IIoT

Context Broker APIs usage

Developer's Guide

July 24th, 2014

Telefónica

CONFIDENTIAL

© Telefónica, S.A.

Not to be copied, distributed or referred to without the prior written consent of Telefónica. All rights reserved.

The information contained herein and all related data or information are proprietary and confidential to Telefónica and are provided in confidence. All use, disclosure, copying, transfer, reproduction, display, distribution, or creation of derivative works, unless expressly authorized in the written by Telefónica, is strictly prohibited.

Information in this document is subject to change without notice and does not represent a commitment on the part of Telefónica.

Contents

Signature control table

Edited by	Reviewed by	Approved by
Fermín Galán	Elena Cruz	

Version	Changed Sections	Date	Description	Author
0.1	-	09/07/2014	First draft	Fermín Galán
0.2	-	17/07/2014	Completing API operations examples Completing references Added Annex A Adding Fiware-Service header in curl examples	Fermín Galán
0.3	all	21/07/2014	Integrating proof-read suggestions from Mª Elena Cruz	Fermín Galán
1.0	all	22/07/2014	General review and cleaning	Elena Cruz
1.1	4.12.2, 5	24/07/2014	Use right distances in geo-query examples	Fermín Galán

Table of illustrations

1 Introduction

The Orion Context Broker is an implementation of a publish-subscribe Context Broker based on the OMA NGSI specification [1]. Using its REST interfaces, clients can do several operations:

- Create context elements, representing entities in the system, e.g. a given sensor which measures temperature
- Update context information, e.g. send temperature updates
- Being notified when changes on context information take place (e.g. the temperature has changed)
- Query context information. The Orion Context Broker stores context information updated from applications, so queries are resolved based on that information.

The goal of this guide is to describe the API exposed by Orion Context Broker.

2 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 to ease the usage of NGSI implementations.

2.1 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 Section 2.3).

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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.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.

Note: 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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
```

```
  </contextResponseList>
</updateContextResponse>
```

JSON

```
{
    "contextResponses": [
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": ""
                    },
                    {
                        "name": "pressure",
                        "type": "mmHg",
                        "value": ""
                    }
                ],
                "id": "Room1",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        }
    ]
}
```

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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>21</contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <contextValue>711</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>APPEND</updateAction>
```

```
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool ) <<EOF
{
    "contextElements": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room2",
            "attributes": [
            {
                "name": "temperature",
                "type": "centigrade",
                "value": "21"
            },
            {
                "name": "pressure",
                "type": "mmHg",
                "value": "711"
            }
            ]
        }
    ],
    "updateAction": "APPEND"
}
EOF
```

The response to this request is:

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>
          <contextAttribute>
            <name>pressure</name>
            <type>mmHg</type>
            <contextValue/>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```

JSON

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

```
                        "value": ""
                    },
                    {
                        "name": "pressure",
                        "type": "mmHg",
                        "value": ""
                    }
                ],
                "id": "Room2",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        }
    ]
}
```

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 Section 4.11and Section 4.2, respectively.

2.2 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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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

```
<?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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```

JSON

```
{
    "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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

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

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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```

JSON

```
{
    "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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
    "entities": [
    {
        "type": "Room",
        "isPattern": "false",
        "id": "Room1"
    },
    {
        "type": "Room",
        "isPattern": "false",
        "id": "Room2"
    }
    ],
    "attributes" : [
        "temperature"
    ]
}
EOF
```

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -d @- | xmllint --format -) <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
  <entityIdList>
    <entityId type="Room" isPattern="true">
      <id>Room.*</id>
    </entityId>
  </entityIdList>
  <attributeList>
```

```
    <attribute>temperature</attribute>
  </attributeList>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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>
      <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>
            <type>centigrade</type>
            <contextValue>21</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```

JSON

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

```
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": "23"
                    }
                ],
                "id": "Room1",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        },
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": "21"
                    }
                ],
                "id": "Room2",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        }
    ]
}
```

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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

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

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

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

Both requests will produce the same error response:

XML

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

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 Section 4.12.
- 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 behavior, see Section 4.1 in this manual.

2.3 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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
        </contextAttribute>
        <contextAttribute>
          <name>pressure</name>
          <type>mmHg</type>
          <contextValue>763</contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>UPDATE</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
    "contextElements": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room1",
            "attributes": [
            {
                "name": "temperature",
                "type": "centigrade",
                "value": "26.5"
            },
            {
                "name": "pressure",
                "type": "mmHg",
                "value": "763"
            }
            ]
        }
    ],
```

```
    "updateAction": "UPDATE"
}
EOF
```

As you can see, the structure of the request is exactly the same we used for updateContext with APPEND for creating entities (see Section 2.1), 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>
            <contextValue/>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```

JSON

```
{
    "contextResponses": [
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": ""
                    },
                    {
                        "name": "pressure",
                        "type": "mmHg",
                        "value": ""
                    }
                ],
                "id": "Room1",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        }
    ]
```

```
}
```

Again, the structure of the response is exactly the same one we used for updateContext with APPEND for creating entities (see Section 2.1).

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 ºC 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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
    "contextElements": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room2",
            "attributes": [
            {
                "name": "temperature",
                "type": "centigrade",
                "value": "27.4"
            }
            ]
        }
    ],
    "updateAction": "UPDATE"
}
EOF
```

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
    "contextElements": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room2",
            "attributes": [
            {
                "name" : "pressure",
                "type" : "mmHg",
                "value" : "755"
            }
            ]
        }
    ],
    "updateAction": "UPDATE"
}
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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
```

```
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</updateContextResponse>
```

JSON

```
{
    "contextResponses": [
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "temperature",
                        "type": "centigrade",
                        "value": ""
                    }
                ],
                "id": "Room2",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
            }
        }
    ]
}
```

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

```
{
    "contextResponses": [
        {
            "contextElement": {
                "attributes": [
                    {
                        "name": "pressure",
                        "type": "mmHg",
                        "value": ""
                    }
                ],
                "id": "Room2",
                "isPattern": "false",
                "type": "Room"
            },
            "statusCode": {
                "code": "200",
                "reasonPhrase": "OK"
```

