be sent in the API invocation. These are the basic steps that must be followed:

1. Put the job properties in a XML format. These XML must be sent as the payload of the job submission request.

```xml
<configuration>
    <property>
        <name>user.name</name>
        <value>frb</value>
    </property>
</configuration>
```
<property>
  <name>nameNode</name>
  <value>hdfs://cosmosmaster-gi:8020</value>
</property>

<property>
  <name>jobTracker</name>
  <value>cosmosmaster-gi:8021</value>
</property>

<property>
  <name>queueName</name>
  <value>default</value>
</property>

<property>
  <name>examplesRoot</name>
  <value>mrjobs</value>
</property>

<property>
  <name>oozie.wf.application.path</name>
  <value>hdfs://cosmosmaster-gi:8020/user/frb/mrjobs</value>
</property>

<property>
  <name>inputDir</name>
  <value>input</value>
</property>

<property>
  <name>outputDir</name>
  <value>output</value>
</property>
</configuration>
2. Submit the workflow:

```bash
$ curl -X POST "http://130.206.80.46:11000/oozie/v0/jobs" --header "Content-Type: application/xml;charset=UTF-8" -d @jobproperties
{"id":"0000092-140116081225611-oozie-oozi-W"}
```

3. Start the workflow:

```bash
```

4. Check the status of the workflow:

```bash
$ curl -X GET "http://130.206.80.46:11000/oozie/v0/job/0000092-140116081225611-oozie-oozi-W?show=info" | python -m "json.tool"
```

```json
{
    "actions": [
        ...
    ],
    "appName": "map-reduce-wf",
    "appPath": "hdfs://cosmosmaster-gi:8020/user/frb/mrjobs/workflow.xml",
    "conf": "<configuration>
        ...
    </configuration>",
    "createdTime": "Wed, 29 Jan 2014 13:14:17 GMT",
    "endTime": "Wed, 29 Jan 2014 13:34:11 GMT",
    "externalId": null,
    "group": "users",
    "id": "0000092-140116081225611-oozie-oozi-W",
    "lastModTime": "Wed, 29 Jan 2014 13:34:11 GMT",
    "run": 0,
    "startTime": "Wed, 29 Jan 2014 13:33:49 GMT",
```
"status": "SUCCEEDED",
"toString": "Workflow id[000092-140116081225611-oozie-oozi-W] status[SUCCEEDED]",
"user": "frb"
}

NOTE: the python -m json.tool is just for pretty-printing purposes.
7 Compressed Domain Video Analysis - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

7.1 Introduction

In the following, a detailed description on how to use the functionality and interfaces of the Compressed Domain Video Analysis GE as realized by Codoan is provided. First, a comprehensive overview of the functionality of Codoan is given. This is further illustrated by an example usage scenario with two separate sinks/users. The respective API calls for this example usage are further detailed, which completes the User Guide. The Programmers Guide mainly consists of an overview of the RESTful API calls and an API reference for this GE.

7.2 User Guide

Codoan provides a set of tools for analyzing video streams in the compressed domain. Its purpose is to avoid costly video content decoding prior to the actual analysis. Thereby, the tool set processes video streams by analyzing compressed or just partially decoded syntax elements. The main benefit is its very fast analysis due to a hierarchical architecture.

7.2.1 Functionality

Codoan follows the following basic design principles:

- Critical product attributes for Codoan are especially high detection rates containing only few false positives and low-complexity operation.
- Partitioning to independent functional blocks enables Codoan to support a variety of analysis methods and to get easily extended by new features. Even several operations can be combined.
- Low-complexity algorithms and implementations enable Codoan to perform very fast analyses and to be highly scalable.
- Codoan supports performing parallel analyses using different subcomponents.

The following diagram depicts the generic functional blocks of Codoan.
7.2.2 Example Usage Scenario

Employing Codoan usually requires several common steps to be performed. The following figure shows an example of a typical usage scenario (two analyzer instances (event/object recognition) attached to a media source). Note that responses and notifications are not shown for reasons of clarity and comprehensibility.
Sample usage scenario of Codoan

This scenario contains the following requests (in chronological order):

- **listInstances** (Sink 1)
  - Lists all existing analyzer instances.
  - A single instance receives and analyzes a single video stream.
  - Request
    - GET //{serverRoot}/codoan/instances HTTP/1.1
    - Accept: application/xml
  - Response
    - HTTP/1.1 200 OK
    - Content-Length: 75
    - Content-Type: application/xml
    - Server: codoan REST server
    - 
    - <?xml version="1.0" encoding="UTF-8"?>
    - <Codoan>
    - <Instances/>
    - </Codoan>

- **createInstance** (Sink 1)
  - Creates a new analyzer instance.
  - An instance with the same parameters (detectEvents, detectObjects, trackObjects, streamURI) may not already exist in order to avoid duplicate processing of the same stream with identical analytics.
  - Request
    - POST //{serverRoot}/codoan/instances HTTP/1.1
    - Accept: application/xml
    - Content-Length: 205
- configureInstance (Sink 1)
  - Configures the parameters of the detection modules.
  - Whether event detection, object detection, or object tracking are activated depends on the createInstance request.

Request

```
PUT //{serverRoot}/codoan/instances/101/config HTTP/1.1
```
Accept: application/xml
Content-Length: 721
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="false" detectObjects="true" trackObjects="true" streamURI="rtsp://192.0.2.2/stream1">
      <Configuration>
        <ObjectDetection type="Person">
          <ThresholdH264MOC>6</ThresholdH264MOC>
          <BoxFilterSize>3</BoxFilterSize>
          <TemporalFilter>
            <NumberOfPreviousFrames>3</NumberOfPreviousFrames>
            <NumberOfSubsequentFrames>3</NumberOfSubsequentFrames>
            <DistanceWeight>1.0</DistanceWeight>
            <FilterStrength>0.5</FilterStrength>
          </TemporalFilter>
        </ObjectDetection>
      </Configuration>
    </Instance>
  </Instances>
</Codoan>

Response
HTTP/1.1 200 OK
Content-Length: 1346
Content-Type: application/xml
Future Internet Core Platform

- Server: codoan REST server
- 

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="false" detectObjects="true" trackObjects="true" id="101" isRunning="false" streamURI="rtsp://192.0.2.2/stream1">
      <Configuration>
        <EventDetection type="GlobalChange">
          <NumberOfTrainingFrames>40</NumberOfTrainingFrames>
          <SlidingWindowSize>10</SlidingWindowSize>
          <ThresholdANORPFactor>1.2</ThresholdANORPFactor>
          <ThresholdARPSFactor>1.75</ThresholdARPSFactor>
          <ThresholdIFrame>5</ThresholdIFrame>
        </EventDetection>
        <ObjectDetection type="Person">
          <ThresholdH264MOC>6</ThresholdH264MOC>
          <BoxFilterSize>3</BoxFilterSize>
          <TemporalFilter>
            <NumberOfPreviousFrames>3</NumberOfPreviousFrames>
            <NumberOfSubsequentFrames>3</NumberOfSubsequentFrames>
            <DistanceWeight>1.0</DistanceWeight>
            <FilterStrength>0.5</FilterStrength>
          </TemporalFilter>
        </ObjectDetection>
        <ObjectTracking>
          <UseMovingObjectHistory>true</UseMovingObjectHistory>
        </ObjectTracking>
      </Configuration>
    </Instance>
  </Instances>
</Codoan>
```
addSink (Sink 1)

- Adds a sink (or observer) to a specific instance.
- Once a sink has been registered it gets notified in case of an event or a moving object.

**Request**

```
POST //{serverRoot}/codoan/instances/101/sinks HTTP/1.1
Accept: application/xml
Content-Length: 329
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="true" detectObjects="true" trackObjects="true" streamURI="rtsp://192.0.2.2/stream1">
      <Sinks>
        <Sink
          sinkNotificationURI="http://192.0.2.3/notification/stream1"
          sendXml="true" sendBinaryMap="true"/>
      </Sinks>
    </Instance>
  </Instances>
</Codoan>
```

**Response**
Future Internet Core Platform

- **startInstance** (Sink 1)
  - Starts an existing instance.
  - A single instance receives and analyzes a single video stream.

**Request**

- PUT //{serverRoot}/codoan/instances/101?action=start
  - HTTP/1.1
  - Accept: application/xml

**Response**

- HTTP/1.1 200 OK
• **listInstances** (Sink 2)
  - Lists all existing analyzer instances.
  - A single instance receives and analyzes a single video stream.
  - Request
    - GET //serverRoot}/codoan/instances HTTP/1.1
    - Accept: application/xml
  - Response
    - HTTP/1.1 200 OK
    - Content-Length: 247
    - Content-Type: application/xml
    - Server: codoan REST server
    -
    - <?xml version=1.0 encoding=UTF-8?>
    - <Codoan>
    - <Instances>
    - <Instance activeSinks="1" detectEvents="true" detectObjects="true" trackObjects="true" id="101" isRunning="true" streamURI="rtsp://192.0.2.2/stream1"/>
    - </Instances>
    - </Codoan>
- **addSink (Sink 2)**
  - Adds a sink (or observer) to a specific instance.
  - Once a sink has been registered it gets notified in case of an event or a moving object.
  - Request

```
POST //{serverRoot}/codoan/instances/101/sinks HTTP/1.1
Accept: application/xml
Content-Length: 329
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="true" detectObjects="true" trackObjects="true" streamURI="rtsp://192.0.2.2/stream1">
      <Sinks>
        <Sink sinkNotificationURI="http://192.0.2.5/notification/stream1" sendXml="true" sendBinaryMap="true"/>
      </Sinks>
    </Instance>
  </Instances>
</Codoan>
```

- Response

```
HTTP/1.1 201 Created
```
Future Internet Core Platform

- removeSink (Sink 1)

  - Removes a sink (or observer) from the given instance.
  - No notifications will be sent anymore when the sink has been removed.

  Request

  ```
  DELETE    //{serverRoot}/codoan/instances/101/sinks/201
  HTTP/1.1
  Accept: application/xml
  ```

  Response

  ```
  HTTP/1.1 200 OK
  Content-Length: 381
  Content-Type: application/xml
  Server: codoan REST server
  ```
- **createInstance** (Sink 2)
  - Creates a new analyzer instance.
  - An instance with the same parameters may not already exist.
  - Request

```
POST //{serverRoot}/codoan/instances HTTP/1.1
Accept: application/xml
Content-Length: 207
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="false" detectObjects="true"
      trackObjects="false" id="101" streamURI="rtsp://192.0.2.4/stream1">
      <Sinks>
        <Sink id="201" sinkNotificationURI="http://192.0.2.3/notification/stream1"
          sendXml="true" sendBinaryMap="true"/>
      </Sinks>
    </Instance>
  </Instances>
</Codoan>
```
• Response

- HTTP/1.1 201 Created
- Content-Length: 250
- Content-Type: application/xml
- Server: codoan REST server

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="false" detectObjects="true" trackObjects="false" id="102" isRunning="false" streamURI="rtsp://192.0.2.4/stream1"/>
  </Instances>
</Codoan>
```

- configureInstance (Sink 2)

  - Configures the parameters of the detection modules.
  - Whether event detection, object detection, or object tracking are activated depends on the createInstance request.

- Request

```xml
PUT //{serverRoot}/codoan/instances/102/config HTTP/1.1
Accept: application/xml
Content-Length: 722
Content-Type: application/xml
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="false" detectObjects="true" trackObjects="false" streamURI="rtsp://192.0.2.4/stream1"/>
  </Instances>
</Codoan>
```
<Configuration>
  <ObjectDetection type="Person">
    <ThresholdH264MOC>5</ThresholdH264MOC>
    <BoxFilterSize>7</BoxFilterSize>
    <TemporalFilter>
      <NumberOfPreviousFrames>3</NumberOfPreviousFrames>
      <NumberOfSubsequentFrames>3</NumberOfSubsequentFrames>
      <DistanceWeight>1.0</DistanceWeight>
      <FilterStrength>0.5</FilterStrength>
    </TemporalFilter>
  </ObjectDetection>
</Configuration>

Response

HTTP/1.1 200 OK
Content-Length: 765
Content-Type: application/xml
Server: codoan REST server

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="false" detectObjects="true" trackObjects="false" id="102" isRunning="false" streamURI="rtsp://192.0.2.4/stream1">
      <Configuration>
- **addSink (Sink 2)**
  - Adds a sink (or observer) to a specific instance.
  - Once a sink has been registered it gets notified in case of an event or a moving object.
  - Request

```xml
<POST //{serverRoot}/codoan/instances/102/sinks HTTP/1.1
Accept: application/xml
Content-Length: 331
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance detectEvents="false" detectObjects="true" trackObjects="false" streamURI="rtsp://192.0.2.4/stream1">
```

```xml
  </Instance>
</Instances>
</Codoan>
```
- `<Sinks>`
  - `<Sink>
    sinkNotificationURI="http://192.0.2.6/notification/stream1"
    sendXml="true" sendBinaryMap="true"/>
  - </Sinks>
  - </Instance>
  - </Instances>
  - </Codoan>

- Response

  - HTTP/1.1 201 Created
  - Content-Length: 383
  - Content-Type: application/xml
  - Server: codoan REST server

  - <?xml version="1.0" encoding="UTF-8"?>
  - <Codoan>
    <Instances>
      <Instance activeSinks="1" detectEvents="false"
      detectObjects="true" trackObjects="false" id="102"
      isRunning="false" streamURI="rtsp://192.0.2.4/stream1">
        <Sinks>
          <Sink id="201"
            sinkNotificationURI="http://192.0.2.6/notification/stream1"
            sendXml="true" sendBinaryMap="true"/>
          </Sinks>
        </Instance>
      </Instances>
    </Codoan>

- `startInstance` (Sink 2)
  - Starts an existing instance.
  - A single instance receives and analyzes a single video stream.
Future Internet Core Platform

o **Request**

  o PUT  
    //{serverRoot}/codoan/instances/102?action=start 
    HTTP/1.1
  o Accept: application/xml

o **Response**

  o HTTP/1.1 200 OK
  o Content-Length: 249
  o Content-Type: application/xml
  o Server: codoan REST server
  o
  o <?xml version="1.0" encoding="UTF-8"?>
  o <Codoan>
  o  <Instances>
  o   <Instance activeSinks="1" detectEvents="false" detectObjects="true" trackObjects="false" id="102" isRunning="true" streamURI="rtsp://192.0.2.4/stream1"/>
  o  </Instances>
  o </Codoan>

- **removeSink (Sink 2)**
  o Removes a sink (or observer) from the given instance.
  o No notifications will be sent anymore when the sink has been removed.
  o **Request**

  o DELETE  
    //{serverRoot}/codoan/instances/101/sinks/202 
    HTTP/1.1
  o Accept: application/xml

  o **Response**

  o HTTP/1.1 200 OK
Content-Length: 380
Content-Type: application/xml
Server: codoan REST server

<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="true" detectObjects="true" trackObjects="true" id="101" isRunning="true" streamURI="rtsp://192.0.2.2/stream1">
      <Sinks>
        <Sink id="202" sinkNotificationURI="http://192.0.2.5/notification/stream1" sendXml="true" sendBinaryMap="true"/>
      </Sinks>
    </Instance>
  </Instances>
</Codoan>

- stopInstance (Sink 2)
  - Stops an existing instance.
  - A single instance receives and analyzes a single video stream.
  - Request
    - PUT //{serverRoot}/codoan/instances/101?action=stop HTTP/1.1
    - Accept: application/xml
    - Response
      - HTTP/1.1 200 OK
      - Content-Length: 248
      - Content-Type: application/xml
      - Server: codoan REST server
- **destroyInstance** (Sink 2)
  - Destroys a previously created instance.
  - An instance cannot be destroyed if it has active sinks connected.

  **Request**
  
  ```
  DELETE //{serverRoot}/codoan/instances/101 HTTP/1.1
  Accept: application/xml
  ```

  **Response**
  
  ```
  HTTP/1.1 200 OK
  Content-Length: 248
  Content-Type: application/xml
  Server: codoan REST server
  ```
  
  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="true" detectObjects="true" trackObjects="true" id="101" isRunning="false" streamURI="rtsp://192.0.2.2/stream1"/>
  </Instances>
  </Codoan>
  ```
- **removeSink** (Sink 2)
  - Removes a sink (or observer) from the given instance.
  - No notifications will be sent anymore when the sink has been removed.
  - Request
    - DELETE //serverRoot/codoan/instances/102/sinks/201
    - HTTP/1.1
    - Accept: application/xml
  - Response
    - HTTP/1.1 200 OK
    - Content-Length: 382
    - Content-Type: application/xml
    - Server: codoan REST server
    - `<xml version="1.0" encoding="UTF-8"?>
      <Codoan>
        <Instances>
          <Instance activeSinks="0" detectEvents="false"
            detectObjects="true" trackObjects="false" id="102"
            isRunning="true" streamURI="rtsp://192.0.2.4/stream1">
            <Sinks>
              <Sink id="201"
                sinkNotificationURI="http://192.0.2.6/notification/stream1"
                sendXml="true" sendBinaryMap="true"/>
            </Sinks>
          </Instance>
        </Instances>
      </Codoan>`
  - stopInstance (Sink 2)
  - Stops an existing instance.
- A single instance receives and analyzes a single video stream.

  - Request

    - PUT //{serverRoot}/codoan/instances/102?action=stop HTTP/1.1
    - Accept: application/xml

  - Response

    - HTTP/1.1 200 OK
    - Content-Length: 250
    - Content-Type: application/xml
    - Server: codoan REST server

    ```xml
    <?xml version="1.0" encoding="UTF-8"?>
    <Codoan>
      <Instances>
        <Instance activeSinks="0" detectEvents="false" detectObjects="true" trackObjects="false" id="102" isRunning="false" streamURI="rtsp://192.0.2.4/stream1"/>
      </Instances>
    </Codoan>
    ```

- `destroyInstance` (Sink 2)

  - Destroys a previously created instance.
  - An instance can not be destroyed if it has active sinks connected.

  - Request

    - DELETE //{serverRoot}/codoan/instances/102 HTTP/1.1
    - Accept: application/xml

  - Response

    - HTTP/1.1 200 OK
7.3 Programmers Guide

7.3.1 RESTful API

For a convenient usage of Codoan a RESTful, resource-oriented API accessed via HTTP, which uses XML-based representations for information interchange, has been provided.

The following graphical diagram summarizes the available resources:

```
Codoan (server)
-------------
//{serverRoot}/codoan
   |          GET
   |-- /version
   --> getVersion
   |
   |-- /instances
   --> listInstances
   |
   --> createInstance
```

- Content-Length: 250
- Content-Type: application/xml
- Server: codoan REST server
- 