```
            }
        }
    ]
}
```

Now, you can use queryContext operation as previously described in Section 2.2 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 or custom metadata. These are advance topics, described in Section 4.11and Section 4.2, respectively.

2.4 Context subscription

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.

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 [cb_host]:[cb_port]/NGSI10/subscribeContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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://[callback_host]:[callback_port]/accumulate</reference>
  <duration>P1M</duration>
  <notifyConditions>
    <notifyCondition>
      <type>ONCHANGE</type>
      <condValueList>
        <condValue>pressure</condValue>
      </condValueList>
    </notifyCondition>
  </notifyConditions>
  <throttling>PT5S</throttling>
</subscribeContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/subscribeContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
```

```
{
    "entities": [
        {
            "type": "Room",
            "isPattern": "false",
            "id": "Room1"
        }
    ],
    "attributes": [
        "temperature"
    ],
    "reference": "http://[callback_host]:[callback_port]/accumulate",
    "duration": "P1M",
    "notifyConditions": [
        {
            "type": "ONCHANGE",
            "condValues": [
                "pressure"
            ]
        }
    ],
    "throttling": "PT5S"
}
EOF
```

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 in the same way as the XML elements, with the same name in queryContext request (Section 2.2). 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. 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 "[callback_host]:[callback_port]" as reference will be actually interpreted as "http://[callback_host]:[callback_port]".
- Subscriptions have a duration, specified using the ISO 8601 [2] standard format. Once that the specified duration has expired, the subscription is simply ignored. You can extend the duration of a subscription by updating it, as described in Section 4.10. We are using "P1M" which means "one month".
- The notifyCondition element uses the type ONCHANGE and 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 (Section 3.1) or update (Section 3.3) trigger the notification. The rule is that if at least one of the attributes in the list changes, then a notification is sent (i.e. e. some kind of "OR" condition). But note that the 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 to be notified of the value of temperature each time the value of pressure changes may not be very 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 have taken place in that period. This is to not stress the notification receptor in case of having context producers that update attribute values too often.

The response consists of a subscription ID (a 24 hexadecimal number used for updating and cancelling the subscription), 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>
```

JSON

```
{
    "subscribeResponse": {
        "duration": "P1M",
        "subscriptionId": "51c0ac9ed714fb3b37d7d5a8",
        "throttling": "PT5S"
    }
}
```

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 (Section 2.2).

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

Orion Context Broker will send one (and just one by the moment, no matter how much you wait) notifyContextRequest, similar to this one:

XML

```
POST http://[callback_host]:[callback_port]/accumulate
Content-Length: 739
User-Agent: orion/0.9.0
Host: [callback_host]:[callback_port]
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>
        <contextAttributeList>
          <contextAttribute>
            <name>temperature</name>
            <type>centigrade</type>
            <contextValue>26.5</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</notifyContextRequest>
```

JSON

```
POST http://[callback_host]:[callback_port]/accumulate
Content-Length: 492
User-Agent: orion/0.9.0
Host: [callback_host]:[callback_port]
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 Orion Context Broker is sending 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.

Note: 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.

Now, do the following exercise, based on what you know from update context (Section 2.3): 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.

Subscriptions can be updated using the NGSI10 updateContextSubcription. The request includes a subscriptionId that identifies the subscription to modify and the actual update payload. For example, if we want to change the throttling interval to 10 seconds we will use the following (of course, replace the subscriptionId value after the one that you have got in the subscribeContext response in the previous step) command:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContextSubscription -s -S --header 'Content-
Type: application/xml' --header 'Fiware-Service: MyService' -d @- | xmllint --format -
) <<EOF
<?xml version="1.0"?>
<updateContextSubscriptionRequest>
  <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
  <throttling>PT10S</throttling>
</updateContextSubscriptionRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContextSubscription -s -S --header 'Content-
Type: application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
    "subscriptionId": "51c0ac9ed714fb3b37d7d5a8",
    "throttling": "PT10S"
}
EOF
```

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

XML

```
<?xml version="1.0"?>
<updateContextSubscriptionResponse>
  <subscribeResponse>
    <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
    <throttling>PT10S</throttling>
  </subscribeResponse>
</updateContextSubscriptionResponse>
```

JSON

```
{
  "subscribeResponse" : {
    "subscriptionId" : "51c04a21d714fb3b37d7d5a7",
    "throttling": "PT10S"
  }
}
```

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

```
(curl [cb_host]:[cb_port]/NGSI10/unsubscribeContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -d @- | xmllint --format - )
<<EOF
<?xml version="1.0"?>
<unsubscribeContextRequest>
  <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
</unsubscribeContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/unsubscribeContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "subscriptionId": "51c0ac9ed714fb3b37d7d5a8"
}
EOF
```

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

XML

```
<?xml version="1.0"?>
<unsubscribeContextResponse>
  <subscriptionId>51c0ac9ed714fb3b37d7d5a8</subscriptionId>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
</unsubscribeContextResponse>
```

JSON

```
{
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    },
    "subscriptionId": "51c0ac9ed714fb3b37d7d5a8"
}
```

2.5 Summary of 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

3 Convenience Operations

This section describes the different convenience operations that Orion Context Broker supports, showing examples of requests and responses. Convenience operations are a set of operations that have been defined to ease the usage of NGSI implementations as a complement to the standard operations defined in the OMA NGSI specification.

3.1 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. Thus, let's first create Room1 entity with temperature and pressure attributes (with its initial values)

XML

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room1 -s -S --header 'Content-Type:
application/xml' -X POST --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room1 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.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>
          <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

```
{
  "contextResponses": [
    {
      "attributes": [
        {
          "name": "temperature",
          "type": "centigrade",
          "value": ""
        },
        {
          "name": "pressure",
          "type": "mmHg",
          "value": ""
        }
      ],
      "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
      }
    }
  ]
}
```

Now, let's do the same with Room2:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room2 -s -S --header 'Content-Type:
application/xml' -X POST --header 'Fiware-Service: MyService' -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>
      <contextValue>711</contextValue>
    </contextAttribute>
  </contextAttributeList>
</appendContextElementRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room2 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextAttributeResponse>
  </contextResponseList>
</appendContextElementResponse>
```

JSON

```
{
  "contextResponses": [
    {
      "attributes": [
        {
          "name": "temperature",
          "type": "centigrade",
          "value": ""
        },
        {
          "name": "pressure",
          "type": "mmHg",
          "value": ""
        }
      ],
      "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
      }
    }
  ]
}
```

Comparing this to entity creation based on standard operation (Section 2.1) 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 requests and responses for convenience operations are very similar to the ones used in standard operations, being the contextAttributeList and contextResponseList elements 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 (see Section 4.9).