```
<?xml version="1.0" encoding="UTF-8"?>
<Codoan>
  <Instances>
    <Instance activeSinks="0" detectEvents="false" detectObjects="true" trackObjects="false" id="102" isRunning="false" streamURI="rtsp://192.0.2.4/stream1"/>
  </Instances>
</Codoan>
```
The **corresponding API operations** cover management, execution, and information purposes:
Get the version of Codoan

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//serverRoot/codoan/version</td>
<td>getVersion: get the current version of Codoan</td>
</tr>
</tbody>
</table>
List all active instances of Codoan

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td><code>{serverRoot}/codoan/instances</code></td>
<td>listInstances: lists all active instances of Codoan</td>
</tr>
</tbody>
</table>

Create a new instance of Codoan

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td><code>{serverRoot}/codoan/instances</code></td>
<td>createInstance: creates a new instance of Codoan</td>
</tr>
</tbody>
</table>

Get information about the instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td><code>{serverRoot}/codoan/instances/{instanceID}</code></td>
<td>getInstanceInfo: get info about the instance with ID <code>instanceID</code></td>
</tr>
</tbody>
</table>

Destroy an instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td><code>{serverRoot}/codoan/instances/{instanceID}</code></td>
<td>destroyInstance: destroys the instance with ID <code>instanceID</code></td>
</tr>
</tbody>
</table>

Start an instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td><code>{serverRoot}/codoan/instances/{instanceID}?action=start</code></td>
<td>startInstance: starts the instance with ID <code>instanceID</code></td>
</tr>
</tbody>
</table>

Stop an instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td><code>{serverRoot}/codoan/instances/{instanceID}?action=stop</code></td>
<td>stopInstance: stops the instance with ID <code>instanceID</code></td>
</tr>
</tbody>
</table>

Get configuration of the instance of Codoan with ID `instanceID`
<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>{serverRoot}/codoan/instances/{instanceID}/config</td>
<td>getInstanceConfig: get the config of the instance with ID instanceID</td>
</tr>
</tbody>
</table>

Configure the instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>{serverRoot}/codoan/instances/{instanceID}/config</td>
<td>configureInstance: configures the instance with ID instanceID</td>
</tr>
</tbody>
</table>

List all sinks (observers) of an instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>{serverRoot}/codoan/instances/{instanceID}/sinks</td>
<td>listSinks: lists all sinks (observers) of the instance with ID instanceID</td>
</tr>
</tbody>
</table>

Add a new sink (observer) to an instance of Codoan with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>{serverRoot}/codoan/instances/{instanceID}/sinks</td>
<td>addSink: adds a new sink (observer) to the instance with ID instanceID</td>
</tr>
</tbody>
</table>

Get information about the sink (observer) `sinkID` from the Codoan instance with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>{serverRoot}/codoan/instances/{instanceID}/sinks/{sinkID}</td>
<td>getSinkInfo: gets info about the sink (observer) with ID sinkID from instance with ID instanceID</td>
</tr>
</tbody>
</table>

Removing the sink (observer) `sinkID` from the Codoan instance with ID `instanceID`

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>{serverRoot}/codoan/instances/{instanceID}/sinks/{sinkID}</td>
<td>removeSink: removes the sink (observer) with ID sinkID from instance with ID instanceID</td>
</tr>
</tbody>
</table>
Notify the sink (observer) in case of events or detected objects

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>//{{sinkNotificationURI}}</td>
<td>notifySink: notifies the sink (observer) in case of events or detected objects</td>
</tr>
</tbody>
</table>

For a more detailed description on how to use Codoan's RESTful API, please refer also to the RESTful API documentation provided here [Open RESTful API Specification](#), that provides examples of requests/responses for each exposed service.

### 7.3.2 Application Programming Interface (API)

It is envisaged that Codoan shall be used as a service employing the RESTful API above.
8 Unstructured Data Analysis - User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

8.1 Introduction

Welcome the User and Programmer Guide for the Unstructured Data Analysis Generic Enabler. The current version of this generic enabler includes a set of server-side component and services for data resource management, unstructured data fetching, crawling, storage and analysis. UDA GE is supported by means of a big data architecture composed mainly by Apache Rome, Nutch, Flume, HBase, Hadoop and Solr.

8.1.1.1 Background and Detail

This User and Programmers Guide relates to the UDA GE. Please find more information about this Generic Enabler follow links to know more about architecture and specifications.

8.2 User guide

As the Unstructured Data Analysis is a backend component, there is no need of a User Guide.

8.3 Programmers Guide

8.3.1 Overview of the GE

The Unstructured Data Analysis is a composition of technologies in order to gather and analyse unstructured data. In a specific way, this generic enable is responsible for acquiring unstructured data from several data sources, preparing it for the analysis, and executed the linguistic and statistical analysis required by the tendencies detection). This GE is running continuously, polling the Web (through RSS feeds) for recent content, turning it into a stream of processed text documents. The documents gathered by this GE are provided in a semi-structured fashion (following the RSS structure) such as titles, publication dates, and other metadata are clearly indicated. Furthermore, the Web pages related with the RSS feeds are also gathered. Finally, the trending topics are identified in texts and article bodies using linguistic and statistical analysis. The GE provides a REST API that allows define the data sources and get the results of the analysis. The web content contains a “noise” that needs to be identified and removed before the content can be analysed. For this reason, a pipeline is executed which consists of (i) unstructured data acquisition (ii) data cleaning, (iii) data storage (iii) natural-language processing analysis, (iv) statistical analysis and (v) Results storage. This GE has been developed using the results and experiences obtained in the FIRST [FIRST] and ALERT [ALERT] projects.
8.3.2 Unstructured Data Analysis GE Architecture

The objective of the Unstructured Data Analysis GE (also known as UDA GE) is to facilitate the monitoring process of web sources in order to detect potential tendencies providing tool for gathering, preprocess, store and analyze the unstructured data contained in those sources.

In order to satisfy the previous objective, the UDA GE is composed by a set of components. Next figure presents the UDA GE Infrastructure architecture.

The Unstructured Data Analysis GE is composed by a set of modules, each one play a specific role in order to provide the functionalities of the GE.

The UDA REST API is the interface provide by the GE to interact with other pieces of software. This API provides the operation that allows creating new unstructured data analysis projects, adding sources (RSS feeds) to be analyzed. Also, this API allows obtaining the results of the analysis and the gathered documents.

Once the sources are included to a project, the GE begins the crawling process. This task is performed by the UDA Crawler, which get the RSS entries from the feeds included in the project in order to process it.
content and also to retrieve the web pages contained in the feeds. The UDA Crawler used a customized version of Apache Nutch to execute this operation. All the retrieve content is cleaned in order to extract the plain text from the RSS/HTML raw data. The data (raw and cleaned data) is stored in an Hbase No-SQL database and also in a Lucene index. The Hbase data is used by the tendencies detection analysis and also it is available for other analysis. On the other hand, the data stored in the Lucene index is used to facilitate the retrieval of the documents gathered by the GE via the REST API.

Finally, the OCELOt (Online Semantic Concept Extractor based on Linked Open Data) component analyze unstructured data in order to detect emerging tendencies, these tendencies could be used to get awareness about relevant challenges in a specific domain. OCELOt uses different natural language processing strategies (tokenization, lemmatization, Part-of-Speech Tagging) and statistical analysis to detect the tendencies. The results of the analysis are stored in a relational database, which are given to the user through the REST API.

### 8.3.3 Frontend Functionality

This GE does not provide a user interface.

### 8.3.4 Backend Functionality

Backend functionality describes functionality provided by the GE as service invocation methods for both human or computer agents. As described in Architecture section, this functionality is accessible by means of REST Web Services API, which provides the next operations:

1. **Create project**: Creates an unstructured data analysis project, a software abstraction that represent the monitoring and analysis of a set of data sources in a specific domain. To invoke the operation, a POST http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]`
2. **Get project configuration**: Obtains the configuration (name, description, analyzed sources) of a specific project. To invoke the operation, a GET http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]`
3. **Delete project**: Remove a project; this will stop the analysis of the sources contained in the project. To invoke the operation, a DELETE http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]`
4. **Get project’s data sources configuration**: Obtains all the sources analyzed in a specific project. To invoke the operation, a GET http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]/sources`
5. **Add data source to a project**: Add a new source to be analyzed in a specific project. To invoke the operation, a POST http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]/sources/[SOURCE_NAME]`
6. **Get data source configuration**: Obtain the configuration of a source analyzed in a specific project. To invoke the operation, a GET http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]/sources/[SOURCE_NAME]`
7. **Remove a data source from a project**: Remove a source from a project; this will stop the data gathering and analysis from that source. To invoke the operation, a DELETE http request should be sent to `http://<ge url location>/uda-service/uda/[PROJECT_NAME]/sources/[SOURCE_NAME]`
Future Internet Core Platform
8. Search in a project: Executes a textual search over the documents gathered in a specific project
using the Lucene query format[LuceneQuery] . To invoke the operation, a GET http request should
be sent to http://<ge url location>/ uda-service/uda/[PROJECT_NAME]/search
All methods described can be invoked by means of regular HTTP requests either using a web browser
(for those ones who rely on GET requests) or by an APIs such as Jersey.

8.3.5

Invocation of services

All methods described in the users guide above are RESTful services and therefore can be invoked by
means of regular HTTP requests or using a REST API.
As a matter of example, the following Java code shows how to get the sources of a unstructured project
using the Jersey and Gson APIs.

String project_name="TestProject";
String service_url="http://localhost:8080/uda-service/uda";
Client c = Client.create();
Gson gson = new Gson();
String resourceName = service_url+"/"+project_name+"/sources";
WebResource r = c.resource(resourceName);
String
rs=r.accept(MediaType.APPLICATION_JSON_TYPE).get(String.class);
ArrayList<LinkedTreeMap> sources =
gson.fromJson(rs,ArrayList.class);
for(LinkedTreeMap<String, String> source:sources){
System.out.println("==Source: "+source.get("name")+"
==");
System.out.println("Id: "+source.get("id"));
System.out.println("URL: "+source.get("url"));
}

The GET operations (as the previous one) can be executed by only introducing a simple URL into a web
browser. The next figure shows the sources getSouces invocation over the "AA" project using this
mechanism.

D.6.4.3: FI-WARE Users and Programmers Guide

403


The Open Specification section [Unstructured_Data_Analysis_Open_RESTful_API_Specification](http://www.bbc.co.uk/science/0/rss.xml) provides the detail for each RESTful operation giving the expected input and output for each URI.
9 Metadata Preprocessing - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

9.1 Introduction

In the following, a detailed description on how to use the functionality and interfaces of the Metadata Preprocessing GE as realized by the MetadataProcessor is provided. First, a comprehensive overview of the functionality of the Metadata Processor is given. This is further illustrated by a quite generic example use case, which should be relevant for many quite different applications areas. The respective API calls for this example usage are further detailed, which completes the User Guide. The Programmers Guide mainly consists of an overview of the RESTful API calls and an API reference for this GE.

More details on the Metadata Preprocessing GE can be found in the FIWARE.OpenSpecification.Data.MetadataPreprocessing.

9.2 User Guide

The MetadataProcessor runs as a backend service daemon, i.e., a web service. Thus, it doesn't have any Graphical User Interface (GUI). It is accessed through its RESTful API, described in the Programmers Guide section.

9.3 Programmers Guide

The Metadata Preprocessing GE, i.e., the MetadataProcessor, is meant to be used as a web service. Therefore, programming against this GE involves using its RESTful API. An overview of the API (including an example usage scenario) is given and the respective HTTP requests of the API are further detailed.

9.3.1 Functionality

The functionality of the MetadataProcessor is illustrated in the following figure.
The functionality is realized by a filter&pipe architecture. Metadata streams are received through the inbound metadata interface. This interface also performs the depacketization of the payload data. The depacketized metadata is transformed and/or filtered. This processing is the core of the MetadataProcessor. Note that transformation and filtering can be performed jointly. The processed metadata is packetized and the resulting stream is sent to one or more receivers. Note that each processing unit can only connect one source (i.e., a single input stream) but multiple sinks (i.e., multiple output streams). However, the MetadataProcessor can instantiate and manage multiple processing units.

The MetadataProcessor can be configured via an API. Input and output streams can be connected and disconnected. Furthermore, the processing engine (i.e., metadata transformation and metadata filtering) can be configured.

9.3.2 Example usage

The most important functionality of the MetadataProcessor is illustrated by an example, which can be applied quite generically to a wide range of usage scenarios with only minor changes. The following figure (i.e., UML sequence diagram) depicts the message flow of this example.
Figure. Example usage of MetadataProcessor.

The following steps are performed in the illustrated example.

1. An application (or a user of the MetadataProcessor) first creates a processing unit ('mddp_instance7') by sending a `createInstance` command to the MetadataProcessor ('metadata_preprocessing_ge').

```
POST //127.0.0.1/mdp/instances HTTP/1.1
```
If the creation of the processing unit is successful, the following response is received.

```
HTTP/1.1 201 Created
Content-Type: application/xml

<InstancesResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://schemas.datacontract.org/2004/07/mdpp.rest.Dto">
  <Instances>
    <InstanceInfo>
      <Id>0</Id>
      <MetadataUri>rtsp://127.0.0.1:1554/0</MetadataUri>
      <SourceUri>rtsp://127.0.0.1:1554/stream0</SourceUri>
    </InstanceInfo>
  </Instances>
</InstancesResponse>
```

2. The processing unit is configured by sending a `configureInstance` command. Note that a specific processing unit is referenced in the resource path of the HTTP request.

```
PUT //127.0.0.1/mdp/instances/0/config HTTP/1.1
Content-Type: application/xml
Content-Length: 544

<Configuration xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://schemas.datacontract.org/2004/07/mdpp.rest.Dto">
  <Config>
    <ProcessorCsv2Xml>
      <Enabled>true</Enabled>
      <FieldDelimiter>SEMICOLON</FieldDelimiter>
      <GroupEmptyFields>true</GroupEmptyFields>
      <NumHeaderRows>1</NumHeaderRows>
      <SkipHeaderRows>false</SkipHeaderRows>
      <TextDelimiter>SINGLE_QUOTE</TextDelimiter>
    </ProcessorCsv2Xml>
  </Config>
</Configuration>
```
With this command, an CSV-to-XML transformation is configured by specifying the relevant parameters. In addition, also the source is configured, i.e., an RSTP server is specified by its URL. If the configuration is successful, the following response is received.

```
HTTP/1.1 200 OK
Content-Type: application/xml

  <Config>
    <ProcessorCsv2Xml>
      <Enabled>true</Enabled>
      <FieldDelimiter>SEMICOLON</FieldDelimiter>
      <GroupEmptyFields>true</GroupEmptyFields>
      <NumHeaderRows>1</NumHeaderRows>
      <SkipHeaderRows>true</SkipHeaderRows>
      <TextDelimiter>SINGLE_QUOTE</TextDelimiter>
    </ProcessorCsv2Xml>
    <ProcessorXslt>
      <Enabled>false</Enabled>
      <Stylesheet />
    </ProcessorXslt>
  </Config>
</ConfigurationResponse>
```
3. In order to receive the output metadata stream, the applications send an `getMetadata` command to the MetadataProcessor.

```
GET //127.0.0.1/mdp/instances/0/metadata HTTP/1.1
Accept: application/xml
```

The application thus receives the RTSP address of the outbound metadata RTP stream, to which it can connect to:

```
HTTP/1.1 200 OK
Content-Type: application/xml

<MetadataResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns="http://schemas.datacontract.org/2004/07/mdpp.rest.Dto">
  <InstanceId>0</InstanceId>
  <MetadataUri>rtsp://127.0.0.1:1554/0</MetadataUri>
</MetadataResponse>
```

4. The application starts the processing of the processing unit of the MetadataProcessor by sending a `startInstance` command.

```
PUT //127.0.0.1/mdp/instances/0?action=start HTTP/1.1
Accept: application/xml
```

If starting the transformation process is successful, the following response is received.

```
HTTP/1.1 200 OK
Content-Type: application/xml
```
5. After metadata processing is finished, the processing unit is stopped by sending a `stopInstance` command.

```
PUT //127.0.0.1/mdp/instances/0?action=stop HTTP/1.1
Accept: application/xml
```

If stopping the transformation process is successful, the following response is received.

```
HTTP/1.1 200 OK
Content-Type: application/xml

<ActionResponse xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/mdpp.rest.Dto">
  <Action>Stop</Action>
  <Id>0</Id>
</ActionResponse>
```

6. The application destroys the processing unit (e.g., because it is not needed for later transformation purposes) by sending a `destroyInstance` command.

```
DELETE //127.0.0.1/mdp/instances/0 HTTP/1.1
Accept: application/xml
```

Note that disconnecting of the source does not have to be done separately since this is handled automatically in case a processing unit is destroyed. If the deletion of the processing unit is successful, the following response is received.

```
HTTP/1.1 200 OK
Content-Type: application/xml
```

D.6.4.3: FI-WARE Users and Programmers Guide
9.3.3 API overview

The following figure gives an overview of the RESTful API of the Metadata Preprocessing GE, i.e., the MetadataProcessor.
9.3.4 API reference

The following table gives a complete reference of the Metadata Preprocessing GE API. Concrete examples for using the RESTful API are given above and in the Metadata Preprocessing Open RESTful API Specification.

<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//{serverRoot}/{assetName}/version</td>
<td>getVersion: returns the current version of the Metadata Preprocessing GE realization/asset (e.g., MetadataProcessor)</td>
</tr>
<tr>
<td>GET</td>
<td>//{serverRoot}/{assetName}/instances</td>
<td>listInstances: lists all instances (i.e., processing units) of the Metadata Preprocessing GE</td>
</tr>
<tr>
<td>POST</td>
<td>//{serverRoot}/{assetName}/instances</td>
<td>createInstance: creates an instance (i.e., a processing unit) of the Metadata Preprocessing GE</td>
</tr>
<tr>
<td>DELETE</td>
<td>//{serverRoot}/{assetName}/instance/{instanceID}</td>
<td>destroyInstance: destroys a specific instance (i.e., processing unit)</td>
</tr>
<tr>
<td>PUT</td>
<td>//{serverRoot}/{assetName}/instance/{instanceID}?action=start</td>
<td>startInstance: starts the processing of the processing unit</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>Request URI</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td><code>{serverRoot}/{assetName}/instance/{instanceID}?action=stop</code></td>
<td><strong>stopInstance</strong>: stops the processing of the processing unit</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td><code>{serverRoot}/{assetName}/instances/{instanceID}/config</code></td>
<td><strong>getConfig</strong>: returns the configuration of an existing processing unit</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td><code>{serverRoot}/{assetName}/instances/{instanceID}/config</code></td>
<td><strong>configureInstance</strong>: configures an existing processing unit</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td><code>{serverRoot}/{assetName}/instances/{instanceID}/metadata</code></td>
<td><strong>getMetadata</strong>: returns metadata URI (i.e., RTSP URL)</td>
</tr>
</tbody>
</table>
10 LOCS - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

10.1 Introduction

This document aggregates Programming Guide and User Guide of Location GE.

LOCS implements the FI-WARE GE Open Specifications associated to the Location GE available at FIWARE.ArchitectureDescription.Data.Location

Whenever the term "Location GE" is used, you may assume that we are indeed referring to LOCS that implements the Location GE Open Specifications or an instance of LOCS.

10.2 User Guide

10.2.1 Location Platform Start-up / Shutdown

Location Platform Server start-up/shutdown is described in chapter 3/4 of of Location GE Installation and Administration Guide.

Refer to the following procedures in LOCS Installation and Administration Guide.

10.2.2 Handset fleet simulation

A Handset simulator is also provided to simulate a fleet of mobile that can move along configurable geographic paths and which positions can be transparently injected through Location GE services.

Refer the the following Handset Simulator REST API Information for details in Location GE Unit Test Plan.

10.2.3 Location Platform Agents Detailed Configuration

This chapter describes in details the configuration items for :

- MLP Agent handling Terminal Location REST API.
- SUPL Agent managing SUPL messages exchanges with terminal handset.

10.2.3.1 MLP Agent detailed configuration

Only parameters used in association with Terminal Location API are indicated. Others parameters shall be kept unchanged. Configuration parameters which could have a need to be modified in scope of Location Platform GE deployment are in green.

<table>
<thead>
<tr>
<th>Section</th>
<th>Parameter Name</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
</table>
**Operator** | **Parameter Name** | **Unit** | **Description**
---|---|---|---
**Operator** | Operator Identifier | String | Sets the name of deployed operator, as configured in dbMonitor.
**Configuration** | HTTP Rest Port | String | Http port through which are received the Terminal Location REST requests.
**Configuration** | HTTP Port | Integer | Http port through which are received the MLP requests.
**Current System Max Location Age** | Integer | Sets the location age for defining the age of a current location if not defined in location request.
**Default coordinate reference system** | String | Sets the default coordinate reference system to be used. Values allowed: EPSG_6_1_4326, EPSG_6_1_4327, EPSG_6_1_412, EPSG_6_1_2400, EPSG_6_1_27572.
**Location timeout in seconds** | Integer | Sets the time interval after which a location request is considered as non-responsive (timeout).

### 10.2.3.2 **SUPL Agent detailed configuration**

Configuration parameters which could have a need to be modified in scope of Location Platform GE deployment are in green.

<table>
<thead>
<tr>
<th>Section</th>
<th>Parameter Name</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operator</strong></td>
<td>Operator Identifier</td>
<td>String</td>
<td>Sets the name of deployed operator, as configured in dbMonitor.</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>Tcp port</td>
<td>Integer</td>
<td>SUPL TCP port, default is 14050.</td>
</tr>
<tr>
<td><strong>SUPL Parameters</strong></td>
<td>Assistance delivery Method</td>
<td>Enum</td>
<td>Assistance data message delivery: ADM/MPR.</td>
</tr>
<tr>
<td></td>
<td>Preferred Trigger Report Capability Mode</td>
<td>Enum</td>
<td>Preferred Trigger Report Capability set for the server: RealTime / QuasiRealTime / Batch.</td>
</tr>
<tr>
<td>Security parameters</td>
<td>Ver field (HMAC) Verification</td>
<td>Boolean</td>
<td>Indicates if activate or not the verification of the HMAC ver field.</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------</td>
<td>---------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Current Based Verification</td>
<td>Boolean</td>
<td>Indicates if activate or not the Current Bases Verification.</td>
</tr>
<tr>
<td><strong>SUPL Time outs</strong></td>
<td>SETINIT : SUPL_START timeout</td>
<td>Integer</td>
<td>SUPL_START maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>SETINIT: SUPL_POS_INIT timeout</td>
<td>Integer</td>
<td>SUPL_POS maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_POS_INIT timeout</td>
<td>Integer</td>
<td>SUPL_POS_INIT maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_POS timeout</td>
<td>Integer</td>
<td>SUPL_POS maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_NOTIF timeout</td>
<td>Integer</td>
<td>SUPL_NOTIF maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_INIT timeout</td>
<td>Integer</td>
<td>SUPL_INIT maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_TRIGGER_STOP timeout</td>
<td>Integer</td>
<td>SUPL_TRIGGER_STOP maximum session duration.</td>
</tr>
<tr>
<td></td>
<td>NETINIT: SUPL_TRIGGER_STOP tolerance</td>
<td>Integer</td>
<td>SUPL_TRIGGER_STOP maximum session tolerance delta time.</td>
</tr>
<tr>
<td><strong>Payload Parameters</strong></td>
<td>Horizontal accuracy</td>
<td>Integer</td>
<td>Expected horizontal accuracy in meters.</td>
</tr>
<tr>
<td></td>
<td>Response time</td>
<td>Integer</td>
<td>Response time value.</td>
</tr>
<tr>
<td></td>
<td>Trace payload data</td>
<td>Boolean</td>
<td>Indicates if trace decoded payload data in agent log.</td>
</tr>
<tr>
<td><strong>Assistance Data General Parameters</strong></td>
<td>Maximum number of assistance data delivery retries to the SET</td>
<td>Integer</td>
<td>Maximum number of assistance data delivery retries to the SET</td>
</tr>
<tr>
<td></td>
<td>SET to GPS satellites elevation cut-off angle</td>
<td>Degrees</td>
<td>Enter SET to GPS satellites elevation cut-off angle</td>
</tr>
<tr>
<td></td>
<td>Minimum Code Search Window</td>
<td>N/A</td>
<td>Select minimum code search window (chips)</td>
</tr>
<tr>
<td>Assistance data specific parameters</td>
<td>Minimum Doppler Uncertainty</td>
<td>Hz</td>
<td>Select minimum Doppler uncertainty</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------</td>
<td>----</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Maximum PRC for local DGPS corrections</td>
<td>Meters</td>
<td>Enter maximum PRC for local differential corrections.</td>
<td></td>
</tr>
<tr>
<td>Maximum PRRC for local DGPS corrections</td>
<td>Meters</td>
<td>Enter maximum PRRC for local differential corrections.</td>
<td></td>
</tr>
<tr>
<td>Assistance Data Set-Assisted only Parameters</td>
<td>Minimum Doppler Uncertainty</td>
<td>Hz</td>
<td>Select minimum Doppler uncertainty</td>
</tr>
<tr>
<td>Fix refining search window</td>
<td>m Seconds</td>
<td>Enter fix refining search window.</td>
<td></td>
</tr>
<tr>
<td>Fix refining search window</td>
<td>m Seconds</td>
<td>Enter fix refining search window.</td>
<td></td>
</tr>
<tr>
<td>Fix refining tolerance</td>
<td>m Seconds</td>
<td>Enter fix refining tolerance.</td>
<td></td>
</tr>
<tr>
<td>Fix refining min C/N0 threshold</td>
<td>DB-Hz</td>
<td>Enter fix refining min C/N0 threshold.</td>
<td></td>
</tr>
<tr>
<td>Fix refining min C/N0 threshold</td>
<td>DB-Hz</td>
<td>Enter fix refining min C/N0 threshold.</td>
<td></td>
</tr>
<tr>
<td>Horizontal Uncertainty Factor (alpha)</td>
<td>N/A</td>
<td>Enter Horizontal Uncertainty Factor * 10.</td>
<td></td>
</tr>
<tr>
<td>Vertical Uncertainty Factor (beta)</td>
<td>N/A</td>
<td>Enter Vertical Uncertainty Factor * 10.</td>
<td></td>
</tr>
<tr>
<td>Confidence Factor (gamma)</td>
<td>N/A</td>
<td>Enter Confidence Factor * 10.</td>
<td></td>
</tr>
<tr>
<td>Maximum latitude error allowed on SET computed position</td>
<td>Integer</td>
<td>Maximum latitude error allowed on SET computed position in sexagesimal seconds of arc.</td>
<td></td>
</tr>
<tr>
<td>Maximum longitude error allowed on SET computed position</td>
<td>Integer</td>
<td>Maximum longitude error allowed on SET computed position in sexagesimal seconds of arc.</td>
<td></td>
</tr>
</tbody>
</table>
10.2.4 Location Platform Database Configuration

10.2.4.1 **CELLS Database**

This database handles description of known network 2G/3G/WLAN cells. The database schema is described here below:

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1, id2, id3, id4</td>
<td>INT</td>
</tr>
<tr>
<td>idType</td>
<td>ENUM(...)</td>
</tr>
<tr>
<td>latitude</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>longitude</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>altitude</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>uncertaintySemiMajor</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>uncertaintySemiMinor</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>uncertaintyAltitude</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>majorAxisOrientation</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>antennaType</td>
<td>ENUM(...)</td>
</tr>
<tr>
<td>description</td>
<td>VARCHAR(256)</td>
</tr>
<tr>
<td>beginAngle</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>endAngle</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
```

As given in example in validation test data file `$LOCS_HOME/data/fiware_validation.sql`, the cell database can be easily populated using the MySQL stored procedure:

- `addCell()` : to create network cell descriptions

10.2.4.2 **OPE Database**

This database handles description of known end-user terminals, service provider, service and associated privacy policies for each end-user.

The database schema is described here below:
As given in example in validation test data file $LOCS_HOME/data/fiware_validation.sql, the cell database can be easily populated using the MySQL stored procedure:

- `addEndUser()`, `addHandset()`, `associateHandset()` : to describe known terminals and their main characteristics.
- `addServiceProvider()`, `addService()`, `associateService()` : to create Service Provider and associated Services.
- `addPrivacyPolicy()` : to create privacy policies for each terminal end-user/service.
10.3 Programmer guide

10.3.1 Location Services

The Network API for Terminal Location is a RESTful, resource-oriented API that allows to access to the following localization services:

- Obtain the current terminal location
- Manage client-specific subscriptions to periodic notifications
- Manage client-specific subscriptions to area (circle) notifications

A simple example of Location Query request for device telephone URI 33611223344 is presented here below:

```
GET /location/v1/queries/location?requester=test:test&address=tel:33611223344&requestedAccuracy=1000&acceptableAccuracy=1000
&maximumAge=1000&tolerance=DelayTolerant HTTP/1.1

Accept: application/xml
Host: example.com
```

The following response is received, including handset position.

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: nnnn
Date: Thu, 02 Jun 2011 02:51:59 GMT

<?xml version="1.0" encoding="UTF-8"?>
<tl:terminalLocationList
xmlns:common="urn:oma:xml:rest:netapi:common:1"
xmlns:tl="urn:oma:xml:rest:netapi:terminallocation:1">
    <tl:terminalLocation>
        <tl:address>33611223344</tl:address>
        <tl:locationRetrievalStatus>Retrieved</tl:locationRetrievalStatus>
        <tl:currentLocation>
        </tl:currentLocation>
    </tl:terminalLocation>
</tl:terminalLocationList>
```
For more complete usage of Location Platform API and description of XML structures, please refer also to the REST API documentation provided here [RESTful API documentation](#), that provides examples of requests/responses for each exposed services.

### 10.3.2 NGSI Integration with Pub/Sub Context Broker

#### 10.3.2.1 Integration schema

The Location GE NGSI integration schema can be summarized in the following sequence diagram covering:

- Initial entity registration
- Location query NGSI scenario
- Trigger Subscription NGSI scenario

The rationale is to insure complete compatibility with the Terminal Location RESTFull API described in Open Specification, by adding limited extensions and profile format for callback data. In these scenarios, notification (end-point) URL refers always to Pub/Sub Context Broker URL.
10.3.2.2 Registering Entity

Prior to use location services, an entity definition must exist/be defined in the Pub/Sub Context Broker in order to own the location position/event entity attributes. By default, Location GE NGSI binding considers the following attributes names:

- **position** attribute handle the position retrieved by invoking the Location GE location services. Position is given as decimal WGS coordinates (lat(deg), long(deg), alt(deg)).
- **locEvent** is meaningful only in case of Area Event Trigger Subscription scenario (set to None if not). Possible event types are None, Inside, Outside, Entering, Leaving.

The binding does not impose any constraint on the **Entity Id** name/format. In this chapter examples, mobile MSISDN is chosen as unique entity id.

To initialize the entity definition in the Pub/Sub Context Broker, send for example the following **RegisterContextRequest** request:

```
POST orion-psb.testbed.fi-ware.eu:1026/NGSI9/registerContext

<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="MobileTerminal" isPattern="false">
          <id>33611223344</id>
        </entityId>
      </entityIdList>
    </contextRegistration>
  </contextRegistrationList>
</registerContextRequest>
```
The Context Broker shall answer using the following RegisterContextResponse

Pour more information on Pub/Sub Context Broker services, see Orion Context Broker User and Programmers Guide

10.3.2.3 Location Query

NGSI Integration is achieved by defining two optional request properties:
- `ngsiUpdateURL`: Pub/Sub Context Broker URL where location updates are notified.
- `ngsiEntity`: ngsi entity id reference/location attribute to update. By default, `<entity id>/position` is equivalent to `<entity id>` only using default location attribute name.

For example, send the following **Location Query** request:

```plaintext
GET /location/v1/queries/location?requester=test:test&address=33611223344&requestedAccuracy=40&acceptableAccuracy=60
&maximumAge=1000&tolerance=DelayTolerant&ngsiUpdateURL=http://orion-psb.testbed.fi-ware.eu:1026&ngsiEntity=33611223344 HTTP/1.1
Accept: application/xml
Host: example.com

Expected Results:

HTTP/1.1 200 No Content

When the location session is complete, the following attribute position update notification is sent to Pub/Sub Context Broker end-point URL indicated in ngsiUpdateURL property.

```xml
PUT orion-psb.testbed.fi-ware.eu:1026/NGSI10/contextEntities/33611223344/attributes

<updateContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>position</name>
      <type/>
      <contextValue>45.8,1.38,219.5</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>locevent</name>
      <type/>
      <contextValue>None</contextValue>
    </contextAttribute>
  </contextAttributeList>
</updateContextElementRequest>
```
Expected Results:

On success, the Pub/Sub Context Broker issues the following response:

```xml
<updateContextElementResponse>
  <contextResponseList>
    <contextAttributeResponse>
      <contextAttributeList>
        <contextAttribute>
          <name>position</name>
          <type></type>
          <contextValue></contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>locevent</name>
          <type></type>
          <contextValue>None</contextValue>
        </contextAttribute>
      </contextAttributeList>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextAttributeResponse>
  </contextResponseList>
</updateContextElementResponse>
```

Update of position attribute value is checked by requesting the Pub/Sub Broker again:
GET `orion-psb.testbed.fi-ware.eu:1026/NGSI10/contextEntities/33611223344/attributes`

**Expected Results:**

On success, the Pub/Sub Context Broker issues the following response with updated position/locevent attributes:

```xml
<contextAttributeResponse>
  <contextAttributeList>
    <contextAttribute>
      <name>position</name>
      <type>wgspos</type>
      <contextValue>1.38,45.8,219.5</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>locevent</name>
      <type>areaevent</type>
      <contextValue>None</contextValue>
    </contextAttribute>
  </contextAttributeList>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
</contextAttributeResponse>
```

10.3.2.4 **Location Trigger Subscription**

NGSI Integration is achieved by defining specific content for the following properties:

- **notifyURL**: Pub/Sub Context Broker URL where location updates are notified.
- **callbackData**: ngsi entity id reference/location attribute to update. By default, `ngsi:<entity id>/position` is equivalent to `ngsi:<entity id>` only using default location attribute name.

Nota: locevent attribute name on entity is mandatory.

For example, send the following **Circle area subscription** request to add new subscription:
POST /location/v1/subscriptions/area/circle HTTP/1.1
Accept: application/xml
Host: example.com

<tl:circleNotificationSubscription
xmlns:tl="urn:oma:xml:rest:netapi:terminallocation:1"
xmlns:common="urn:oma:xml:rest:netapi:common:1">
  <tl:clientCorrelator>0003</tl:clientCorrelator>
  <tl:callbackReference>
    <common:callbackData>ngsi:33611223344</common:callbackData>
  </tl:callbackReference>
  <tl:requester>test:test</tl:requester>
  <tl:address>tel:+33611223344</tl:address>
  <tl:latitude>43.60091</tl:latitude>
  <tl:longitude>1.44299</tl:longitude>
  <tl:radius>1500</tl:radius>
  <tl:trackingAccuracy>50</tl:trackingAccuracy>
  <tl:enteringLeavingCriteria>Entering</tl:enteringLeavingCriteria>
    <tl:frequency>60</tl:frequency>
    <tl:duration>120</tl:duration>
    <tl:count>2</tl:count>
</tl:circleNotificationSubscription>

Expected Results : subscription created

HTTP/1.1 201 Created
Content-Type: application/xml
Location: http://127.0.0.1:10000/location/v1/subscriptions/area/circle/sub003
<tl:circleNotificationSubscription
xmlns:common="urn:oma:xml:rest:netapi:common:1"
xmlns:tl="urn:oma:xml:rest:netapi:terminallocation:1">
  <tl:clientCorrelator>0003</tl:clientCorrelator>
  <tl:resourceURL>/location/v1/subscriptions/area/circle/sub0003</tl:resourceURL>
  <tl:callbackReference>
    <common:callbackData>ngsi:33611223344</common:callbackData>
  </tl:callbackReference>
  <tl:requester>test:test</tl:requester>
  <tl:address>tel:33611223344</tl:address>
  <tl:latitude>43.60091</tl:latitude>
  <tl:longitude>1.44299</tl:longitude>
  <tl:radius>1500.0</tl:radius>
  <tl:trackingAccuracy>50.0</tl:trackingAccuracy>
  <tl:enteringLeavingCriteria>Entering</tl:enteringLeavingCriteria>
  <tl:checkImmediate>false</tl:checkImmediate>
  <tl:frequency>60</tl:frequency>
  <tl:duration>120</tl:duration>
  <tl:count>2</tl:count>
</tl:circleNotificationSubscription>

Expected results: scenario is configured to send two times position/locEvent update notification for entering events to Pub/Sub Context Broker end-point URL.

PUT orion-psb.testbed.fi-ware.eu:1026/NGSI10/contextEntities/33611223344/attributes
<updateContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>position</name>
      <type/>
      <contextValue>45.8,1.38,219.5</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>locevent</name>
      <type/>
      <contextValue>Entering</contextValue>
    </contextAttribute>
  </contextAttributeList>
</updateContextElementRequest>

Expected Results:

On success, the Pub/Sub Context Broker issues the following response:

<updateContextElementResponse>
  <contextResponseList>
    <contextAttributeResponse>
      <contextAttributeList>
        <contextAttribute>
          <name>position</name>
          <type/>
          <contextValue></contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>locevent</name>
          <contextValue>></contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextAttributeResponse>
  </contextResponseList>
</updateContextElementResponse>
Future Internet Core Platform

Update of position attribute value is checked by requesting the Pub/Sub Broker again:

```xml
GET orion-psb.testbed.fi-ware.eu:1026/NGSI10/contextEntities/33611223344/attributes

Expected Results:

On success, the Pub/Sub Context Broker issues the following response with updated position/locevent attributes:

```xml
<contextAttributeResponse>
  <contextAttributeList>
    <contextAttribute>
      <name>position</name>
      <type>greatpos</type>
      <contextValue>1.38,45.8,219.5</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>locevent</name>
      <type>areaevent</type>
      <contextValue>Entering</contextValue>
  </contextAttributeList>
</contextAttributeResponse>
```
</contextAttribute>
</contextAttributeList>
<statusCode>
  <code>200</code>
  <reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextAttributeResponse>

D.6.4.3: FI-WARE Users and Programmers Guide
11 Query Broker - User and Programmer Guide

You can find the content of this chapter as well in the wiki of fi-ware.

11.1 Introduction

In the following a detailed description on how to access the software modules and interfaces of the Media-enhanced QueryBroker is provided. Beside a comprehensive overview on the RESTful interface for accessing the Query Broker the necessary steps to integrate the QueryBroker into another application is explained. Additionally a code example is given, which shows an example implementation of all required steps to initialize and run the QueryBroker.

11.2 User guide

The QueryBroker provides a REST interface - a RESTful, resource-oriented API accessed via HTTP that uses XML-based representations for information interchange. It offers a convenient way to manage a QueryBroker instance and to submit complex multi-part and multimodal queries to multiple connected data repositories by sending according MPQF expressions.

The QueryBroker is implemented as a middleware to establish unified retrieval in distributed and heterogeneous environments with extension functionalities to integrate multimedia specific retrieval paradigms in the overall query execution plan, e.g., multimedia fusion technique. To ensure interoperability between the query applications and the registered database services, the QueryBroker uses as internal query representation format the MPEG Query Format (MPQF). MPQF is an XML-based (multimedia) query language which defines the format of queries and replies to be interchanged between clients and servers in a (multimedia) information search and retrieval environment.

The normative parts of the MPEG Query Format define three main components:

- The Input Query Format provides means for describing query requests from a client to a information retrieval system.
- The Output Query Format specifies a message container for the connected retrieval systems responses and finally
- the Query Management Tools provide means for functionalities such as service discovery, service aggregation and service capability description (e.g., which query types or formats are supported).

Therefore MPQF can be and is used for managing all essential tasks submitting complex multi-part and multimodal queries to multiple connected data resources, namely

- (de-)register a retrieval system/service,
creating a semantic link in case of an included join operation, and
the actual query.

As according MPQF expressions can be lengthy this is done by POST allowing to transmit the data in the body of the http request.

Resources summary:

**QueryBroker Component**

`${serverRoot}/QueryBrokerServer`

/  
/help  
/version  
/query  
/services  

/`{serviceID}`  
/`{serviceID}/{capability}`  
/`{serviceID}/CapabilityDescription`  
/`{capability}`  
/link  
/`{serviceID1}/`{linkField1}`/{serviceID2}/`{linkField2}`  
/`{serviceID1}/`{serviceID2}`

For a more detailed description on how to use the QueryBroker REST-API, please refer also to the REST API documentation provided on [Query Broker Open RESTful API Specification](#), that provides examples of requests/responses for each exposed service.

### 11.3 Programmer guide

The QueryBroker is a J2SE (Java 2 Platform, Standard Edition) application. It requires Java JRE 6 or later installed. There exist two different programming APIs for invoking the QueryBroker, namely a **JAVA-based** one provisioning according methods and a **RESTful**, resource-oriented API accessed via HTTP that
uses XML-based representations for information interchange. The REST interface is realized as a wrapper and provided as WAR file (Web application ARchive) together with the QueryBroker core.

The implementation at its core is based on the Spring Framework (e.g., enabling extensibility and inversion of control) and MAVEN (e.g., quality assurance and build management).

11.3.1.1 Architecture

The following figure shows an overview over the QueryBroker software architecture listing only the key elements, to give a short briefing how the elements are related.

The functionalities of the key elements are as follows:

- **BackendManagement** provides the functionality to register and remove service endpoints.
  - Service interface has to be implemented by any service endpoint. A service endpoint connects a database or another dataset to the multimedia query framework.

- **Broker** represents the central access point to the federated query framework. It provides the functionality to query distributed and heterogeneous multimedia databases using MPQF as query format. The main task is to receive MPQF-queries and control the following request
processing (synchronous / asynchronous mode of operation or result fetching). See Chapter Frontend Functionality for more information.

- **QueryManager** handles all received and active queries. So, new queries can be checked-in and corresponding result sets can be checked-out by the Broker.
- **RequestProcessing** controls a single query processing in a parallelized way. First an execution plan for the received query is created, followed by an optimization of the plan. Afterwards the query distribution and aggregation of the resulting sub-queries are performed. The implementations of the 4 parts are injected via the Spring framework and can be modified easily by XML configuration.
- **ExecutionPlanCreator** transforms the received MPQF query tree into an internal execution plan tree structure.
- **ExecutionPlanOptimizer** optimizes the default execution plan by replacing or switching the original tree nodes. The tree can be also transformed to a directed acyclic graph (DAG), to avoid isomorphic sub-trees in the execution plan.
- **QueryDistributor** has to analyse which sub-trees of the execution plan have to be distributed. That sub-queries can consist of one or many distributed queries to service endpoints. Each distributed query gets encapsulated in a Service Execution.
  - ServiceExecution is a wrapper for a parallel execution of a service endpoint to utilize multicore processors.
- **QueryAggregator** gets the sub-queries including the results from the service endpoints and the query execution plan. So the aggregator can combine theses two elements and process the queried results.

11.3.1.2  **RESTful Interface**

The REST API offers a convenient way to manage a QueryBroker instance and to submit complex multi-part and multimodal queries to multiple connected MMRS by sending appropriate MPQF expressions.

Running the QueryBroker requires at least registering one or more available data stores before submitting according queries.

In order to register a certain data repository a "data base connector", a.k.a. service interface has to be implemented by any service endpoint and installed at the QueryBroker. A service endpoint connects a database or another dataset to the multimedia query framework (see QueryBroker GE Installation and Administration Guide how to do that).

Assuming that according service interfaces are already implemented and installed, a service endpoint can be registered by the following POST method:

Request example:

```
POST //localhost:8080/QueryBrokerServer/services/de.uop.dimis.air.adapters.LireAdapter/QueryByMedia HTTP/1.1
Accept: /*/
```
This registers the service with the serviceID "de.uop.dimis.air.adapters.LireAdapter" at the QueryBroker endpoint. The ServiceID must be equal to the qualified name of the implementation class. The DesiredCapabilities declare which query types the service can handle. In this example the "de.uop.dimis.air.adapters.LireAdapter" can handle Query-By-Media.

Now a simple query can be submitted by calling the following POST method (The request body must contain a valid MPQF query in XML serialization.):

Request example:

```
POST //localhost:8080/QueryBrokerServer/query HTTP/1.1
Host: localhost:8080
Accept: */*
Content-Type:multipart/form-data
Content-Length: 425

<?xml version="1.0" encoding="UTF-8"?>

<mpqf:MpegQuery mpqfID="" ...>
  <mpqf:Query>
    <mpqf:Input immediateResponse="true">
      <mpqf:QueryCondition>
        <mpqf:Condition xsi:type="mpqf:QueryByMedia" matchType="similar">
          <mpqf:MediaResource resourceId="de.uop.dimis.air.adapters.LireAdapter">
            <mpqf:MediaResource>
              <mpqf:MediaUri>http://any.uri.com</mpqf:MediaUri>
            </mpqf:MediaResource>
          </mpqf:MediaResource>
        </mpqf:Condition>
      </mpqf:QueryCondition>
    </mpqf:Input>
  </mpqf:Query>
</mpqf:MpegQuery>
```
A typical response may be:

HTTP/1.1 200 OK
Content-Length: 738
Content-Type: text/plain;charset=ISO-8859-1

<?xml version="1.0" encoding="UTF-8"?>
<mpqf:MpegQuery>
  <mpqf:Query>
    <mpqf:Output>
      <mpqf:GlobalComment>This is the message from the server</mpqf:GlobalComment>
      <mpqf:ResultItem recordNumber="001" rank="1" confidence="1.0">
        <mpqf:MediaResource>http://www.mpeg7qf/db/image/19701221.jpg</mpqf:MediaResource>
      </mpqf:ResultItem>
      <mpqf:ResultItem recordNumber="002" rank="2" confidence="0.99">
      </mpqf:ResultItem>
    </mpqf:Output>
  </mpqf:Query>
</mpqf:MpegQuery>
11.3.1.3 **JAVA Interface**

In the following a detailed description on how to access the software modules and interfaces of the QueryBroker is provided. It explains the necessary steps to integrate the QueryBroker into another application and how to access its actual backend and frontend functionalities. Additionally a code example is given, which shows an example implementation of all required steps to initialize and run the QueryBroker.

As already mentioned "data base connectors" or service interfaces need to be implemented in order to be able to register and access data repositories. A description on how to realize such a service interface is given in *QueryBroker GE Installation and Administration Guide*.

11.3.1.3.1 **Backend Functionality**

Before queries can be sent to the QueryBroker, the backend management has to be set up. All backend functionalities are reachable through the BackendManagement singleton *(de.uop.dimis.air.backend.BackendManagement)*. Here, services endpoints can be (de-) registered and semantic links between them created. A service endpoint provides the functionality to connect a database or dataset to the multimedia query framework. A semantic link is meant to define the atomic connection between two heterogeneous and distributed knowledge bases on the basis of semantically equal properties. The semantic links can be set by **XPath** expressions.

### (De-)Register a Service

*Executable by BackendManagement.getInstance().registerDatabase(mqt)*;

Service endpoints are able to execute sub trees of the query execution plan. At the moment only single leaves are supported as sub trees. These can be Query-By-Media or Query-by-Description. To register a service endpoint, which has to implement the Service Interface *(de.uop.dimis.air.backend.Service)*, a valid MPQF message needs to be formulated like the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mpqf:MpegQuery mpqfID="" xmlns:mpqf="urn:mpeg:mpqf:schema:2008"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:mpeg:mpqf:schema:2008
mpqf_semantic_enhancement.xsd">
  <mpqf:Management>
    <mpqf:Input>
      <mpqf:DesiredCapability>
        <!-- Example Implementation goes here -->
      </mpqf:DesiredCapability>
    </mpqf:Input>
  </mpqf:Management>
</mpqf:MpegQuery>
```
This contains the ServiceID, which is equal to the qualified name of the implementation class. The DesiredCapabilities declare which query types the service can handle. In this example the ExampleService can handle Query-By-Media. See the MPQF-Standard Annex B.2 for more Query URNs.

In order to deregister a service endpoint a MPQF-Register-Message must be sent with an empty list of desired capabilities.

Java example:

```java
// get the register query xml file as stream
InputStream stream = ...

// unmarshal the xml file
Unmarshaller u = NamespaceHelper.getInstance().getJAXBContextMpqfJPSearch().createUnmarshaller();
MpegQueryType mpqfQuery = (MpegQueryType) u.unmarshal(stream);

// register the database
BackendManagement.getInstance().registerDatabase(mpqfQuery);
```

Creating a Semantic Link

Executable by BackendManagement.getInstance().registerSemanticLink(sl);

To be able to merge results from different services it is necessary to know which fields can be used for identification (cp. Primary key in relational database systems). For every pair of services a semantic link can be defined. If such a link is undefined, a default semantic link will be created at runtime. The default semantic link uses the identifier field of the JPSearch Core Meta Schema for every Service.

KeyMatchesType-Messages are used for the registration of a semantic link:
The `KeyMatchesType` contain the IDs of source and target/referenced database (service endpoint) and the fields that should be used to identify results from both services as equal. A `KeyMatchesType` can contain multiple referenced databases. When you register a new semantic link between two Services, three semantic links will be generated. In addition to the registered link, the reflexive links will also be created by using the identifier for this database. If this particular reflexive semantic link already exists, it will be updated with the current field. Note that semantic links are symmetric (undirected edges between services). One has to be aware that semantic links are not transitive.

**Java example:**

```java
// get the semantic link xml file as stream
InputStream stream = ...;
// unmarshal the xml file
Unmarshaller u = NamespaceHelper.getInstance().getJAXBContextsSemanticLinks().createUnmarshaller();
KeyMatchesType link = (KeyMatchesType) ((JAXBElement<?>) u.unmarshal(stream)).getValue();
// register the semantic link
BackendManagement.getInstance().registerSemanticLink(link);
```
11.3.1.3.2  Frontend Functionalities

After at least one service endpoint is registered and the backend configuration is done, the QueryBroker is available for multimedia queries. The frontend functionalities are reachable through the Broker singleton (de.uop.dimis.air.Broker). Here you can start synchronous/asynchronous queries or fetch the query results for a specified asynchronous query.

Querying

The QueryBroker uses the MPEQ Query Format (MPQF) to describe queries. The XML-based query format is implemented by use of the Java Architecture for XML Binding (JAXB). The transformed binding java code is located in the package de.uop.dimis.air.internalObjects.mpqf. It is possible to describe a query with an xml file or specify the conditions directly in Java. Since the MPQF-Standard has much complex functionality, not all query operators are currently implemented in the QueryBroker. The section Code Example shows how the operators are used properly. Implemented operators:

- Projection
- Limit
- Distinct
- GroupBy (with aggregation) over multiple attributes
- Or (half blocking, merging, using hashmaps for improved runtime)
- And (half blocking, merging, using hashmaps for improved runtime)
- SortBy over a single attribute

Synchronous Query

A synchronous query can be sent by setting the isImmediateResponse-field of the MPQF-Query to true. The QueryBroker blocks the query until the query process is finished and the client gets the results immediately. A possible minimal synchronous query can look like the following XML-file. Here, a single Query-By-Media (similar search for an image with the url 'http://any.uri.com') is sent to the QueryBroker:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mpqf:MpegQuery mpqfID="" ...>
  <mpqf:Query>
    <mpqf:Input immediateResponse="true">
      <mpqf:QueryCondition>
        <mpqf:Condition xsi:type="mpqf:QueryByMedia" matchType="similar">
          <mpqf:MediaResource resourceID="res01">
          </mpqf:MediaResource>
        </mpqf:Condition>
      </mpqf:QueryCondition>
    </mpqf:Input>
  </mpqf:Query>
</mpqf:MpegQuery>
```
The following Java-Code example shows how the specified MPQF-Query can be forwarded to the QueryBroker and how the results can be retrieved from the response object:

**Java example:**

```java
// get the query xml file as stream
InputStream stream = ...
// unmarshal the xml file
Unmarshaller u = NamespaceHelper.getInstance().getJAXBContextMpqfJPSearch().createUnmarshaller();
MpegQueryType mpqfQuery = (MpegQueryType) u.unmarshal(stream);
// query the databases
MpeqQueryType response = Broker.getInstance().query(mpqfQuery);
// get the results from the response object
List<ResultItemType> results = response.getQuery().getOutput().getResultItem();
```

Alternatively the same query can be described and forwarded to the QueryBroker via pure Java Code:

**Java example:**

```java
// create mpqf objects with the mpqf object factory
ObjectFactory factory = new ObjectFactory();
MpegQueryType mpqfQuery = factory.createMpegQueryType();
Query query = factory.createMpegQueryTypeQuery();
mpqfQuery.setQuery(query);
```
InputQueryType input = factory.createInputQueryType();
// activate the synchronous query
input.setImmediateResponse(true);
query.setInput(input);
QueryConditionType conditions = factory.createQueryConditionType();
input.setQueryCondition(conditions);
QueryByMedia qbm = factory.createQueryByMedia();
conditions.setCondition(qbm);
MediaResourceType resource = factory.createMediaResourceType();
qbm.setMediaResource(resource);
MediaLocatorType locator = factory.createMediaLocatorType();
resource.setMediaResource(locator);
locator.setMediaUri("http://any.uri.com");
// query the databases
MpegQueryType response = Broker.getInstance().query(mpqfQuery);
// get the results from the response object
List<ResultItemType> results =
    response.getQuery().getOutput().getResultItem();

Asynchronous Query
To start an asynchronous query the isImmediateResponse-field of the MPQF-Query has to be set to false. The QueryBroker sends a response with a unique MPQF query id. So, the results for the query can be fetched afterwards by referring to the retrieved id. The following code example demonstrates an asynchronous result retrieval in detail.

Java example:

// create a MPQF-Query with XML or pure Java with isImmediateResponse = false
MpegQueryType mpqfQuery = ...
// query the databases
MpegQueryType response = Broker.getInstance().query(mpqfQuery);
// get the mpqf id for result fetching
String mpqfID = response.getMpqfID();
// ... wait ...
// create the fetch query
ObjectFactory factory = new ObjectFactory();
MpegQueryType fetchMpqf = factory.createMpegQueryType();
Query query = factory.createMpegQueryTypeQuery();
fetchMpqf.setQuery(query);  
FetchResult fetch = factory.createMpegQueryTypeQueryFetchResult();
// refer to the retrieved MPQF-ID
fetch.setQueryID(mpqfID);
query.setFetchResult(fetch);
// fetch the results from the query broker
MpegQueryType response2 = Broker.getInstance().query(fetchMpqf);
// get the results from the response object
List<ResultItemType> results =
response2.getQuery().getOutput().getResultItem();

If the broker hasn’t finished the retrieval already or an error occurs during the processing the system message in the MPQF-Response has a corresponding status message (e.g. “101 – Server resource busy”). These messages can be retrieved via java as follows:

Java example:

```java
response.getQuery().getOutput().getSystemMessage().getStatus();
```

See the MPQF-Standard for more information about system messages and error codes.

**Complex Query Example**

The following XML code shows a more complex query example. The result count is limited to 10 items (maxItemCount), the results are sorted ascending by the 'identifier'-field and a projection on the field 'description' (ReqField) takes place. The query condition consists of a join of a QueryByMedia and a QueryByDescription, which contains metadata constraints described by the MPEG-7 metadata schema.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mpqf:MpegQuery mpqfID="101" xmlns:mpqf="urn:mpeg:mpqf:schema:2008"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:mpeg:mpqf:schema:2008 mpqf_semantic_enhancement.xsd">
 <mpqf:Query>
   <mpqf:Input>

```
<mpqf:OutputDescription maxItemCount="10" distinct="true">
  <mpqf:ReqField typeName="description"></mpqf:ReqField>
  <mpqf:SortBy xsi:type="mpqf:SortByFieldType" order="ascending">
    <mpqf:Field>identifier</mpqf:Field>
  </mpqf:SortBy>
</mpqf:OutputDescription>
<mpqf:QueryCondition>
  <mpqf:Condition xsi:type="mpqf:AND">
    <mpqf:Condition xsi:type="mpqf:QueryByMedia">
      <mpqf:MediaResource resourceID="ID_5001">
        <mpqf:MediaUri>http://tolle.uri/1</mpqf:MediaUri>
      </mpqf:MediaResource>
    </mpqf:Condition>
    <mpqf:Condition xsi:type="mpqf:QueryByDescription">
      <mpqf:DescriptionResource resourceID="desc001">
        <mpqf:AnyDescription xmlns:mpeg7="urn:mpeg:mpeg7:schema:2004"
          xsi:schemaLocation="urn:mpeg:mpeg7:schema:2004 M7v2schema.xsd">
          <mpeg7:Mpeg7>
            <mpeg7:DescriptionUnit xsi:type="mpeg7:CreationInformationType">
              <mpeg7:Creation>
                <mpeg7:Title>Example Title</mpeg7:Title>
              </mpeg7:Creation>
            </mpeg7:DescriptionUnit>
          </mpeg7:Mpeg7>
        </mpqf:AnyDescription>
      </mpqf:DescriptionResource>
    </mpqf:Condition>
  </mpqf:Condition>
</mpqf:QueryCondition>
Query Execution Tree Evaluation
The query aggregator evaluates the query execution plan (QEP). The result of this evaluation is a number of results that will later be returned to the querying client. Every leaf of the QEP has a reference to a sub-query, which includes one or many service executions, so the results from the sub-query are accessible. For the evaluation an iterator driven model is used. Every node has a next and hasNext method to get the next result item from its children. Since the QEP can be a DAG, hasNext and next need a parameter to decide on which path in the DAG they are called, to be able to choose the correct iterator. hasNext checks if there is a next result item in the local result list. If there is one, next can call this item. If not, hasNext tries to query its child nodes for a next valid result item. If this is possible the new item will be appended in the local result list. Only if no successor can be computed hasNext will return false. So make always sure to call hasNext before next.

There are blocking, half blocking and none blocking operators. A blocking operator needs all results from its children to decide which result will returned next. The SortBy operator is a blocking operator. An operator is half blocking, if it doesn’t need all results from every child. The AND operator is implemented in such a way. None blocking operators like LIMIT can forward results without knowing every other possible result. Some operators have to merge results. If two results are equal (according to the specific semantic link) they must be merged. Merging operators are for example AND and OR. Merging two results means that one result is augmented with additional information from the second result. No information is overwritten.

11.3.1.3.3 Code Example
The following code examples describe a full initial setup of the QueryBroker. An implementation of a QueryByDescription-Service is presented (ExampleService.java), followed by the registration of this service and the registration of the semantic link to a fictional second service endpoint. The semantic link registration can be omitted if the default semantic link (‘identifier’) is demanded.

Java example:

```java
// ExampleService.java

public class ExampleService implements Service {

    @Override
```
public MpegType execute(MpegQueryType distributedQuery) {
    // get the query conditions
    BooleanExpressionType conditions = distributedQuery.getQuery().getInput().getQueryCondition();

    // ... do your program logic with the query conditions ...