Apart from simple values (i.e. strings) for attribute values, you can also use complex structures or custom metadata. These are advanced topics, described in Section 4.11and Section 4.2, respectively.

3.2 Query context operation

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

XML

```
curl [cb_host]:[cb_port]/ngsi10/contextEntities/Room1 -s -S --header 'Content-Type:
application/xml' | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/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>
      <contextAttribute>
        <name>pressure</name>
        <type>mmHg</type>
        <contextValue>720</contextValue>
      </contextAttribute>
    </contextAttributeList>
  </contextElement>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
```

```
</contextElementResponse>
```

JSON

```
{
  "contextElement": {
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": "23"
    },
    {
      "name": "pressure",
      "type": "mmHg",
      "value": "720"
    }
    ],
    "id": "Room1",
    "isPattern": "false",
    "type": ""
  },
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  }
}
```

We can also query a single attribute of a given entity, e.g. Room2 temperature:

XML

```
curl
[cb_host]:[cb_port]/ngsi10/contextEntities/Room2/attributes/temperatur
e -s -S --header 'Content-Type: application/xml' | xmllint --format -
```

JSON

```
curl
[cb_host]:[cb_port]/NGSI10/contextEntities/Room2/attributes/temperatur
e -s -S --header 'Content-Type: application/json' --header 'Accept:
application/json' | python -mjson.tool
```

which response is:

XML

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

JSON

```
{
  "attributes": [
  {
    "name": "temperature",
    "type": "centigrade",
    "value": "21"
  }
  ],
```

```
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  }
}
```

Comparing to standard queryContext operation (Section 2.2) 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 the 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 or lists of attributes.

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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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"
      }
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
```

JSON

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

```
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

Request to get all the attributes:

XML

```
curl [cb_host]:[cb_port]/ngsi10/contextEntityTypes/Car -s -S --header
'Content-Type: application/xml' | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/NGSI10/contextEntityTypes/Car -s -S --header
'Content-Type: application/json' --header 'Accept: application/json' |
python -mjson.tool
```

Response:

XML

```
<?xml version="1.0"?>
<?xml version="1.0"?>
<queryContextResponse>
  <contextResponseList>
    <contextElementResponse>
      <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>
      <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>
          <contextAttribute>
            <name>fuel</name>
            <type>liter</type>
            <contextValue>25.7</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
```

```
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```

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"
    }
  }
  ]
}
```

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

XML

```
curl [cb_host]:[cb_port]/ngsi10/contextEntityTypes/Car/attributes/speed -s -S --header
'Content-Type: application/xml' | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/NGSI10/contextEntityTypes/Car/attributes/speed -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>
            <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>
</queryContextResponse>
```

JSON

```
{
  "contextResponses": [
  {
    "contextElement": {
      "attributes": [
      {
        "name": "speed",
        "type": "km/h",
        "value": "75"
      }
      ],
      "id": "Car1",
      "isPattern": "false",
      "type": "Car"
    },
    "statusCode": {
      "code": "200",
      "reasonPhrase": "OK"
    }
  },
  {
    "contextElement": {
      "attributes": [
      {
        "name": "speed",
        "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 advanced topic, described in Section 4.12.
- Note that by default only 20 entities are returned (which is fine for this example, but probably not for a real utilization scenario). In order to change this behavior, see Section 4.1 in this manual.

3.3 Update context elements

Let's set the Room1 temperature and pressure values:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room1/attributes -s -S --header
'Content-Type: application/xml' -X PUT --header 'Fiware-Service: MyService' -d @- |
xmllint --format - ) << EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextElementRequest>
  <contextAttributeList>
    <contextAttribute>
      <name>temperature</name>
      <type>centigrade</type>
      <contextValue>26.5</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>pressure</name>
      <type>mmHg</name>
      <contextValue>763</contextValue>
    </contextAttribute>
  </contextAttributeList>
</updateContextElementRequest>
EOF
```

JSON

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

the 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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextAttributeResponse>
  </contextResponseList>
</updateContextElementResponse>
```

JSON

```
{
  "contextResponses": [
  {
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": ""
    },
    {
      "name": "pressure",
      "type": "mmHg",
      "value": ""
    }
    ],
    "statusCode": {
      "code": "200",
      "reasonPhrase": "OK"
    }
  }
  ]
}
```

Now, let's do the same with Room2:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room2/attributes -s -S --header
'Content-Type: application/xml' -X PUT --header 'Fiware-Service: MyService' -d @- |
xmllint --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

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/Room2/attributes -s -S --header
'Content-Type: application/json' --header 'Accept: application/json' -X PUT --header
'Fiware-Service: MyService' -d @- | python -mjson.tool) << EOF
{
  "attributes" : [
  {
    "name" : "temperature",
    "type" : "centigrade",
    "value" : "27.4"
  },
  {
    "name" : "pressure",
    "type" : "mmHg",
    "value" : "755"
  }
  ]
}
EOF
```

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>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextAttributeResponse>
  </contextResponseList>
</updateContextElementResponse>
```

JSON

```
{
  "contextResponses": [
  {
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": ""
    },
    {
      "name": "pressure",
      "type": "mmHg",
      "value": ""
    }
    ],
    "statusCode": {
      "code": "200",
      "reasonPhrase": "OK"
```

```
        }
    }
    ]
}
```

You can update a single attribute of a given entity.

Comparing to standard updateContext operation (Section 2.3) 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 Section 4.11and Section 4.2, respectively.

3.4 Context subscription

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

- POST /ngsi10/contextSubscriptions, to create the subscription, using the same payload as standard susbcribeContext operation (see Section 2.4).
- PUT /ngsi10/contextSubscriptions/{subscriptionID}, to update the subscription identified by {subscriptionID}, using the same payload as standard updateContextSubscription operation (see Section 2.4). 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

3.5 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 will find a summary in Annex A.

4 Advanced Topics

4.1 Pagination

In order to help clients to organize query and discovery requests with a large number of responses (for example, think on how costly it could be to 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, maximun 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 the count of total elements (in the case of using "on"). Note that using details set to "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 ordered by increasing entity/registration creation time, to ensure that if a new entity/registration is 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 this 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 code (using '...' to hide those elements of the payload that are irrelevant here) 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
subsequence queries to do)

  <errorCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
    <details>Count: 322</details>
  </errorCode>

POST <orion_host>:1026/NGSI10/queryContext?offset=100&limit=100
...
(Entities from 101 to 200)

POST <orion_host>:1026/NGSI10/queryContext?offset=200&limit=100
...
```

```
(Entities from 201 to 300)

POST <orion_host>:1026/NGSI10/queryContext?offset=300&limit=100
...
(Entities from 301 to 222)
```

Note that the if request uses an “out of bound” offset value you will get a 404 NGSI error, as shown below:

```
POST <orion_host>:1026/NGSI10/queryContext?offset=1000&limit=100
...
<queryContextResponse>
  <errorCode>
    <code>404</code>
    <reasonPhrase>No context element found</reasonPhrase>
    <details>Number of matching entities: 5. Offset is 100</details>
  </errorCode>
</queryContextResponse>
```

4.2 Custom attribute metadata

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 retrieves it in queryContext or notifyContext.

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

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header ‘Fiware-Service: MyService’ -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>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header ‘Fiware-Service:
MyService’ -d @- | python -mjson.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"
      }
      ]
    }
    ]
  }
  ],
  "updateAction": "APPEND"
}
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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
  <updateAction>UPDATE</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.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"
      }
      ]
    }
    ]
  }
  ],
  "updateAction": "UPDATE"
}
EOF
```

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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
  {
    "type": "Room",
    "isPattern": "false",
    "id": "Room1",
    "attributes": [
    {
      "name": "temperature",
      "type": "centigrade",
      "value": "26.5",
      "metadatas": [
      {
```

```
        "name": "average",
        "type": "centigrade",
        "value": "22.4"
      }
      ]
    }
    ]
  }
  ],
  "updateAction": "UPDATE"
}
EOF
```

We can check that temperature includes both attributes

XML

```
curl [cb_host]:[cb_port]/ngsi9/contextEntities/Room1 -s -S | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/NGSI9/contextEntities/Room1 -s -S --header 'Accept:
application/json' | python -mjson.tool
```

XML

```
<?xml version="1.0"?>
<contextElementResponse>
  <contextElement>
    <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"
      }
      ]
    }
    ]
  }
  ],
  "statusCode": {
    "code": "200",
    "reasonPhrase": "OK"
  }
}
```

Note that, from the ONCHANGE subscription point of view (Section 2.4), changing the metadata of a given attribute or adding a new metadata element is considered a change even if the attribute value itself hasn't changed. Metadata elements cannot be deleted once they have been introduced. In order to delete metadata elements you have to remove the entity attribute (using updateContext DELETE, see Section 4.6), then re-create it (using updateContext APPEND, see Section 4.6).

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

- ID (described in Section 4.8)
- location (described in Section 4.12)
- creData (reserved for future use)
- modDate (reserved for future use)

4.3 Default subscription duration

If you don't specify a duration in subscribeContext a default of 24 hours is used. You will get a confirmation of the duration in these cases in the response.

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 (Section 2.5) and the set of URLs used by convenience operations (Section 3.5) 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 (see Section 4.9).

4.5 Updating subscriptions

You have previously seen in this document that context subscriptions (Section 2.4) can be updated. However, not everything can be updated.

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

4.6 *Adding and removing context elements using APPEND and DELETE updateContext*

We have seen how to use updateContext with APPEND action type to create new entities (see Section 2.1). 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':

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
```

```
    {
      "type": "T",
      "isPattern": "false",
      "id": "E1",
      "attributes": [
      {
        "name": "A",
        "type": "TA",
        "value": "1"
      }
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

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

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
    {
      "type": "T",
      "isPattern": "false",
      "id": "E1",
      "attributes": [
      {
        "name": "B",
        "type": "TB",
        "value": "2"
      }
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

Now we can check with a query to that entity that both attributes A and B are there:

XML

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/xml' | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' | python -mjson.tool
```

XML

```
<?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>
  </contextElement>
  <statusCode>
    <code>200</code>
    <reasonPhrase>OK</reasonPhrase>
  </statusCode>
</contextElementResponse>
```

JSON

```
{
    "contextElement": {
        "attributes": [
            {
                "name": "B",
                "type": "TB",
                "value": "2"
            },
            {
                "name": "A",
                "type": "TA",
                "value": "1"
            }
        ],
        "id": "E1",
        "isPattern": "false",
        "type": ""
    },
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    }
}
```

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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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/>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>DELETE</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "contextElements": [
    {
      "type": "T",
      "isPattern": "false",
      "id": "E1",
      "attributes": [
      {
        "name": "A",
        "type": "TA",
        "value": ""
      }
      ]
    }
  ],
  "updateAction": "DELETE"
}
EOF
```

Now, a query to the entity shows attribute B:

XML

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/xml' | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' | python -mjson.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

```
{
    "contextElement": {
        "attributes": [
            {
                "name": "B",
                "type": "TB",
                "value": "2"
            }
        ],
        "id": "E1",
        "isPattern": "false",
        "type": ""
    },
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    }
}
```

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 [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/xml' -X POST --header 'Fiware-Service: MyService' -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>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "attributes" : [
    {
      "name" : "C",
      "type" : "TC",
      "value" : "3"
    },
    {
      "name" : "D",
      "type" : "TD",
      "value" : "4"
    }
  ]
}
EOF
```

Remove attribute 'B':

XML

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1/attributes/B -s -S --header
'Content-Type: application/xml' -X DELETE | xmllint --format -
```