    // create a result container for the computed results
    ObjectFactory mpqfObjFac = new ObjectFactory();
    MpegQueryType result = mpqfObjFac.createMpegQueryType();
    Query qry = mpqfObjFac.createMpegQueryTypeQuery();
    result.setQuery(qry);
    OutputQueryType oqt = mpqfObjFac.createOutputQueryType();
    qry.setOutput(oqt);
    List<ResultItemType> resultItems = oqt.getResultItem();

    // for each result of the service endpoint create a result item and add it
    // to the results list
    de.uop.dimis.air.internalObjects.jpsearch.ObjectFactory
    jpsearchObjFac =
        new
    de.uop.dimis.air.internalObjects.jpsearch.ObjectFactory();
    for(...) {
        ResultItemType resultItem = mpqfObjFac.createResultItemType();
        resultItems.add(resultItem);
        Description description = mpqfObjFac.createResultItemTypeDescription();
        resultItem.getDescription().add(description);
        JPSearchCoreType coreType =
            jpsearchObjFac.createJPSearchCoreType();
description.getContent().add(coreType);

    // set the result properties in the jpsearch coreType
    object. (e.g.
    // identifier)
    coreType.setIdentifier("...");

    // set the origin to identify from which service endpoint
    the
    // result item was generated
    resultItem.setOriginID("MedicoExecuteDICOM");

    }

    return result;

}

// RegisterExampleService.xml
<?xml version="1.0" encoding="UTF-8"?>
<mpqf:MpegQuery mpqfID="" xmlns:mpqf="urn:mpeg:mpqf:schema:2008"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:mpeg:mpqf:schema:2008
mpqf_semantic_enhancement.xsd">
    <mpqf:Management>
      <mpqf:Input>
        <mpqf:DesiredCapability>
          <!-- Query By Description: 100.3.6.2 -->
          <mpqf:SupportedQueryTypes
            href="urn:mpeg:mpqf:2008:CS:full:100.3.6.2" />
        </mpqf:DesiredCapability>
        <mpqf:ServiceID>
          de.uop.dimis.air.ExampleService
        </mpqf:ServiceID>
      </mpqf:Input>
    </mpqf:Management>
We assume that a second service with the identifier ‘de.uop.dimis.air.SecondService’ gets registered, too. The two databases have the semantic link between the fields ‘identifier’ (ExampleService) and ‘title’ (SecondService).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<key:KeyMatches xmlns:key="urn:keyMatches:schema:2011"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
schemaLocation="urn:keyMatches:schema:2011 keys.xsd">
  <key:DB id="de.uop.dimis.air.ExampleService">
    <key:Key>
      <key:Field>identifier</key:Field>
      <key:ReferencedDB>
        de.uop.dimis.air.SecondService
      </key:ReferencedDB>
    <key:ReferencedDBField>title</key:ReferencedDBField>
    </key:Key>
  </key:DB>
</key:KeyMatches>
```

Now the xml files are loaded and transferred to the QueryBroker. This has to be done only once for initialization. After these steps the QueryBroker is available for query requests.

**Java example:**

```java
// Initialize the QueryBroker and register the service (e.g. in a main-method)

// get the register query xml file as stream
InputStream stream =
ExampleService.class.getResourceAsStream("RegisterExampleService.xml")
;
// unmarshal the xml file
Unmarshaller u =
```
NamespaceHelper.getInstance().getJAXBContextMpqfJPSearch().createUnmarshaller();

MpegQueryType mpqfQuery = (MpegQueryType) u.unmarshal(stream);

// register the database
BackendManagement.getInstance().registerDatabase(mpqfQuery);

// get the semantic link xml file as stream
InputStream stream =

ExampleService.class.getResourceAsStream("SemanticLinks.xml");

// unmarshal the xml file
Unmarshaller u =
NamespaceHelper.getInstance().getJAXBContextsSemanticLinks().createUnmarshaller();

KeyMatchesType link = (KeyMatchesType) ((JAXBElement<?>) u.unmarshal(stream)).getValue();

// register the semantic link
BackendManagement.getInstance().registerSemanticLink(link);

// the queryBroker is now ready for queries

11.4 References

12 Semantic Application Support - Users and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

12.1 Introduction

Welcome the User and Programmer Guide for the Semantic Application Support Generic Enabler. The current version of this generic enabler has two main parts: An open source isolated GUI client for ontology engineering management based on Web Protege and server-side component and services for ontology storage, versioning and management.

Please find more information about this Generic Enabler in the following SAS Open Specification and SAS Architecture.

12.1.1 Overview of the GE

The Semantic Web Application Support GE aims at providing an effective environment for developers to implement and deploy high quality Semantic Web-based applications. The Semantic Web was first envisioned more than a decade ago by Tim Berners-Lee, as a way of turning the Web into a set of resources understandable not only for humans, but also by machines (software agents or programs). Following this idea, Internet will evolve into a machine understandable Web, therefore increasing its exploitation capabilities. The Semantic Web has focused the efforts of many researchers, institutions and IT practitioners, and received a fair amount of investment from European and other governmental bodies. As a result of these efforts, a large amount of mark-up languages, techniques and applications, ranging from semantic search engines to query answering system, have been developed. Nevertheless the adoption of Semantic Web from the IT industry is still following a slow and painful process.

In recent years, several discussions had taken place to find out the reasons preventing Semantic Web paradigm adoption. There is a general agreement that those reasons range from technical (lack of infrastructure to meet industry requirements in terms of scalability, performance, distribution, security, etc.) to engineering (not general uptake of methodologies, lack of best practices and supporting tools), and finally commercial aspects (difficulties to penetrate in the market, lack of understanding of the main strengths and weaknesses of the semantic technologies by company managers, no good sales strategies, etc.).

The Semantic Application Support enabler addresses part of the abovementioned problems (engineering and technical) from a data management point of view, by providing:

- An infrastructure for metadata publishing, retrieving and subscription that meets industry requirements like scalability, distribution and security. From now and so on, we will refer to this infrastructure as SWAS Architecture.
- A set of tools for infrastructure and data management, supporting most adopted methodologies and best practices. From now and so on, we will refer to this tools as SWAS Engineering Environment.
12.1.2 Semantic Application Support GE Architecture

Semantic Application Support GE can be split in a client-side Engineering Environment and a server-side Infrastructure. Figure SWAS-1 presents the SWAS Architecture.

As it is shown in the diagram, it follows a typical three layer Java Enterprise Architecture. Components included in business and presentation layers are JEE based. In the data layer, two components can be found:

- A relational database, that will be used by Ontology Registry to store ontology documents loaded into the GE.
- A Knowledge Base providing OWL-2RL support. This Knowledge Base is used by ontology and workspace registries to store ontology and workspace related metadata and by managing, querying and publishing modules to provide their functionality.

Business components interact with data layer components by means of two different mechanisms. To interact with the relational database, business components use JPA (Java Persistence API) that make business components database system independent. Unfortunately such an abstraction mechanism is not available for knowledge base required interaction so business components interacting with the knowledge base is knowledge base implementation dependant. In Semantic Web Application Support reference implementation, the combination of Sesame and OWLIM has been chosen as knowledge base implementation. Knowledge base independence feature is provided in the final release.

Business Layer contains following components:
• Ontology registry that manages ontologies loaded into the system and its related metadata. Operations such as retrieving / uploading ontology, retrieving / uploading metadata, etc are provided by this component. A description of methods provided for FI-WARE first release can be found in Backend Functionality section.

• Workspace registry that manages workspaces and their related metadata created by users to be used by their semantic enable applications. Operations such as creating / deleting a workspace, listing ontologies loaded into the workspace, etc are provided by this component.

• Publishing module that allow user to publish data into the GE. Data can be either ontologies or RDF serialized content. In case of ontologies, publishing module relies on ontology registry functionality. In case of RDF serialized content, publishing module stores the content in proper knowledge base workspace in collaboration with workspace registry. In both cases publishing module will update subscription module if needed. Description of methods belonging to this component will be described in future FI-WARE releases.

• Managing module that allow users to monitor the status of the GE. Operations such as retrieving a list of available ontologies, retrieving a list of subscriptions, etc are provided by this module. Managing module relies on the rest of business components to provide its functionality.

• Subscription module that allows users to subscribe to events produced in the GE. Operations such as subscribing to ontology updates or workspace modifications are provided by this module.

• Querying module that allows users to query their workspace following SPARQL Query Protocol.

In order to provide GE functionality in a platform independent way, several Rest APIs have been developed. In the final release, all the methods belonging to publish and managing APIs are provided. Therefore, clients or presentation layer applications interact with business components by means of HTTP requests / responses.

SWAS Engineering Environment provides comprehensive support for the ontology engineering life-cycle. Concrete details about SWAS Engineering Environment functionality are provided in Frontend Functionality section. SWAS architecture is based on Eclipse architecture, a leading development environment providing a technical layer for easy creation of new features and supported for a huge development community. Figure SWAS-2 shows the SWAS Engineering Environment architecture.
As it shown in the diagram SWAS Engineering Environment is divided into two layers, the SWAS Engineering Environment core and the contributed plug-ins. The SWAS Engineering Core provides the core ontology editing functionality. The contributed plug-ins are extensions that provide extra functionality supporting different phases of the NeOn Methodology.

### 12.2 User guide

Semantic Web Application Support Engineering Environment functionality is based on the functionality provided by the baseline asset WebProtégé. WebProtégé is a state-of-the-art, free, open source collaborative ontology development environment for the Web, which provides comprehensive support for the ontology engineering life-cycle, with the following features:

- Support for editing OWL 2 ontologies
- A default simple editing interface, which provides access to commonly used OWL constructs
- Full change tracking and revision history
- Collaboration tools such as, sharing and permissions, threaded notes and discussions, watches and email notifications
- Customizable user interface
- Customizable Web forms for application/domain specific editing
- Support for editing OBO ontologies
- Multiple formats for upload and download of ontologies (supported formats: RDF/XML, Turtle, OWL/XML, OBO, and others)

Due to its nature, it wouldn’t be possible to describe all SWAS Engineering Environment provided functionality in a service like manner. Therefore this user guide provides a short overview of the

Figure SWAS-1 presents an overview of WebProtégé GUI:

![WebProtégé GUI](image)

**Figure SWAS-1: SWAS GUI**

Some of the functionality provided by SWAS Engineering Environment is provided as new functionalities on WebProtégé. Figure SWAS-2 presents the WebProtégé perspective.
Under this perspective users are able to manage their projects and ontologies, creating or removing projects, loading or creating new ontologies, etc. Projects may be created in four different ways:

- **“Create Project..” button:** Allows to create a new project from zero, giving a Project Name and Description:

  ![Figure SWAS-3: SWAS Create Project](image3)

- **“Upload Project…” button:** Allows to create a new project using an existing ontology file. For that, it’s necessary to introduce a Project Name and Description, and look for the ontology file:

  ![Figure SWAS-4: SWAS Upload Project](image4)

- **“Upload from Workspace..” button:** Allows to create a new project by choosing a workspace from a list of workspaces, and one Context from the list of contexts associated with the chosen workspace:
“Upload from Ontology Registry” button: Allows to create a new project by choosing an ontology by its name and version.

Create Project: As the project is empty at the beginning, it is possible to create the whole structure using the WebProtégé basic functionalities. Users are able to manage (adding, removing, etc) main ontology contents such as classes, object properties and data properties. Once selected, project contents can be edited by means of a proper editor:
Once the project is created (for more detail of the basic use of WebProtégé, please see its user guide documentation at [http://protegewiki.stanford.edu/wiki/WebProtegeUsersGuide](http://protegewiki.stanford.edu/wiki/WebProtegeUsersGuide)), edited, and modified if needed, users can save it in two ways, by clicking one of the two buttons placed on:

**Figure SWAS-7: Building project structure**

1 – On the Ontology Registry: Export the current ontology (project) changes to Ontology Registry. In this case, as it is a new project, the ontology must be created on Ontology Registry. WebProtégé asks the version number under which the user wants to save the created ontology and saves the ontology on Ontology Registry using the project’s name and the given version number.

2 – On the Semantic Workspace: Saves current ontology (project) changes on Semantic Workspace. In this case, WebProtégé shows a list of the existing workspaces. Once selected one workspace, users can click on Select button to retrieve a list of contexts related to the selected workspace, but only those that can be modified (those contexts that ends with * cannot be modified and are not showed here, although they are related to the selected workspace).

**Figure SWAS-8: Saving project structure**

**Figure SWAS-9: Introduce ontology version**
The ontology (project) will be saved using the selected workspace and context. Also, users have the option to introduce a new context name, instead of choose one of the list, and then create a new context related to the chosen workspace. Here, the context list will only show contexts that can be changed (those finished with * will not appear). Here, WebProtégé uses the Backend Functionalities:

- List Workspaces to show the list of existing workspaces
- List Contexts to show the list of existing contexts related to a workspace
- List Ontologies: Retrieves a list with the ontologies included in a workspace (cannot be changed)
- Add Statement or Create Context with RDF depending on if user decide to choose one context or introduce a new one.

PD: Ontologies in List Ontologies are marked in List Contexts with * meaning they are not editable.

Upload Project: As the project is created from an existing ontology file, project structure will be already available. Users are able to manage (adding, removing, etc) main ontology contents such as classes, object properties and data properties. Once selected, project contents can be edited by means of a proper editor:
Once the project is created, edited, and modified if needed, users can save it in exactly the same way as the Create Project option.

Upload from Workspace Once users have selected a workspace from the list of existing workspaces and clicked on Select button, they can see a list of contexts related to the chosen workspace. Some of these contexts are not editable (those that ends with *). Users can create projects with both kinds of contexts (editable and non editable), and their name will end with (WS) or (WS)*, indicating both that they have been created from Semantic Workspace and that the context chosen was editable or non editable, respectively. By editable or non editable, we understand that the project can or cannot be saved (exported) on Semantic Workspace. When an non editable project is opened, users can make all actions allowed by WebProtégé basic application. When a editable project is opened, besides all actions allowed by WebProtégé basic application, users can save (export) the changes done on Semantic Workspace, clicking on icon Save:

Here, WebProtégé uses the Backend Functionalities:

- Load RDF into Context: Load RDF data into a context of an existing workspace

Upload from Ontology Registry When a project is created using an ontology from the Ontology Registry, it name ends with (OR). When one of these projects is opened users can make all actions allowed by WebProtégé basic application and also, can save (export) the changes done on the Ontology Registry, clicking on the icon:
Future Internet Core Platform

Figure SWAS-13: Project created using an ontology selected from the Ontology Registry (its name ends with (OR)). We can see the icon that allows users to save changes on Ontology Registry.

When clicking on this icon, a new window is opened, where users can introduce the new version number of the ontology:

Figure SWAS-14: Introduce ontology new version to save all changes Clicking OK, the changes made by users will be saved on the Ontology Registry.

Here, WebProtégé uses the Backend Functionalities:

- Upload ontology version: Uploads to the GE an ontology document and identifies it with a given ontology IRI and version IRI.

12.3 Programmers Guide

12.3.1 Backend Functionality

This section describes the functionality provided by the SWAS GE enabler as service invocation methods for both human or computer agents. As described in Architecture of the GE FIWARE ArchitectureDescription.Data.SemanticSupport, this functionality is accessible by means of Web API (also known as RESTful services). In the current FIWARE release the following subset of methods belonging to publishing and managing rest APIs has been provided:

- **Publishing Rest API.**
  - **Get ontology version:** Retrieves from the GE the ontology document identified by a given ontology IRI and version IRI. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI>/<version IRI>.
  - **Get ontology:** Similar to Get ontology version. It retrieves from the GE the latest version of the ontology document identified by a given ontology IRI. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI>.
o **Delete ontology version**: Removes from the GE the ontology document identified by a given ontology IRI and version IRI. To invoke the operation, a DELETE http request should be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI>/<version IRI>.

o **Delete ontology**: Similar to Delete ontology version. It removes from the GE the latest version of the ontology document identified by a given ontology IRI. To invoke the operation, a DELETE http request will be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI>.

o **Upload ontology version**: Uploads to the GE an ontology document and identifies it with a given ontology IRI and version IRI. To invoke the operation, a PUT http request should be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI>/<version IRI> with an file attachment including the ontology RDF/XML serialization.

o **Upload ontology**: Similar to Upload ontology version. Uploads an ontology document to the GE and identifies it with a given ontology IRI and with the latest version IRI available. To invoke the operation, a PUT http request should be sent to http://<ge url location>/ontology-registry/ontologies/<ontology IRI> with an file attachment including the ontology RDF/XML serialization.

o **Get ontology version metadata**: Retrieves from the GE an ontology document containing the metadata related to an ontology document identified by a given ontology IRI and version IRI. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI>/<version IRI>.

o **Get ontology metadata**: Similar to Get ontology version metadata. It retrieves from the GE an ontology document containing the metadata related to the latest version of the ontology document identified by a given ontology IRI. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI>.

o **Delete ontology version metadata**: Removes from the GE the metadata related to an ontology document identified by a given ontology IRI and version IRI. To invoke the operation, a DELETE http request will be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI>/<version IRI>.

o **Delete ontology metadata**: Similar to Delete ontology version metadata. Removes from the GE the metadata related to the latest version of the ontology document identified by a given ontology IRI. To invoke the operation, a DELETE http request should be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI>.

o **Upload ontology version metadata**: Uploads to the GE an ontology document containing metadata related to an ontology document identified by a given ontology IRI and version IRI. To invoke the operation, a PUT http request should be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI>/<version IRI> with an file attachment including the metadata RDF/XML serialization. Metadata uploaded must comply to OMV (Ontology metadata vocabulary).

o **Upload ontology metadata**: Similar to Upload ontology version metadata. It uploads to the GE an ontology document containing metadata related to the latest version of an ontology document identified by a given IRI. To invoke the operation, a PUT http
request should be sent to http://<ge url location>/ontology-registry/metadata/<ontology IRI> with an file attachment including the metadata RDF/XML serialization. Metadata uploaded must complain to OMV (Ontology metadata vocabulary).

- **Managing Rest API.**
  - **List ontologies**: Retrieves an XML document containing the list of ontology documents and their versions loaded into the GE. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/mgm/list. As a result, an xml encoding the requested information will be sent as response.
  - **List ontology versions**: Similar to List ontologies. Retrieves an XML document containing the versions of an ontology document identified by a given ontology IRI loaded into the GE. To invoke the operation, a GET http request should be sent to http://<ge url location>/ontology-registry/mgm/<ontology IRI>. As a result, an xml encoding the requested information will be sent as response.

- **Workspaces Management Rest API.**
  - **List Workspaces**: Retrieves an XML document which contains a list of all workspaces included in the server, a GET http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/mgm/list. As result, a xml encoding the list of workspaces.

- **Workspace Operations Rest API.**
  - **Create Workspace**: Creates a new semantic workspace, a POST http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]. As a result, an xml encoding the result of the operation.
  - **Remove Workspace**: Remove an existing semantic workspace, a DELETE http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]. As a result, an xml encoding the result of the operation.
  - **Duplicate Workspace**: Creates a duplicate of an existing workspace with his metadata (ontologies and triples), a PUT http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/duplicate. As result, a xml encoding the result of the operation.
  - **Execute Query**: Execute a SPARQL query into an existing workspace, a POST http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/sparql/. As result, a xml encoding the result of the query.
  - **Get Workspace**: Retrieves the RDF from a specific workspace, a GET http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]. As result, a RDF/XML encoding the data.
  - **Get ontologies updates**: Retrieves a list of available updates for the ontologies included in a workspace, a GET http request should be sent to http://<ge url location>/semantic-
workspaces-service/rest/workspaces/[WORKSPACE_NAME]/checkupdates. As result, a XML encoding the list of updates.

- **Load Ontology**: Load an ontology into a workspace from a specific ontology registry, a POST http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/ontology/[ONTOLOGY_NAME]. As result, a XML encoding the operation result.

- **List Ontologies**: Retrieves a list with the ontologies included in a workspace, a GET http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/ontology/list. As result, a XML encoding the list of ontologies.

- **Update ontology**: Update an ontology included in a workspace using a specific ontology registry, a GET http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/ontology/[ONTOLOGY_NAME]/update. As result, a XML encoding the result of the operation.

- **Delete Ontology**: Delete an ontology from a workspace, a DELETE http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/ontology/[ONTOLOGY_NAME]. As result, a XML encoding the result of the operation.

- **Create Context with RDF**: Create a context with RDF data into an existing workspace, a POST http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]. The context will be cleared and then the RDF will be loaded. As result, a xml encoding the result of the operation.

- **Load RDF into Context**: Load RDF data into a context of an existing workspace, a PUT http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]. The context will be cleared and then the RDF will be loaded. As result, a xml encoding the result of the operation.

- **Delete Context**: Removes a context of a specific workspace, a DELETE http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]. As result, a XML encoding the result of the operation.

- **List Contexts**: List the contexts included in a specific workspace, a GET http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/list. As result, a XML encoding the list of context.

- **Add Statement**: Add a statement (RDF triple) into a specific workspace, a POST http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]/statement. As result, a XML encoding the result of the operation.

- **Remove Statement**: Remove a statement (RDF triple) from a specific workspace, a DELETE http request should be sent to http://<ge url location>/semantic-workspaces-service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]/statement. As result, a XML encoding the result of the operation.
service/rest/workspaces/[WORKSPACE_NAME]/context/[CONTEXT_NAME]/statement. As result, a XML encoding the result of the operation.

12.3.2 Invocation of services

All methods described in the users guide above are RESTful services and therefore can be invoked by means of regular HTTP requests or using a REST API.

As a matter of example, the following Java code shows how to invoke Upload ontology version by using Jersey API.