JSON

```
curl [cb_host]:[cb_port]/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

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/xml' | xmllint --format -
```

JSON

```
 /contextEntities/E1 -s curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1
-S --header 'Content-  -s -S --header 'Content-Type: application/json' --
Type: application/xml' header 'Accept: application/json' | python -
| xmllint --format -   mjson.tool
```

XML

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

```
{
    "contextElement": {
        "attributes": [
            {
                "name": "C",
                "type": "TC",
                "value": "3"
            },
            {
                "name": "D",
                "type": "TD",
                "value": "4"
            }
        ],
        "id": "E1",
        "isPattern": "false",
        "type": ""
    },
```

```
    "statusCode": {
        "code": "200",
        "reasonPhrase": "OK"
    }
}
```

4.7 Delete entities

Apart from deleting individual attributes from a given entity (see Section 4.6), 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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -d @- | xmllint --format - )
<<EOF
<?xml version="1.0" encoding="UTF-8"?>
<updateContextRequest>
  <contextElementList>
    <contextElement>
      <entityId type="T" isPattern="false">
        <id>E1</id>
      </entityId>
      <contextAttributeList/>
    </contextElement>
  </contextElementList>
  <updateAction>DELETE</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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 [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/xml' -X DELETE
```

JSON

```
curl [cb_host]:[cb_port]/NGSI10/contextEntities/E1 -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' -X DELETE
```

4.8 Metadata ID for attributes

Sometimes, you could want to model attributes belonging to an entity which share the same name and type. For example, let's consider an entity "Room1" which has two temperature sensors: one in the ground and other in the wall. We can model this as two instances of the attribute "temperature", one with ID "ground" and the other

with the ID "wall". We use the metadata ID for this purpose. Let's illustrate with an example.

First, we create the Room1 entity:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>23.5</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>
          <metadata>
             <contextMetadata>
                <name>ID</name>
                <type>string</type>
                <value>wall</value>
             </contextMetadata>
          </metadata>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>APPEND</updateAction>
</updateContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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"
          }
          ]
        }
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

Now, we can query for temperature to get both instances:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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

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

We can update an specific instance (e.g. ground), letting the other untouched:

XML

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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"
        }
      ]
    }
  ],
  "updateAction": "APPEND"
}
EOF
```

XML

```
...
 <statusCode>
   <code>472</code>
   <reasonPhrase>request parameter is invalid/not
allowed</reasonPhrase>
   <details>action: APPEND - entity: (Room1, Room) - offending
attribute: temperature</details>
 </statusCode>
 ...
```

JSON

```
...
"statusCode": {
  "code": "472",
  "details": "action: APPEND - entity: (Room1, Room) - offending
attribute: temperature",
  "reasonPhrase": "request parameter is invalid/not allowed"
}
...
```

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 Section 3.3).
- DELETE /ngsi10/contextEntities/Room1/attributes/temperature/ground: to remove an specific attribute identified by ID (see DELETE attribute semantics described in Section 4.6).

4.9 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 queryContext using:

```
...
<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 behavior. For example, in the above case of temperature-centigrade and temperature-Fahrenheit (for Room8), consider the following updateContext request with empty type for temperature:

```
(curl [cb_host]:[cb_port]/ngsi10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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:

```
(curl [cb_host]:[cb_port]/ngsi10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
EOF

<?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>Fahrenheit</type>
            <contextValue>30</contextValue>
          </contextAttribute>
        </contextAttributeList>
      </contextElement>
      <statusCode>
        <code>200</code>
        <reasonPhrase>OK</reasonPhrase>
      </statusCode>
    </contextElementResponse>
  </contextResponseList>
</queryContextResponse>
```

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.

4.10 Extending subscription duration

We have seen that subscriptions have a duration. The expiration date is calculated using the following formula:

```
expiration = current-time + duration
```

The behavior of the broker regarding expired subscriptions is the following: an expired subscription is not taken into account to send new notifications based on it, but it is still updatable (using updateContextSubscription) and it can be canceled (using unsubscribeContext).

Finally, take into account that the expiration is recalculated on updates, not 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.

4.11 Structured attribute values

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 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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
 {
   "contextElements": [
       {
           "type": "T1",
           "isPattern": "false",
           "id": "E1",
           "attributes": [
           {
               "name": "A",
               "type": "T",
               "value": [ "22" ,
                          {
                             "x": [ "x1", "x2"],
                             "y": "3"
                          },
                          [ "z1", "z2" ]
                        ]
           },
           {
               "name": "B",
               "type": "T",
               "value": {
                  "x": {
                           "x1": "a",
                           "x2": "b"
                  },
                  "y": [ "y1", "y2" ]
               }
           }
           ]
       }
   ],
   "updateAction": "UPDATE"
 }
EOF
```

In the case of XML, the structure is a bit less "natural" than in JSON, but equivalent:

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
            <item>
              <x type="vector">
                <item>x1</item>
                <item>x2</item>
              </x>
              <y>3</y>
            </item>
            <item type="vector">
              <item>z1</item>
              <item>z2</item>
```

```
            </item>
          </contextValue>
        </contextAttribute>
        <contextAttribute>
          <name>B</name>
          <type>T</type>
          <contextValue>
             <x>
               <x1>a</x1>
               <x2>b</x2>
             </x>
             <y type="vector">
                <item>y1</item>
                <item>y2</item>
             </y>
          </contextValue>
        </contextAttribute>
      </contextAttributeList>
    </contextElement>
  </contextElementList>
  <updateAction>UPDATE</updateAction>
</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 a 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 made 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" }).

4.12 Geo-location capabilities

Orion Context Broker has several capabilities related to geolocation that are described in this section.

4.12.1 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

```
(curl [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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 [cb_host]:[cb_port]/NGSI10/updateContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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 information about adding and removing attributes dynamically in Section 4.6).
- The value of the location metadata is the coordinates system used. Current version only support WSG84 [3] 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").

4.12.2 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.

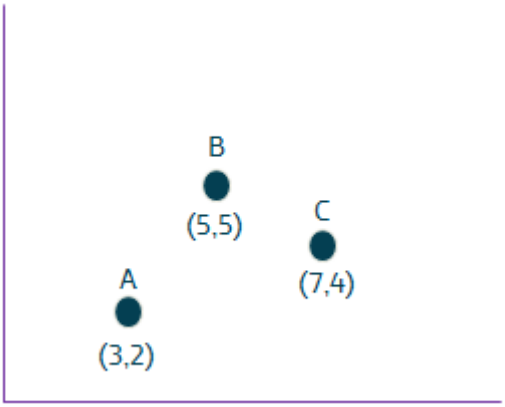

Illustration 1. Sample entity locations

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).

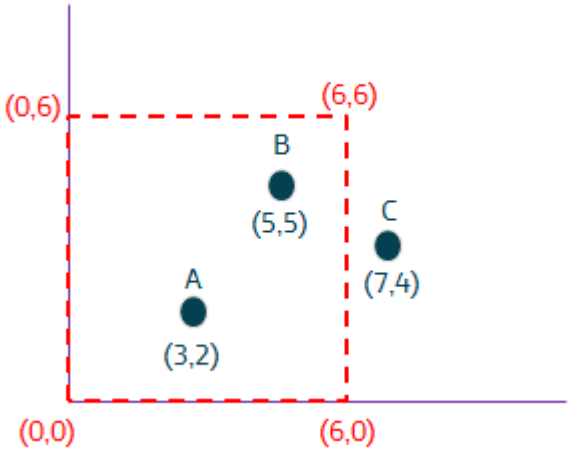

Illustration 2. Sample area (case 1)

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.

TABLE

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).

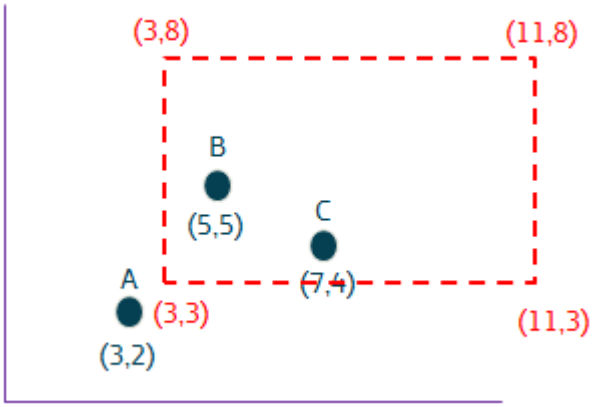

Illustration 3. Sample area (case 2)

The result of the query would be B and C.

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>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>
          </polygon>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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"
          }
          ]
        }
      }
    }
    ]
  }
}
EOF
```

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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>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>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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"
          }
          ],
          "inverted": "true"
        }
      }
    }
    ]
  }
}
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).

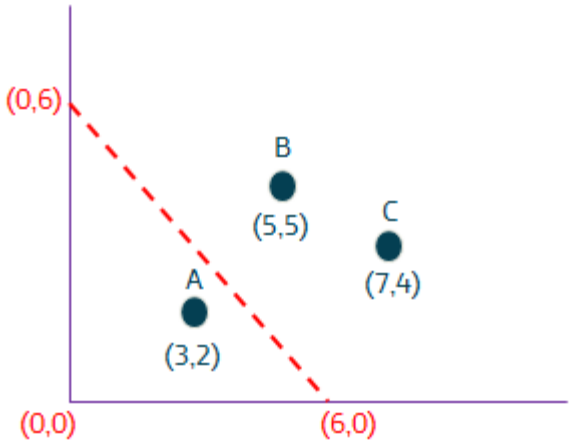

Illustration 4. Sample area (case 3)

The result of the query would be A.

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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": "6"
          },
          {
            "latitude": "6",
            "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 [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>
            <inverted>true</inverted>
          </polygon>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
```

```
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -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": "6"
          },
          {
            "latitude": "6",
            "longitude": "0"
          }
          ],
          "inverted": "true"
        }
      }
    }
    ]
  }
}
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 13.65 km, and the distance between Madrid and Leganes is around 12.38 km. You can calculate distances using this tool.

Let's consider a query whose scope is inside a radius of 13.5 km (13500 meters) centred in Madrid.

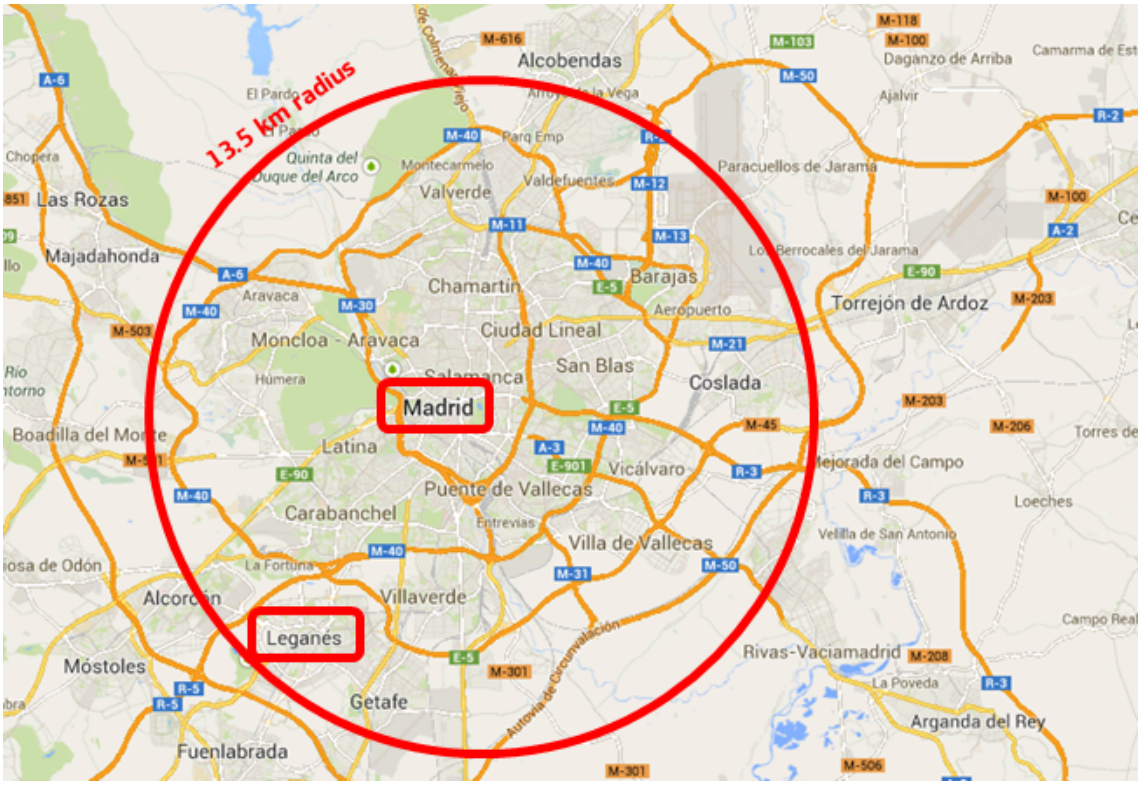

Illustration 5. Sample city area (13.5 km radius)

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

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>13500</radius>
          </circle>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
```

```
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
  {
    "type": "City",
    "isPattern": "true",
    "id": ".*"
  }
  ],
  "restriction": {
    "scopes": [
    {
      "type" : "FIWARE_Location",
      "value" : {
        "circle": {
          "centerLatitude": "40.418889",
          "centerLongitude": "-3.691944",
          "radius": "13500"
        }
      }
    }
    ]
  }
}
EOF
```