```java
File file = new File("Directory");
if(!file.isDirectory())
    throw new Exception("Invalid starting point");
File[] contexts = file.listFiles();
//Iterating over folders
for(int i = 0; contexts != null && i < contexts.length; i++){
    // Iterating over files
    File[] ontologies = contexts[i].listFiles();
    for(int j = 0; ontologies != null && j < ontologies.length; j++){
        if(ontologies[j].isFile()){
            System.out.println(ontologies[j].getPath());
            String ontologyIRI = ontologies[j].getAbsolutePath().substring(ontologies[j].getPath().lastIndexOf(\'\'\') + 1);
            String versionIRI = "";
            versionIRI = "1.0";
            FormDataMultiPart form = new FormDataMultiPart();
            ContentDisposition cd = form.getContentDisposition();
            FileDataBodyPart fdp = new FileDataBodyPart(ontologyIRI, ontologies[j], MediaType.APPLICATION_OCTET_STREAM_TYPE);
            form.bodyPart(fdp);
            Client c = Client.create();
        }
    }
}
```
String resourceName = "http://localhost:8080/ontology-registry-service/webresources/ontology-registry/" + ontologyIRI;
    if (versionIRI != null && versionIRI.compareTo("") != 0)
        resourceName = resourceName + "/" + versionIRI;
    System.out.println("resourceName: " + resourceName);
    WebResource r = c.resource(resourceName);
    String response = r.type(MediaType.MULTIPART_FORM_DATA).accept(MediaType.TEXT_PLAIN_TYPE).post(String.class, form);

    System.out.println("Ontology " + ontologyIRI + "/" + versionIRI + " loaded: " + response);
}

System.out.println("Repository loaded");

Some other methods (those ones who rely on GET requests) can just be invoked by introducing a simple URL into a web browser. Figure SWAS-7 shows Get ontology version invocation using this mechanism.

As a matter of example of the method GetOntologyVersion see below the request and expected response of the service:

Figure SWAS-7: Invoking Get ontology version using web browser
<table>
<thead>
<tr>
<th>Verb</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/ontologies/{ontologyIRI}/{versionIRI}</td>
<td>Retrieves the ontology field identified by a given ontology IRI and version IRI</td>
</tr>
</tbody>
</table>

**Response codes:**
- HTTP/1.1 200 - If the ontology is successfully retrieved from the registry
- HTTP/1.1 404 - If there is no ontology in the registry identified by given ontology IRI and version IRI
- HTTP/1.1 500 - If there are some unidentified error.

**Request example:**
GET /ontology-registry-service/webresources/ontology-registry/ontologies/merm.owl/7 HTTP/1.1
Host: localhost:8080
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

**Response example:**
HTTP/1.1 200 OK
Content-Type: application/xml

```xml
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [ 
   <!ENTITY sioc "http://rdfs.org/sioc/ns#" >
   <!ENTITY dcterms "http://purl.org/dc/terms/" >
   <!ENTITY foaf "http://xmlns.com/foaf/0.1/" >
   <!ENTITY sawsdl "http://www.w3.org/ns/sawsdl#" >
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
   <!ENTITY owl2 "http://www.w3.org/2006/12/owl2#" >
   <!ENTITY dc "http://purl.org/dc/elements/1.1/" >
   <!ENTITY posm "http://www.wsmo.org/ns/posm/0.1#" >
   <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
   <!ENTITY swr1x "http://www.w3.org/2003/11/swrlx#" >
]>
```
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
<!ENTITY rdf "http://www.w3.org/1999/02/rdf-syntax-ns#">
<!ENTITY so "http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#">

<rdf:RDF xmlns="http://www.tatoo-fp7.eu/tatooweb/merm#"
    xml:base="http://www.tatoo-fp7.eu/tatooweb/merm"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:so="http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#"
    xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
    xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
    xmlns:owl2="http://www.w3.org/2006/12/owl2#"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:sioc="http://rdfs.org/sioc/ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns:rdf="http://www.w3.org/1999/02/rdf-syntax-ns#"
    xmlns:posm="http://www.wsmo.org/ns/posm/0.1#">
    <owl:Ontology rdf:about="http://www.tatoo-fp7.eu/tatooweb/merm">
        <owl:imports rdf:resource="http://www.tatoo-fp7.eu/tatooweb/V1_0/sioc_pruned"/>
    </owl:Ontology>
</rdf:RDF>
<owl:DatatypeProperty rdf:about="http://www.tattoo-fp7.eu/tatooweb/merm#hasEvaluationMetric">
  <rdfs:label xml:lang="en">has Evaluation Metric</rdfs:label>
  <rdfs:range rdf:resource="&xsd;String"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:about="http://www.tattoo-fp7.eu/tatooweb/merm#dateEvaluated">
  <rdfs:label xml:lang="en">date evaluated</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="http://www.tattoo-fp7.eu/tatooweb/merm#dateProvided"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&dc;publisher">
  <rdfs:label xml:lang="en">publisher</rdfs:label>
  <rdfs:comment>The person who publishes the resource in real world</rdfs:comment>
  <rdfs:range rdf:resource="&foaf;Agent"/>

</owl:ObjectProperty>
...

13 StreamOriented - Users and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

13.1 Introduction

The Stream Oriented GE Kurento is a multimedia platform helping developers to add multimedia capabilities to their applications. The core element is the Kurento Media Server (or "KMS", for short), a Gstreamer based multimedia engine that provides the following features:

- Networked streaming protocols, including HTTP working as client and server, RTP and WebRTC.
- Media transcodification between any of the codecs currently supported by Gstreamer.
- Generic support for computational vision and augmented reality filters.
- Media storage supporting writing operations for WebM and MP4 and reading operations for any of Gstreamer’s muxers.

Java and Javascript SDKs are available for developers, to incorporate the above features in their applications.

13.1.1 Background and Detail

This User and Programmers Guide relates to the Stream Oriented GE which is part of the Data/Context Management chapter. Please find more information about this Generic Enabler in the following FIWARE.OpenSpecification.Data.StreamOriented.

13.1.2 About this guide

This guide, as the Stream Oriented GE Kurento itself, is under very active development. Many features are constantly evolving and are not completely documented yet. You can contribute to complete this guide and also to Kurento effort by joining its community.

13.2 User Guide

The Stream Oriented GE Kurento offers APIs devoted to programmers, not to final users, so this section does not apply.

13.3 Programmer Guide

13.3.1 Things you need to know before start programing

- The Stream Oriented GE Kurento software is released under LGPL version 2.1 license. This is quite a convenient license for programmers, but it is still recommended you check if it actually fits your application needs.
Maven is used as dependency management tool for Java SDKs. Most likely Gradle can also be used, but we still haven't tested it. If you don't use any dependency management you can still download the KMF API Bundle and incorporate manually all dependencies to your application, but this is not recommended.

Spring framework is extensively used for lifecycle management of Java components. Developers are not required to develop Spring applications when using the Stream Oriented GE Kurento - Content API in Java EE environments, but they'll have to when developing applications with Media API.

13.3.2 Quick start
This section is intended to provide the very basic steps required to integrate the Stream Oriented GE Kurento’s framework into applications.

13.3.2.1 Basic Setup

- **Install and configure Kurento Media Server (KMS):** This piece of software is the actual engine providing media processing and delivery.
- **Install and configure JBoss 7 Application Server (KAS):** This is a Java EE container that hosts the server side of applications. Other Java enterprise servers can be used, although no support from Kurento will be provided. This server will also be called Kurento Application Server (KAS) through the document.

The StreamOriented - Installation and Administration Guide provides detailed information on installation and setup of above components.

13.3.2.2 Create your first application

13.3.2.2.1 Server side of your first application

The Stream Oriented GE Kurento server SDK is a Java library known as Kurento Media Framework (KMF).

The following steps are required to create a Kurento based application:

1. Create a Maven web project with your favourite IDE. You can use the following pom.xml template. Please notice that Java 1.7 is required to compile KMF-based Java projects.

2. ```xml
   <?xml version="1.0" encoding="UTF-8"?>
   <project xmlns="http://maven.apache.org/POM/4.0.0"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>my.organization</groupId>
   <groupId>my.organization</groupId>
```
9. `<artifactId>my-kurento-demo</artifactId>`
10. `<version>1.0.0</version>`
11. `<packaging>war</packaging>`
12. `<properties>`
13. `<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>`
14. `<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>`
15. `<maven.compiler.source>1.7</maven.compiler.source>`
16. `<maven.compiler.target>1.7</maven.compiler.target>`
17. `</dependencies>`
18. `</project>`
19. `</properties>`
20. `</project>`
21. **You can add KMF dependencies to the pom.xml file as follows:**
22. `<dependencies>`
23. `...`
24. `<dependency>`
25. `<groupId>com.kurento.kmf</groupId>`
26. `<artifactId>kmf-content-api</artifactId>`
27. `<version>4.2.2</version>`
28. `</dependency>`
29. `...`
30. `</dependencies>`

**NOTE:** We are in active development. Be sure that you have the latest Kurento version in your POM. You can find it in the [Maven Central](https://mvnrepository.com/artifact/com.kurento/kmf-content-api) section of kurento.org.
32. **KMF requires to be deployed in an Application Server with a container supporting the version 3.0 of the Servlet specification. Therefore, ensure that this version is established in WEB-INF/web.xml:**
34. `<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`  
35. `xmlns="http://java.sun.com/xml/ns/javaee"  
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"  
36. `xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"`  
37. `version="3.0">`  
38. `</web-app>`

39. Create a properties file named `<code>kurento.properties` including the following configuration keys:

40. 
41. # Put here the IP address where the KMS process is executing  
42. # If you launched KMS in the same hosts where you are executing KAS, let it as 127.0.0.1  
43. `thriftInterfaceConfiguration.serverAddress=127.0.0.1`  
44.  
45. # Put here the port where KMS management daemon is bound  
46. # If you did not modify KMS default configuration, let it as 9090  
47. `thriftInterfaceConfiguration.serverPort=9090`  
48.  
49. # Put here the IP address where KAS management handler must listen  
50. # If you launched KMS int the same host where you are executing KAS, let it as 127.0.0.1  
51. `mediaApiConfiguration.handlerAddress=127.0.0.1`  
52.  
53. # Port where KAS management daemon will bind  
54. # Your can choose the port you want. By default we assume 9100.  
55. `mediaApiConfiguration.handlerPort=9100`

*Kurento* framework will search this file in the following locations (in the specified order):

- *JBoss configuration folder defined by property:* `${jboss.server.config.dir}`
- *Directory specified by java option* `kurento.properties.dir` - `Dkurento.properties.dir=/home/user/kurento`
Create a Java Class that extends HttpPlayerHandler, and add the annotation @PlayerService. You’ll have to implement the method onContentRequest() to set the media resource to be played.

```java
import com.kurento.kmf.content.HttpPlayerHandler;
import com.kurento.kmf.content.HttpPlayerService;
import com.kurento.kmf.content.HttpPlayerSession;

@HttpPlayerService(path = "/playerService", useControlProtocol=false)
public class MyService extends HttpPlayerHandler {
    @Override
    public void onContentRequest(HttpPlayerSession session) throws Exception {
        session.start("file:///path/to/myvideo.webm");
    }
}
```

Place a WebM video so that the KMS process can reach it at whatever path you specified in /path/to/myvideo.webm. This video will be the one read by the player element. You can replace the file:// type URL by another one where a WebM file can be found, such as http://media.w3.org/2010/05/sintel/trailer.webm

Package the project into a .war file.

Deploy your project into JBoss 7 server installed during the basic setup and start it.

```bash
sudo cp mykurento.war $JBOSS_HOME/standalone/deployments
sudo /etc/init.d/jboss7 start
```

13.3.2.2 Client side of your first application

The Stream Oriented GE Kurento is designed to work with plain HTML5 code. For testing your application, you just have to include a <video> tag linked to the service URL defined above. To do it, for example, create an HTML file in your local machine containing the code shown below and open it with your browser.

---

D.6.4.3: FI-WARE Users and Programmers Guide 476
You can read also section #Programming with the Stream Oriented GE HTML5 SDK to find out more sophisticated ways to access media resources.

13.3.2.3 Next steps

- Read #Basic streaming concepts in order to understand how Stream Oriented GE Kurento features can help you to build multimedia applications.
- Review #Programming with the Stream Oriented GE Java EE Content API for a detailed reference on content services.
- Go to #Programming with the Stream Oriented GE Java Media API for a detailed explanation about how to achieve full control of Kurento Media Server.
- Review #Programming with the Stream Oriented GE HTML5 SDK for a detailed reference of capabilities available in browsers.

13.3.3 Basic streaming concepts

There are several streaming concepts that might be of interest, in order to know the precise behaviour that can be expected when adding multimedia resources to applications.

Any streaming protocol requires two main components: a control function to manage connection setup, and a media function that actually provides media process & transfer capabilities. For true streaming protocols, like RTP, RTSP, RTMP or WebRTC there is a clear differentiation between both functions. Actually RTP is the media function of the RTSP protocol. RTP can also be used in conjunction with other control protocols like SIP or XMPP. WebRTC is a media function like RTP and it also requires a control protocol that negotiates connection setup.

Streaming over HTTP (a.k.a. HTML5 streaming) is somehow special because HTTP is a not protocol designed for media transfer. HTML5 streaming sessions start with the browser sending a GET request to the server. In this step both browser and server play the control function role. The server then maps the URL to the actual resource, encapsulates its content in the response, and sends it back to the <video> component, just like any download operation. Now browser and server switch to the media function. There isn't a clear differentiation between control and media functions that are played sequentially by the same element in both sides. Apart form this function mixup, many people will argue that HTTP is not really a streaming protocol, since there is no relation at all between media transfer pace an playing pace, i.e. the network transfer rate is not limited by the media consumption rate and you might find situations where the whole content of a 1 hour video is already downloaded when still playing the first minute.
There is quite an important and somehow confusing concept related to the capability to jump to a time position within a stream. This operation is normally called SEEK and streams that supports it are called seekable. Those not supporting SEEK operation are called live or non-seekeable. There are two conditions a stream must meet in order to be seekable. First, the control protocol must provide a SEEK command and second, the media resource must be completely available before the stream starts transmission. The reason for the second condition, is because seeks must specify somehow the file position where the stream must jump, and that requires to know in advance the size or length of the media resource. Hence the whole resource must be available in advance. Streaming protocols like RTSP and HTTP use header Range as a mean to build seek command. When the <video> component in an HTML5 application request a seek operation, the browser sends a new GET request with the appropriate Range header. But this is only available if the server provided the resource size in advance in the first request (the one that initiated the stream). If resource size is not available at start time, the video component does not show any kind of progress bar, switching into live mode. Stream Oriented GE Kurento is currently supporting only live mode, independently of whether the media resource is available in advance or not.

When designing streaming services it is also very important to determine the type of service that is being offered. There are two main classifications for streaming services: Video on demand (VoD) and Broadcast. Main difference between these two services is the streaming time scale. In Broadcast mode any new client connecting to the streaming service assumes the time scale defined by the source, and this time scale is shared among all connected clients. In VoD service a new time scale is built for each client. The client not only selects resource, but also the time of origin. When many VoD clients access the same resource, each one has its own time scale, and this time scale is reset if the client breaks the connection. Stream Oriented GE Kurento is currently supporting Broadcast services, but in future versions it will also support true VoD mode.

13.3.4 Stream Oriented GE Kurento API architecture

The Stream Oriented GE Kurento is a multimedia platform that provides streaming capabilities in a very flexible way. As described in the Architecture Description, Kurento is a modular system where a set of basic functional blocks, called MediaElements, that live in containers, called MediaPipelines, are connected together to build multimedia services. There are three main MediaElement families:

- **Endpoints**: Endpoints provide transfer capabilities, allowing bidirectional communication channels with external systems. Supported protocols include muxers, like WebM or MP4 for file operations and following streaming protocols: HTTP, RTP and WebRTC.
- **Filters**: Filters are responsible of media processing, including transcodification, computer vision, augmented reality, etc.
- **Mixers**: Mixers combines the stream from endpoints. They are also known as Hub. The main mixers types are Dispatcher and Composite.

The Stream Oriented GE Kurento consists of two main software components: Kurento Media Server (KMS) and Kurento Media Framework (KMF):

- **KMS**: Kurento Media Server is a stand-alone server responsible of the media process and delivery. It is the component that hosts Endpoints and Filters.
- **KMF**: Kurento Media Framework is the SDK that enables applications to control KMS features and publish multimedia services. KMF provides the following APIs:
Future Internet Core Platform

- **Content API**: High-level middleware layer of services intended to simplify communications with clients. It also offers Open API to clients.
- **Media API**: Low-level API that provides full control of KMS elements. It is normally used in conjunction with **Content API**.
- **HTML5 SDK**: Javascript SDK intended to provide better control of media playing in web applications. It uses Open API (based on JSON-RPC over http and websockets) to communicate with Content API in the server.

### 13.3.5 Programming with the Stream Oriented GE Java EE Content API

The **Content API SDK** is intended to simplify setup and management of multimedia connections between KMS and web applications. Built on top of the **Java EE Servlet API**, implements a REST-like interface based on JSON-RPC that controls following multimedia services:

- **HTTP services**: Enables download and upload of multimedia contents.
- **RTP services**: Allows the setup of bidirectional RTP connections.
- **WebRTC services**: Controls WebRTC connections with browsers and mobile devices implementing the WebRTC stack.

It is important to notice that the **Content API** is just a KMS control interface and does not handles media directly.

#### 13.3.5.1 Content services

Applications offering multimedia services have to setup and manage **KMS Endpoints**. The **Endpoints** are heterogeneous and their operation depends on the underlying streaming protocol. This is the reason why the **Content API** defines the concept of **content service**, as a mechanism to provide a simple and homogeneous interface, for the creation and management of multimedia connections.

A **content service** consists of a standard **Java bean** implementing the **service handler** interface. **Service handlers** are identified because they are annotated as follows:

- **@HttPlayerService**: Declares a player service intended to deliver content to **HTML5** `<video>` elements. The **service handler** must extend class `HttpPlayerHandler`.

```java
@HttPlayerService(path = "/myPlayerService")
public class MyService extends HttpPlayerHandler{
    ...
}
```

- **@HttpRecorderService**: Allows the application to publish a recorder service, enabling media injection into KMS through the **HTTP file upload** protocol. The recorder **service handler** must extend class `HttpRecorderHandler`.

```java
@HttpRecorderService
public class MyRecorderService extends HttpRecorderHandler{...}
```
Future Internet Core Platform

- @HttpRecorderService(path = "/myRecorderService")

```java
public class MyService extends HttpRecorderHandler{
    ...
}
```

- @RtpContentService: Defines a bidirectional RTP connection. The service handler must extend class RtpContentHandler.

```java
@RtpContentService(path = "/myRtpService")
public class MyService extends RtpContentHandler{
    ...
}
```

- @WebRtcContentService: Intended for bidirectional WebRTC connections. Its service handler must extend class WebRtcContentHandler

```java
@WebRtcContentService(path = "/myWebRtcService")
public class MyService extends WebRtcContentHandler{
    ...
}
```

When application starts, the Content API framework searches for content service annotations, instantiating a service entry point for each service handler found. Internally, a service entry point is basically an HTTP servlet, mapped to a service URL where clients can send HTTP requests with control commands. Developers do not have to care about servlet configuration or initialisation, as the Content API takes care of this operations. The service URL has the format specified below:

http://myserver/myApp/myServiceName

where

- **myserver**: is the IP address or hostname of the Kurento Application Server.
- **myApp**: is the application context, namely the WAR archive name if not otherwise specified.
- **myServiceName**: is the value given to the path attribute of service annotation.
As a summary, in order to create a content service the application must implement a service handler, which is a Java bean with a common interface. The Content API instantiates an HTTP servlet for each service handler found. This servlet is known as the service entry point, and can be reached at the service URL. Service operation and management is independent of the underlying KMS Endpoint type. It is important to understand that developers do not need to care about instantiation of service entry points’ servlets and that these are used just for control purposes and not for media delivery.

13.3.5.1 HTTP Player Service
The HTTP Player service instantiates a download service intended for HTML5 streaming. Method onContentRequest() is called every time the service entry point receives a GET request from a client using Open API (directly or with HTML5 SDK).

```java
import com.kurento.kmf.content.HttpPlayerHandler;
import com.kurento.kmf.content.HttpPlayerService;
import com.kurento.kmf.content.HttpPlayerSession;

@HttpPlayerService(path = "myPlayerService")
public class MyService extends HttpPlayerHandler{

    @Override
    public void onContentRequest(HttpPlayerSession session) throws Exception {
        session.start("file:///path/to/myvideo");
    }
}
```

KMS instantiates HTTP Endpoints on behalf of this service every time a new request arrives. HTTP Endpoints transform content on the fly to WebM (by default) or MP4 before encapsulation and delivery, allowing source files to have any format supported by Gstreamer.