Let's consider a query whose scope is inside a radius of 15 km (15000 meters) centred in Madrid.

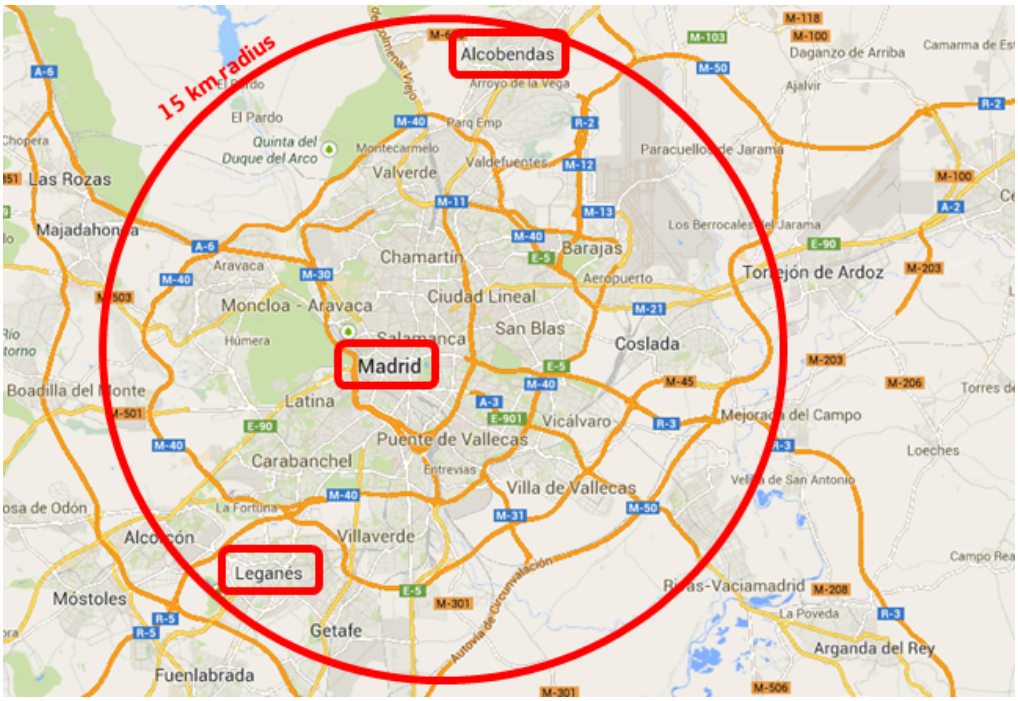

Illustration 6. Sample city area (15 km radius)

The result of the query would be Madrid, Leganes and Alcobendas.

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>15000</radius>
          </circle>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
  {
    "type": "City",
    "isPattern": "true",
    "id": ".*"
  }
  ],
  "restriction": {
    "scopes": [
    {
      "type" : "FIWARE_Location",
      "value" : {
        "circle": {
          "centerLatitude": "40.418889",
          "centerLongitude": "-3.691944",
          "radius": "15000"
        }
      }
    }
    ]
  }
}
EOF
```

Let's consider a query whose scope is outside a radius of 13.5 km (13500 meters) centred in Madrid.

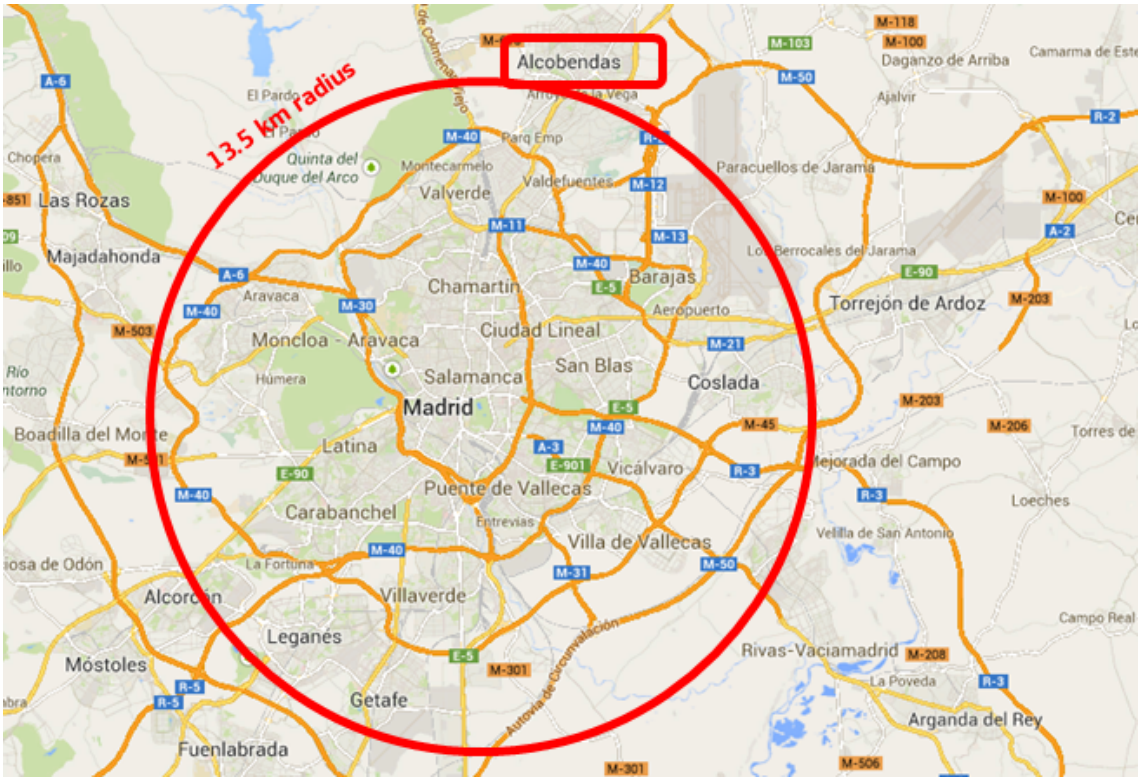

Illustration 7. Sample city area (13.5 km outside)

We use the inverted element set to "true". The result of the query would be Alcobendas.

XML

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/xml' --header 'Fiware-Service: MyService' -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>13500</radius>
            <inverted>true</inverted>
          </circle>
        </scopeValue>
      </operationScope>
    </scope>
  </restriction>
</queryContextRequest>
EOF
```

JSON

```
(curl [cb_host]:[cb_port]/NGSI10/queryContext -s -S --header 'Content-Type:
application/json' --header 'Accept: application/json' --header 'Fiware-Service:
MyService' -d @- | python -mjson.tool) <<EOF
{
  "entities": [
```

```
  {
    "type": "City",
    "isPattern": "true",
    "id": ".*"
  }
  ],
  "restriction": {
    "scopes": [
    {
      "type" : "FIWARE_Location",
      "value" : {
        "circle": {
          "centerLatitude": "40.418889",
          "centerLongitude": "-3.691944",
          "radius": "13500",
          "inverted": "true"
        }
      }
    }
    ]
  }
}
EOF
```

4.13 Mixing JSON and XML requests

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

4.14 Cross-format notifications

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 '[cb_host]:[cb_port]/NGSI10/subscribeContext?notifyFormat=json' -s -S --header
'Content-Type: application/xml' --header 'Fiware-Service: MyService' -d @- | xmllint --
format -) <<EOF
<?xml version="1.0"?>
<subscribeContextRequest>
   ...
</subscribeContextRequest>
EOF
```

The above example shows a subscribeContext done using XML that will receive notifications in JSON.

4.15 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 [cb_host]:[cb_port]/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: [cb_host]:[cb_port]
> 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>
  <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.

```
# curl [cb_host]:[cb_port]/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 NSS/3.13.1.0
zlib/1.2.3 libidn/1.18 libssh2/1.2.2
> Host: [cb_host]:[cb_port]
> 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>
</orionError>
```

4.16 Known limitations

4.16.1 Attribute values

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.:

```
...
  <contextValue><tag>value</tag></contextValue>
...
  <contextValue>value</tag></contextValue>
...
```

- The escaped version of the above, e.g.

```
...
  <contextValue>&lt;tag&gt;value&lt;/tag&gt;</contextValue>
...
  <contextValue>value&lt;/tag&gt;</contextValue>
...
```

Using the above patterns for attribute values may lead to unexpected behavior in the context broker.

4.16.2 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 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>
  <head><title>Internal server error</title></head>
```

```
  <body>Some programmer needs to study the manual more carefully.</body>
</html>
```

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

4.16.3 Notification maximum size

Notification maximum size is set to 8MB. 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.

4.16.4 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.

5 References

[1] NGSI Context Management, Approved Version 1.0, May 2012 http://technical.openmobilealliance.org/Technical/release_program/docs/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf

[2] ISO 8601:1988 - Data elements and interchange formats -Information interchange - Representation of dates and times - The International Organization for Standardization, June, 1988.

[3] World Geodetic System (WGS) 1984, http://en.wikipedia.org/wiki/World_Geodetic_System

[4] Coordinate distance calculator. http://boulter.com/gps/distance/.

A. Convenience Operations summary

Operation	Description
GET /ngsi10/contextEntities/{EntityID}	Retrieve all available information about the context entity
PUT /ngsi10/contextEntities/{EntityID}	Replace a number of attribute values
POST /ngsi10/contextEntities/{EntityID}	Append context attribute values
DELETE /ngsi10/contextEntities/{EntityID}	Delete all entity information
GET /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}	Retrieve attribute value(s) and associated metadata
POST /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}	Append context attribute value
PUT /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}	Update context attribute value
DELETE /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}	Delete all attribute values
GET /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}/{valueID}	Retrieve specific attribute value
PUT /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}/{valueID}	Replace attribute value
DELETE /ngsi10/contextEntities/{EntityID}/attributes/{attributeName}/{valueID}	Delete attribute value
GET /ngsi10/contextEntityTypes/{typeName}	Retrieve all available information about all context entities having that entity type
GET /ngsi10/contextEntityTypes/{typeName}/attributes/{attributeName}	Retrieve all attribute values of the context entities of the specific entity type
POST /ngsi10/contextSubscriptions	Create a new subscription
PUT /ngsi10/contextSubscriptions/{subscriptionID}	Update subscription
DELETE /ngsi10/contextSubscriptions/{subscriptionID}	Cancel subscription