HTML5 browsers can access the content by adding the service URL as source of the tag <video>.

```html
<video>
    <source src="http://myServer/myApp/myPlayerService" type="video/webm"/>
</video>
```
Current version of the Content API only supports live mode independently of the nature of the media archive. Future versions will support pseudo-streaming for media resources whose file size can be known before transmission is started.

- **Known issue**: A bad behavior with Chrome has been observed, when the service URL is placed in the address bar of the browser. This is due to a reconnection Chrome performs when it detects MIME of type video or audio. Root cause for this problem relates to the fact that Kurento provides VoD services based on top of a broadcast service, and time scale initialisation is not performed on reconnection. Future versions will provide true VoD capabilities, solving this problem.

### 13.3.5.1.2 HTTP Recorder Service

HTTP recorder service allows applications to inject contents into KMS, through the standard file upload protocol. Method `onContentRequest()` will be called for each multipart/form POST request received in the service entry point. The receiver HTTP Endpoint will search for the first content part with a supported multimedia format, and will feed the media resource specified by the handler (file://myfile.webm). Recorder service accepts from a client any multimedia format supported by Gstreamer, but transforms content to WebM or MP4 before storing it.

```java
import com.kurento.kmf.content.HttpRecorderHandler;
import com.kurento.kmf.content.HttpRecorderService;
import com.kurento.kmf.content.HttpRecorderSession;

@HttpRecorderService(path = "/myRecorderService")
public class MyRecorderService extends HttpRecorderHandler {

    @Override
    public void onContentRequest(HttpRecorderSession contentSession) throws Exception {
        contentSession.start("file:///myfile.webm");
    }
}
```

Browsers can access this service through HTML forms, addressed to the service URL, that include inputs of type file. If more than one file is present the request will accept only first one found.

```html
<form action="http://myServer/myApp/myRecorderService">
    File: <input type="file" name="data" >
</form>
```
13.3.5.1.3  **RTP & WebRTC Service**

RTP and WebRTC require a negotiation process where each side sends its connection details and supported formats encoded in a *SDP (Session Description Protocol)* packet. RTP and WebRTC services hide negotiation complexity, offering applications the same interface used for the well-known HTTP services. Method `onContentRequest()` is called each time a *POST* request with a connection offer is received by the *service entry point*.

```java
import com.kurento.kmf.content.WebRtcContentHandler;
import com.kurento.kmf.content.WebRtcContentService;
import com.kurento.kmf.content.WebRtcContentSession;
import com.kurento.kmf.media.MediaPipeline;
import com.kurento.kmf.media.MediaPipelineFactory;
import com.kurento.kmf.media.PlayerEndpoint;
import com.kurento.kmf.media.RecorderEndpoint;

@WebRtcContentService(path = "/myWebRtcService")
public class MyWebRtpService extends WebRtcContentHandler {

    @Override
    public void onContentRequest(WebRtcContentSession contentSession) throws Exception {

        contentSession.start("file:///fileToSend.webm","file:///fileToRecord.webm");
    }
}
```

RTP and WebRTC are bidirectional protocols that can send and receive at the same time. For that reason, *start* method requires both *source* and *sink* elements.

In the previous example, the received media from the client will be recorded into `fileToRecord.webm` and the media to deliver to the client is read from `fileToSend.webm`. The start method it not limited to read from files and write to files. More complex media pipelines can be created with *Media API* as we will see in the following sections of this document.
The client starting the communication with the server specifies some constraints for media direction in
the negotiating phase. The handler can access to this constraints individually for video and audio
streams with methods `WebRtcContentSession.getVideoConstraints()` and
`WebRtcContentSession.getAudioConstraints()`. These methods return one of the following
values:

- **SENDONLY**: KMS receives media from the client and does not deliver to it.
- **RECVONLY**: KMS delivers media to the client and does not receive from it.
- **SENDRECV**: KMS sends and receives media at the same time.
- **INACTIVE**: There is no media transfer in any direction, independently of any player or recorded
  connected. This is useful when only video or audio can be transmitted.

Played file can take any format supported by `Gstreamer`, and will be translated to a format negotiated
with the remote peer. Stored file will be converted to `WebM` or `MP4` from the format negotiated with
remote peer.

13.3.5.2 Content Session & Media lifecycle

The content session is the mechanism offered by the Content API to manage multimedia transactions. Its
state depends on media events detected in the Endpoint, control events detected in the service entry
point and application commands.

The content session is created when a request is received in the service entry point. Method
onContentRequest() is called in the service handler, so the application can accept or reject requests.
Rejected requests must provide the message and the HTTP error code that will be returned to browser.

```java
@Override
public void onContentRequest(WebRtcContentSession contentSession)
   throws Exception {
    contentSession.terminate(404, "Content not found");
}
```

When the service handler wants to accept a request, it must provide the source and sink media
resources that will be connected to the Endpoint. Method `start()` is called for this purpose.

```java
@Override
public void onContentRequest(WebRtcContentSession contentSession)
   throws Exception {
    contentSession.start("file:///fileToSend.webm","file:///fileToRecord.webm");
}
```
The **Endpoint** informs applications when a media transfer starts by calling the optional method `onContentStarted()`. 

```java
@Override
public void onContentStarted(WebRtcContentSession contentSession)
  throws Exception {
    // Execute specific application logic when content (media) starts being served to the client
}
```

Optional method `onSessionTerminated()` is called when **Endpoint** completes media transfer. The **content session** termination code is provided in this call.

```java
@Override
public void onSessionTerminated(WebRtcContentSession contentSession,
                                 int code, String reason) throws Exception {
    // Execute specific application logic when content session terminates
}
```

The **content session** is terminated automatically if the **Endpoint** experiences an unrecoverable error not caused by a direct application command. Events like client disconnection, file system access fail, etc. are the main error cause. Any of these exceptions can be handled on `onUncaughtException()`. 

```java
@Override
public void onUncaughtException(HttpPlayerSession contentSession,
                                 Throwable exception) throws Exception {
    // Execute specific application logic if there is an unrecoverable
    // error on the media infrastructure. Session is destroyed after
    // executing this code
}
```

If exceptions are not handled, they will be propagated and the method `onSessionError()` will be called with the error code and description.
The content session is able to store and manage application attributes through its lifecycle, in a similar way as HttpSession does. Method setAttribute() stores an object of any class that can later be retrieved with method getAttribute() or deleted with method removeAttribute().

```java
@Override
public void onSessionError(WebRtcContentSession contentSession, int code, String description) throws Exception {
    // Execute specific application logic if there is an unrecoverable
    // error on the media infrastructure. Session is destroyed
    // after
    // executing this code
}
```

One important feature of the content session, is its capability to share real time information with clients through a bidirectional channel. In order to interchange messages with a client, the Open API has to be

```java
@Override
public void onContentRequest(WebRtcContentSession contentSession) throws Exception {
    contentSession.setAttribute("source", "source.avi");
    contentSession.setAttribute("sink", "sink.webm");
    //...
}
```

```java
@Override
public void onContentStarted(WebRtcContentSession contentSession) throws Exception {
    String source = (String) contentSession.getAttribute("source");
    String sink = (String) contentSession.getAttribute("sink");
    log.info("Start playing: " + source);
    log.info("Start recording:" + sink);
}
```
used. In the web browsers is recommended connect to the server with the HTML5 SDK, because it fully implements OpenAPI.

The OpenAPI is implemented following a signalling protocol based on JSON-RPC 2.0. Messages can be interchanged between the service handler and the client, while the content session is active. Method publishEvent() can be used from the handler to sent events to the client. This capability is quite useful combined with computer vision filters, as it allows sending events to clients coming from video content analysis (e.g. plate recognised, QR code detected, face detected, etc.).

```java
@Override
public void onContentStarted(WebRtcContentSession contentSession) throws Exception {
    ContentEvent event = new ContentEvent();
    event.setType("title");
    event.setData("My Video");
    contentSession.publishEvent(event);
}
```

Clients can also send messages to the content session through this channel. Client messages are called commands, and are received on handler method onContentCommand()

```java
@Override
public ContentCommandResult onContentCommand( WebRtcContentSession contentSession, ContentCommand contentCommand) throws Exception {
    String data = contentCommand.getData();
    String type = contentCommand.getType();

    //Process command...

    return new ContentCommandResult("OK");
}
```

See the Open API specification for a detailed reference of available commands and events that can be exchanged between service handlers and clients.
13.3.5.3 **Content identification**

Content identification can be understood as the process of mapping media resources to URLs. The rules and algorithms used are quite variable and application dependent, although there are several possible strategies. A very common one is the direct mapping between the URL path and a file system path, which actually is the strategy used by the most HTTP servers to map static resources. Other alternative is to assign a content ID to each media resource. This content ID can be placed in the URL’s path info or in the query string, as parameter. The server searches for the content ID in the appropriate place and looks up a mapping table.

The *content session* provides method `getContentId()` that returns the path info of requested URL’s, assuming the content ID is placed there, as shown below:

**Content URL**: `http://myserver/myApp/myServicePath/{contentId}`

- `myserver`: IP address or name of *Kurento Application Server*
- `myApp`: Application name. Normally is the WAR archive name
- `myServicePath`: Value assigned to `path` attribute of service annotation
- `{contentId}`: URL’s path info. Everything left between service name and the URL’s query string.

```java
@Override
public void onContentRequest(HttpPlayerSession contentSession) throws Exception {
    String contentId = contentSession.getContentId();
    contentSession.start("file:///path/to/myrepo/" + contentId);
}
```

If a different content ID strategy, based in a query string parameter or the like, is used, the application can directly access the requested URL through method `getHttpServletRequest()`

```java
@Override
public void onContentRequest(HttpPlayerSession contentSession) throws Exception {
    String contentId;
    HttpServletRequest request = contentSession.getHttpServletRequest();
    request.getContextPath();
    request.getQueryString();
}
```
// build content ID from URL

collectionSession.start("file:///path/to/myrepo/" +contentId);
}

Notice you'll have to add the Servlet API dependency to the pom.xml before being able to import HttpServletRequest in your code.

```
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>3.0.1</version>
   <scope>provided</scope>
</dependency>
```

13.3.5.4 Media resource management

The Content API does not require an explicit resource management unless the application directly builds KMS MediaElements. Lifecycle of created MediaElements is not managed anymore by the content session, so the application must care about how and when resources are released. In order to facilitate resource management, the content session provides a mechanism to attach MediaElements to the session lifecycle. Method releaseOnTerminate() can be used for this purpose.

```
MediaPipelineFactory mpf = collectionSession.getMediaPipelineFactory();
MediaPipeline mp = mpf.create();

PlayerEndpoint player =
   mp.createPlayerEndpoint("file:///path/to/myplayed.avi");
contentSession.releaseOnTerminate(player);

HttpGetEndpoint httpEndpoint =
   mp.newHttpGetEndpoint().terminateOnEOS().build();
player.connect(httpEndpoint);
contentSession.start(httpEndpoint)
```
Single elements can be attached to a session lifecycle, but also the whole *MediaPipeline*, depending on application needs.

```java
MediaPipelineFactory mpf = contentSession.getMediaPipelineFactory();
MediaPipeline mp = mpf.create();
contentSession.releaseOnTerminate(mp);
```

*MediaElements* not attached to the *content session* will remain active until an explicit release is performed.

```java
@Override
public void onContentRequest(WebRtcContentSession contentSession) throws Exception {
    MediaPipelineFactory mpf = contentSession.getMediaPipelineFactory();
    MediaPipeline mp = mpf.create();

    PlayerEndpoint player = mp.newPlayerEndpoint("file:///d").build();

    contentSession.start(player);
}
```

```java
@Override
public void onSessionTerminated(WebRtcContentSession contentSession, int code, String reason) throws Exception {
    player.release();
}
```

13.3.5.5  *Content Repository*

The Stream Oriented GE Java Content API provides a built-in *content repository* to store media streams (video and audio files). The elements stored in the repository (called *repository items*) can be accessed using the method *start* of the *content session* provided by the Java Content API.
The list of features implemented by the content repository are:

- Create repository items
- Set metadata in the repository items (key-value attributes)
- Find repository items (by its identifier, attribute value or regular expressions)
- Remove repository items

Let see a couple of examples to illustrate the way of working of the content repository. First, the following example shows how to use the content repository to store the stream from an HttpRecorderEndpoint:

```java
@HttpRecorderService(path = "/recorderRepository")
public class RecorderRepository extends HttpRecorderHandler {

    @Override
    public void onContentRequest(HttpRecorderSession contentSession)
        throws Exception {
        final String itemId = "itemTunnel";
        Repository repository = contentSession.getRepository();
        RepositoryItem repositoryItem;
        try {
            repositoryItem = repository.findRepositoryItemById(itemId);
            getLogger().info("Deleting existing repository '{}'", itemId);
            repositoryItem = repositoryItemByWithId(itemId);
            repositoryItem = contentSession.getRepository().createRepositoryItem(
```
This other example shows how to implement an `HttpPlayerHandler` to play a repository item identified by the `contentId` parameter:

```java
@HttpPlayerService(path = "/*/playerRepository/**")
public class PlayerRepository extends HttpPlayerHandler {

    @Override
    public void onContentRequest(HttpPlayerSession contentSession)
        throws Exception {
        String contentId = contentSession.getContentId();
        RepositoryItem repositoryItem = contentSession.getRepository() .findRepositoryItemById(contentId);
        if (repositoryItem == null) {
            String message = "Repository item " + contentId + " does not exist";
            getLogger().warn(message);
            contentSession.terminate(404, message);
        } else {
            contentSession.start(repositoryItem);
        }
    }
}
```
13.3.6 Programming with the Stream Oriented GE Java Media API

Kurento Media API is a low level Java SDK providing full control of Kurento Media Server. It is intended to be used at server side, in conjunction with Kurento Content API, although it can also be used on its own and even within standard Java projects, outside Kurento Application Server.

Following dependency has to be added to pom.xml in order to use Kurento Media API

```xml
<dependencies>
  ...
  <dependency>
    <groupId>com.kurento.kmf</groupId>
    <artifactId>kmf-media-api</artifactId>
    <version>4.2.2</version>
  </dependency>
  ...
</dependencies>
```

13.3.6.1 MediaPipeline

The MediaPipelineFactory is the API entry point. It can be obtained from the content session when used in conjunction with the Content API.

```java
@Override
public void onContentRequest(HttpPlayerSession contentSession) throws Exception {
  MediaPipelineFactory mpf = contentSession.getMediaPipelineFactory();
}
```

In order to use the Media API in stand-alone mode the application must setup a Spring framework context.

```java
public static void main(String[] args) {
  ApplicationContext context = new AnnotationConfigApplicationContext("classpath:kmf-media-config.xml");
```
MediaPipelineFactory mpf =
context.getBean(MediaPipelineFactory.class);
}

The Spring configuration file (kmf-media-config.xml in example above) must contain directive
<context:component-scan base-package="com.kurento.kmf.media" /> , so Media API
components can be found. Optionally a bean of class
com.kurento.kmf.media.MediaApiConfiguration can be added with custom configurations.

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.kurento.kmf.media" />

    <bean id="mediaApiConfiguration"
          class="com.kurento.kmf.media.MediaApiConfiguration">
        <property name="serverAddress" value="127.0.0.1" />
        <property name="serverPort" value="9090" />
        <property name="handlerAddress" value="127.0.0.1" />
        <property name="handlerPort" value="9191" />
    </bean>
</beans>
```

The MediaPipelineFactory can now be injected with any of the mechanism provided by Spring.

```java
public class MyApplication {
```
A `MediaPipeline` object is required to build media services. Method `create()` can be used in the `MediaPipelineFactory` for this purpose.

```java
public void init() {
    MediaPipeline mp = mpf.create();

    // Other initializations
}
```

`MediaPipelines` are the containers where KMS `MediaElements` live. `MediaElements` within a pipeline can be connected to build services, but they are isolated from the rest of the system. This has to be taken into account when programming applications.

As introduced before, currently there are two kinds of `MediaElements`, namely `Endpoints` and `Filters`.

### 13.3.6.2 Endpoints

KMS `MediaElements` are created through specific builders, allowing a flexible initialization. Mandatory parameters must be provided in the builder constructor, like the URL in the `PlayerEndpoint`. Optional parameters are set to defaults unless the application overrides their values.

```java
public void createMediaElements() {
    MediaPipeline mp = mpf.create();
    HttpGetEndpoint httpEndpoint = mp.newHttpEndpoint()
        .withDisconnectionTimeout(1000).
        .withGarbagePeriod(100)
        .withMediaProfile(MediaProfileSpecType.WEBM).build();

    PlayerEndpoint player = mp.newPlayerEndpoint("file:///myfile.avi")
```
RecorderEndpoint recorder = mp.newRecorderEndpoint("file:///myfile.mp4")
    .withMediaProfile(MediaProfileSpecType.MP4)
    .build();

RtpEndpoint rtp = mp.newRtpEndpoint()
    .build();

WebRtcEndpoint webrtc = mp.newWebRtcEndpoint()
    .build();

ZBarFilter zbar = mp.newZBarFilter().build();

// Do something with media elements

public void connectElements() {
    MediaPipeline mp = mpf.create();

    HttpGetEndpoint httpEndpoint = mp.newHttpGetEndpoint()
        .terminateOnEOS().build();

    PlayerEndpoint player = mp.newPlayerEndpoint("file:///myfile.avi")
        .build();

    player.connect(httpEndpoint);

*MediaElements* can be connected with method `connect()`. This method creates a directional connection between elements *source* and *sink* provided as parameters. All output streams of the *source* element are connected to the input streams of the *sink* element.
In order to create bidirectional connections the application must perform a connect operation in both directions.

```java
public void back2back () {
    MediaPipeline mp = mpf.create();

    RtpEndpoint rtpA = mp.newRtpEndpoint().build();
    RtpEndpoint rtpB = mp.newRtpEndpoint().build();

    rtpA.connect(rtpB);
    rtpB.connect(rtpA);
}
```

Notice that method `connect()` won't do anything when elements without input streams, like `PlayerEndpoint` are passed as sink or elements with no output streams, like `RecorderEndpoint`, are passed as source.

The Media API provides an asynchronous interface for those applications that cannot afford to block their calls until KMS responds. The asynchronous interface improves performance at a cost of increase in complexity.

```java
private MediaPipeline mp;

public void buildAsync () {
    mp = mpf.create();

    mp.newHttpGetEndpoint().buildAsync( new Continuation<HttpGetEndpoint>() {
        @Override
        public void onSuccess(HttpGetEndpoint result) {
            connectAsync (null, result);
        }
    });

    @Override
```
public void onError(Throwable cause) {
    // log error
}
});

mp.newPlayerEndpoint("file:///myfile.webm").buildAsync(new
Continuation<PlayerEndpoint>() {
    @Override
    public void onSuccess(PlayerEndpoint result) {
        connectAsync(result, null);
    }
    @Override
    public void onError(Throwable cause) {
        // log error
    }
});

private HttpGetEndpoint http;
private PlayerEndpoint player;

public void connectAsync(PlayerEndpoint player, HttpGetEndpoint http) {
    if (player != null) {
        this.player = player;
    }
    if (http != null) {
        this.http = http;
    }
    if (player != null && http != null){
13.3.6.2.1 **HttpGetEndpoint**

An **HttpGetEndpoint** contains source Media Pads for audio and video, delivering media using HTML5 pseudo-streaming mechanism. This type of endpoint provide unidirectional communications. Its Media *Sink* is associated with the HTTP GET method.

A Media Pad is an element's interface with the outside world. The data streams flow from the Media Source pad to another element's Media Sink pad.

13.3.6.2.2 **HttpPostEndpoint**

An **HttpPostEndpoint** contains sink pads for audio and video, which provide access to an HTTP file upload function. This type of endpoint provide unidirectional communications. Its Media Sources are accessed through the HTTP POST method.

13.3.6.2.3 **PlayerEndpoint**

A **PlayerEndpoint** retrieves content from seekable sources in reliable mode (does not discard media information) and inject them into KMS. It contains one Media Source for each media type detected.

13.3.6.2.4 **RecorderEndpoint**

A **RecorderEndpoint** provides function to store contents in reliable mode (doesn't discard data). It contains Media Sink pads for audio and video.

13.3.6.2.5 **RtpEndpoint**

A **RtpEndpoint** provides bidirectional content delivery capabilities with remote networked peers through RTP protocol. It contains paired sink and source Media Pads for audio and video.

13.3.6.2.6 **WebRtcEndpoint**

A **WebRtcEndpoint** provides media streaming for Real Time Communications (RTC) through the web.

**Filters**

Filters perform media processing, computer vision, augmented reality, and so on.

13.3.6.3.1 **JackVaderFilter**

**JackVaderFilter** detects faces in a video feed. Those on the right half of the feed are overlaid with a pirate hat, and those on the left half are covered by a Darth Vader helmet. This is an example filter, intended to demonstrate how to integrate computer vision capabilities into the KMS multimedia infrastructure.

```java
JackVaderFilter filter = mediaPipeline.newJackVaderFilter().build();
```
13.3.6.3.2  *ZBarFilter*
This filter detects QR and bar codes in a video feed. When a code is found, the filter raises a `CodeFoundEvent`. Clients can add a listener to this event using the method.

```java
ZBarFilter zBarFilter = mediaPipeline.newZBarFilter().build();
zBarFilter.addCodeFoundDataListener(new MediaEventListener<CodeFoundEvent>() {
   @Override
   public void onEvent(CodeFoundEvent event) {
      log.info("Code Found " + event.getValue());
      // ...
   }
});
```

13.3.6.3.3  *FaceOverlayFilter*
This type of filter detects faces in a video feed. The face is then overlaid with an image.

```java
MediaPipeline mp = session.getMediaPipelineFactory().create();
FaceOverlayFilter faceOverlayFilter = mp.newFaceOverlayFilter().build();
// xoffset%, y offset%, width%, height%
faceOverlayFilter.setOverlaidImage("/img/masks/mario-wings.png", -0.35F, -1.2F, 1.6F, 1.6F);
```

13.3.6.3.4  *PointerDetectorFilter and PointerDetectorAdvFilter*
These type of filters detects pointers in a video feed. The difference is in the way of calibration of such pointers.

```java
// start
PointerDetectorWindowMediaParam start = new PointerDetectorWindowMediaParamBuilder("start", 100, 100, 280, 380).withImage("/img/buttons/start.png").build();

// start
PointerDetectorAdvFilter pointerDetectorAdvFilter = mediaPipeline.newPointerDetectorAdvFilter(new WindowParam(5, 5, 50, 50))
```

D.6.4.3: FI-WARE Users and Programmers Guide 500
13.3.6.3.5  **GStreamerFilter**
This is a generic filter interface, that creates GStreamer filters in the media server.

```java
GStreamerFilter mirrorFilter = mediaPipeline.newGStreamerFilter("videoflip method=4")
                           .build();
```

13.3.6.3.6  **ChromaFilter**
This type of filter makes transparent a colour range in the top layer, revealing another image behind.

```java
ChromaFilter chromaFilter = mediaPipeline.newChromaFilter(
                                  new WindowParam(100, 10, 500, 400)).build();
```

13.3.6.3.7  **CrowdDetectorFilter**
This type of filter detects people agglomeration in video streams.

```java
List<Point> points = new ArrayList<Point>();
points.add(new Point(0, 0));
points.add(new Point(640, 0));
points.add(new Point(640, 480));
points.add(new Point(0, 480));
RegionOfInterestConfig config = new RegionOfInterestConfig();
config.setFluidityLevelMin(10);
config.setFluidityLevelMed(35);
config.setFluidityLevelMax(65);
config.setFluidityNumFramesToEvent(5);
config.setOccupancyLevelMin(10);
config.setOccupancyLevelMed(35);
config.setOccupancyLevelMax(65);
config.setOccupancyNumFramesToEvent(5);
config.setSendOpticalFlowEvent(false);
```
List<RegionOfInterest> rois = newArrayList(new RegionOfInterest(
    points, config, "Roi");
CrowdDetectorFilter crowdDetector = mp.newCrowdDetectorFilter(
    rois).build();
playerEndpoint.connect(crowdDetector);

13.3.6.3.8  PlateDetectorFilter
This filter detects vehicle plates in a video feed.

PlateDetectorFilter plateDetectorFilter = mp
    .newPlateDetectorFilter().build();
plateDetectorFilter
    .addPlateDetectedListener(new MediaEventListener<PlateDetectedEvent>() {
        @Override
        public void onEvent(PlateDetectedEvent event) {
            session.publishEvent(new ContentEvent(event
                .getType(), event.getPlate()));
        }
    });

13.3.6.4  Mixers
13.3.6.4.1  Dispatcher
A Dispatcher is a Hub that allows routing between arbitrary port pairs. The connections made by a
Dispatcher is through Media Elements called HubPort:

Dispatcher dispatcher = mp.newDispatcher().build();
HubPort hubPort1 = dispatcher.newHubPort().build();
HubPort hubPort2 = dispatcher.newHubPort().build();
endpoint1.connect(hubPort1);
endpoint2.connect(hubPort2);
dispatcher.connect(hubPort1, hubPort2);
13.3.6.4.2 **DispatcherOneToMany**

A *DispatcherOneToMany* is a *Hub* that sends a given source to all the connected sinks:

```java
Dispatcher dispatcherOneToMany = mediaPipeline.newDispatcherOneToMany().build();
HubPort hubPort1 = dispatcherOneToMany.newHubPort().build();
endpoint1.connect(hubPort1);
dispatcherOneToMany.setSource(hubPort1);
```

13.3.6.4.3 **Composite**

A *Composite* is a *Hub* that mixes the audio stream of its connected sources and constructs a grid with the video streams of its connected sources into its sink:

```java
Composite composite = mediaPipeline.newComposite().build();
HubPort hubPort = composite.newHubPort().build();
endpoint.connect(hubPort);
```

13.3.7 **Programming with the Stream Oriented GE HTML5 SDK**

The *Stream Oriented GE HTML5* SDK is a *Javascript* library implementing a *Content API* and a *Media API* client. The following sections provides details about these SDK libraries.

13.3.7.1 **KWS Content API**

It has been designed to be compatible with *node.js* infrastructure and all its dependencies have been included into the *Node Package Modules (NPM)*. For that reason it is required the *NPM* dependency management infrastructure to be installed.

```bash
sudo apt-get install npm
```

Current release of HTML5 SDK does not provide a library archive, so it must be built directly from the *source code*. A *bundle file* is also available at FI-WARE download page.

```bash
git clone https://github.com/Kurento/kws-content-api.git
cd kws-content-api
npm install
npm update
```
Grunt will place into directory dist four different Javascript bundles adapted to browser usage.

If you are developing your application with maven, simply add the Kurento Content Management API for Web SDK library (kws-content-api.js) as a regular dependency:

```xml
<dependencies>
  ...
  <dependency>
    <groupId>com.kurento.kmf</groupId>
    <artifactId>kws-content-api</artifactId>
    <version>4.2.2</version>
  </dependency>
  ...
</dependencies>
```

This way, kws-content-api.js will be available in your web application root, as follows:

```html
<html>
  <head>
    <script src="./js/kws-content-api.js"/>
  </head>
  <body>
    ...
  </body>
</html>
```

In order to use the Stream Oriented GE HTML5 SDK the Content API must activate the control protocol at handler level. Boolean attribute useControlProtocol is used for this purpose.

```java
@HttpPlayerService(path = "/myPlayerService" ,
  useControlProtocol=true)
public class MyPlayerService extends HttpPlayerHandler {
```
The Stream Oriented GE HTML5 SDK provides the following set of Content API clients:

- **KwsContentPlayer**: Allows connection with Kurento’s HTTP player handler in order to implement download services.
- **KwsContentUploader**: Intended to interoperate with the HTTP recorder handler. It allows implementing file upload services.
- **KwsWebRtcContent**: Helps applications to setup WebRTC connections with the WebRTC handler.

Clients above are intended to connect one Content API service. The constructor must provide the URL of the service entry point.

```javascript
<script>
function play(){
    var KwsContentPlayer = kwsContentApi.KwsContentPlayer;
    conn = new KwsContentPlayer("http://myServer/myApp/myPlayerService", options);
}
</script>
```

Optional parameters can be provided with configurations customized to the service.

- **audio**: Sets the audio stream mode. Can be any of `inactive`, `sendonly`, `recvonly` and `sendrecv`. Default value is `sendrecv`.
- **video**: Sets the video stream mode with the same alternatives available to audio. Default value is `sendrecv`.
- **localVideoTag**: ID of the `<video>` tag where local video will be displayed. No local video will be displayed if not defined.
- **remoteVideoTag**: ID of the `<video>` tag where remote video will be displayed. No remote video will be displayed if not defined.
- **iceServers**: STUN/TURN server array used by WebRTC ICE client. By default Google public STUN server is used.
Upon creation the client sends a start request to the server, causing the method `onContentRequest()` to be called in the service handler.

The same content session events received in the service handler are also available on the client side. Listeners are provided for this purpose.

```html

<html>
  <script>
    var uri = "http://www.example.com/jsonrpc";

    var options =
    {
      localVideoTag: 'localVideo',
      remoteVideoTag: 'remoteVideo'
    };

    var conn = new KwsWebRtcContent(uri, options);

    // Start and terminate events
    conn.on('start', function()
    {
      console.log("Connection started");
    });
    conn.on('terminate', function(reason)
    {
      console.log("Connection terminated due to "+reason.message);
    });

    // LocalStream and remoteStream events
    conn.on('localstream', function(data)
    {
```
console.info("LocalStream set to "+data.url);
});
conn.on('remotestream', function(data)
{
    console.info("RemoteStream set to "+data.url);
});

// Media event
conn.on('mediaevent', function(data)
{
    console.info("MediaEvent: "+JSON.stringify(data));
});

// Error
conn.on('error', function(error)
{
    console.error(error.message);
});
</script>
<body>
    <video id="localVideo"/>
    <video id="remoteVideo"/>
</body>
</html>

13.3.7.2 **KWS Media API**

KWS Media API provides the capabilities to create Media Pipelines and Media Elements in the KMS without a KAS. In other words, with KWS Media API we can create Kurento-based applications directly in JavaScript.

To describe this API, we are going to show how to create a basic pipeline that play a video file from its URL and stream it over HTTP. You can also download and check this [https://github.com/Kurento/kws-media-api/tree/develop/example/PlayerEndpointHttpGetEndpoint](https://github.com/Kurento/kws-media-api/tree/develop/example/PlayerEndpointHttpGetEndpoint) example full source code.
1. Create an instance of the KwsMedia class that will manage the connection with the Kurento Media Server, so you'll need to provide the URI of its WebSocket endpoint. Alternatively, instead of using a constructor, you can also provide success and error callbacks:

```javascript
var kwsMedia = kwsMediaApi.KwsMedia(ws_uri);

kwsMedia.onconnect = function(kwsMedia)
{
    ...
};

kwsMedia.onerror = function(error)
{
    ...
};

kwsMediaApi.KwsMedia(ws_uri, function(kwsMedia)
{
    ...
},
function(error)
{
    ...
});
```

22. Create a pipeline. This will host and connect the different elements. In case of error, it will be notified on the ```error``` parameter of the callback, otherwise this will be null as it's common on Node.js style APIs:

```javascript
kwsMedia.createMediaPipeline(function(error, pipeline)
{
    ...
});
```
28. Create the elements. The player need an object with the URL of the video, and we'll also subscribe to the 'EndOfStream' event of the HTTP stream:

```javascript
29. PlayerEndpoint.create(pipeline,
30. {uri: "https://ci.kurento.com/video/small.webm"},
31. function(error, player)
32. {
33. ...
34. });
35. 
36. 
37. HttpGetEndpoint.create(pipeline, function(error, httpGet)
38. {
39. httpGet.on('EndOfStream', function(event)
40. {
41. ...
42. });
43. 
44. ...
45. });
46. 
```

46. Connect the elements, so the media stream can flow between them:

```javascript
47. pipeline.connect(player, httpGet, function(error, pipeline)
48. {
49. ...
50. });
```

52. Get the URL where the media stream will be available:

```javascript
53. httpGet.getUrl(function(error, url)
54. {
55. ...
56. });
```
13.3.8 Examples

This section provides several examples of the Stream Oriented GE Kurento platform. To that aim we are going to use the Java Content and Media API in the server-side, and the JavaScript Content API in the client-side. The provided examples implement a MediaPipeline composed by a PlayerEndpoint connected to a Filter and generating a media flow through an HttpGetEndpoint. The main difference between these two examples is the filter. The first example uses the JackVaderFilter. This filter is an example of augmented reality element, since it recognizes faces in media streams adding Jack Sparrow or Darth Vader hat onto these faces. The second example uses the ZBarFilter. This filter is an example of computational vision element, since it recognize bar and QR codes in a media stream generating events with the information of the detected codes in the stream. Therefore, the MediaPipelines used in these examples are the following:

- PlayerEndpoint \(\rightarrow\) JackVaderFilter \(\rightarrow\) HttpGetEndpoint
- PlayerEndpoint \(\rightarrow\) ZBarFilter \(\rightarrow\) HttpGetEndpoint

For both examples, the handler (Java) and client (JavaScript) code is provided.

13.3.8.1 JackVaderFilter

The handler code (Java) for this example is shown in the snippet below. This handler is deployed in the KAS at the path \(http://myserver/myApp/playerJsonJackVader\). The PlayerEndpoint uses an URL to locate a media stream (\(https://ci.kurento.com/video/fiwarecut.webm\)) and then JackVaderFilter puts a pirate hat in the faces of this video.

```java
//This annotation configures the platform to deploy a handler on the specified path
@HttpPlayerService(path = " /playerJsonJackVader")
public class PlayerJsonJackVaderFilter extends HttpPlayerHandler {
  @Override
```
public void onContentRequest(HttpPlayerSession session) throws Exception {
    MediaPipelineFactory mpf = session.getMediaPipelineFactory();
    MediaPipeline mp = mpf.create();

    //This makes the pipeline (and all its elements) to be released when the session terminates
    session.releaseOnTerminate(mp);

    //Create a PlayerEndpoint for injecting a video into the platform
    PlayerEndpoint playerEndpoint = mp.newPlayerEndpoint("https://ci.kurento.com/video/fiwarecut.webm").build();

    //Create a filter for augmenting the video stream in real time.
    JackVaderFilter filter = mp.newJackVaderFilter().build();

    //Connect both elements
    playerEndpoint.connect(filter);

    //Store a player reference for later use
    session.setAttribute("player", playerEndpoint);

    //Create a HttpGetEndpoint and connects it to the filter
    HttpGetEndpoint httpEndpoint = mp.newHttpGetEndpoint().
        terminateOnEOS().build();
    filter.connect(httpEndpoint);

    //Start session
In order to perform a request to this handler, we create a simple HTML page in which the JavaScript Content API library (i.e. kws-content-api.js) is used. Depending on your development methodology, you may need to download that library to the appropriate directory. This HTML page must be included in the same WAR than the handler. Thus, in order to locate the handler path the JavaScript object document.URL is used:

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Stream Oriented GE Kurento</title>
<script src="/js/kws-content-api.js"></script>
<script>
    var conn;

    function start() {
        // Handler
```

```java
    @Override
    public void onContentStarted(HttpPlayerSession session) {
        //Content starts when the client connects to the HttpEndpoint
        //At that instant, the player must start reproducing the file
        PlayerEndpoint playerEndpoint = (PlayerEndpoint) session.getAttribute("player");
        playerEndpoint.play();
    }
```
```javascript
var handler = document.getElementById("handler").value;

// Options
var options = {
    remoteVideoTag: "remoteVideo"
};

// KwsContentPlayer instantiation
var KwsContentPlayer = kwsContentApi.KwsContentPlayer;
conn = new KwsContentPlayer(handler, options);

// Media events log
conn.on("mediaevent", function(data) {
    document.getElementById("events").value +=
        JSON.stringify(data) + "\n"
});

function stop() {
    conn.terminate();
}
</script>
</head>

<body>
    <h1>Stream Oriented GE Kurento Examples</h1>

    <label for="selectFilter">Handler</label>
    <select id="handler">
        <option value="./playerJsonJackVader">JackVaderFilter</option>
```
All in all, to run this example we have to make a request using a browser to the URL of this HTML page (e.g. http://myserver/myApp/mypage.html), select the JackVaderFi filter option and finally press the Start button. As a result, the stream played is the video located in the URL determined in the handler (https://ci.kurento.com/video/fiwarecut.webm) but showing the speaker of the video with a pirate hut in his head. Notice that this example is providing the media in WebM format, so it will only work on browsers supporting it (e.g. Chrome and Firefox).

13.3.8.2 ZBarFilter
The handler code (Java) for this example is shown below. This handler is deployed in the KAS at the path http://myserver/myApp/playerJsonZBar. The PlayerEndpoint uses an URL to locate a media stream (https://ci.kurento.com/video/barcodes.webm) and then ZBarFilter generates media events with the detected codes within the video.

```java
@HttpPlayerService(path = "/playerJsonZBar")
public class PlayerJsonZBarFilter extends HttpPlayerHandler {

    @Override
    public void onContentRequest(final HttpPlayerSession session) {
```
throws Exception {
    MediaPipelineFactory mpf = 
        session.getMediaPipelineFactory();
    MediaPipeline mp = mpf.create();
    PlayerEndpoint player = mp.newPlayerEndpoint(
        "https://ci.kurento.com/video/barcodes.webm").build();
    session.setAttribute("player", player);
    ZBarFilter zBarFilter = mp.newZBarFilter().build();
    player.connect(zBarFilter);
    HttpGetEndpoint httpEndpoint = mp.newHttpGetEndpoint()
        .terminateOnEOS().build();
    zBarFilter.connect(httpEndpoint);
    session.start(httpEndpoint);
    zBarFilter
        .addCodeFoundDataListener(new
        MediaEventListener<CodeFoundEvent>() {
            @Override
            public void onEvent(CodeFoundEvent
                event) {
                session.publishEvent(new
                ContentEvent(event.getType(),
                event.getValue()));
            }
        });
}

@Override
public void onContentStarted(HttpPlayerSession session) {
    PlayerEndpoint playerEndpoint = (PlayerEndpoint) session

To visualize the result of this handler, we use the same JavaScript code included in the previous example. This time, we select the ZBarFilter in the combo box and then press the Start button. As a result, the video containing QR codes is played (https://ci.kurento.com/video/barcodes.webm) and the detected codes by the filter are written in the HTML textarea with id events.

Both JackVaderFilter and ZBarFilter examples can be developed as a Maven project, and the resulting WAR is deployed in the KAS. An example of pom.xml for this Maven project in shown below.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.kurento.kmf</groupId>
  <artifactId>kmf-content-helloworld</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>

    <!-- Kurento Dependencies Versions -->
    <kmf-content-api.version>4.2.2</kmf-content-api.version>
  </properties>
</project>
```
<kws-content-api.version>4.2.2</kws-content-api.version>

<!-- Plugins Versions -->
<maven-war-plugin.version>2.3</maven-war-plugin.version>
</properties>

<dependencies>
<dependency>
<groupId>com.kurento.kmf</groupId>
<artifactId>kmf-content-api</artifactId>
<version>${kmf-content-api.version}</version>
</dependency>
<dependency>
<groupId>com.kurento.kmf</groupId>
<artifactId>kws-content-api</artifactId>
<version>${kws-content-api.version}</version>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-war-plugin</artifactId>
<version>${maven-war-plugin.version}</version>
</plugin>
</plugins>
</build>
</project>
The examples before and many others are available on GitHub:

```
    git clone https://github.com/Kurento/kurento-media-framework.git
```