IoT Device API

The API has been developed with this original objective in mind:

Parsing an input JSON and directly executing delete/insert SPARQL queries on a Virtuoso graph database, based on a widely configurable mapping.

Nevertheless, during its design and development, in the attempt of achieving the most possible configuration flexibility, we have ended up laying the groundwork for:

A software architecture that could perform any sort of data mining and presentation based on a XML configuration, made up of an orchestrating nucleus, and a set of pluggable components.

Current API Behaviour (i.e. what the API does at now)

The IoT Device API:

1. accepts in input a JSON representation of a device and its broker;
2. transforms the JSON in a set of N-Quads;
3. loads the N-Quads to a graph database;
4. returns the device URI in response.

The transformation at step 2 is based on a XML configuration document available at:

`https://servicemap.disit.org/WebAppGrafo/api/v1/iot/insert-api-cfg.xml`

If the input JSON is incomplete, the N-Quads based on the missing data are skipped.

The API accepts HTTP POST requests to the following endpoint:

`https://servicemap.disit.org/WebAppGrafo/api/v1/iot/insert`

Before inserting, all subjects that are going to be inserted are cleared.

As a result, the API is suitable for full updates too.

If the execution completes successfully, the API returns a single line plain text response that consists of the service URI of the inserted device. Otherwise, an empty response with a *Warning* header is generated.

The IoT Device API also provides a *delete* primitive.

For the purpose, it accepts HTTP POST requests to the following endpoint:

`https://servicemap.disit.org/WebAppGrafo/api/v1/iot/delete`

The request body is expected to be a JSON that describes the device that has to be deleted, the same way as it had to be inserted.

All properties of all subjects that appear in the *generated* N-Quads are deleted. We say, generated, because the delete primitive is configurable the same way as the insert primitive is. In the current configuration, the N-Quads that represent the broker are not generated, so that the brokers are never deleted, because they could be shared among several devices.

If the delete completes successfully, a single line plain text is returned in the response body where the service URI of the deleted device can be found. Otherwise, an empty response body with a *Warning* header is generated.

The current production configuration of the delete primitive can be found at:

http://www.disit.org/ServiceMap/api/v1/iot/delete-api-cfg.xml

The IoT Device API also provides a *disable* primitive for device attributes.

For the purpose, it accepts HTTP POST requests to the following endpoint:

https://servicemap.disit.org/WebAppGrafo/api/v1/iot/disable

The request body is expected to be a JSON object with a single property, named *uri*, where the URI of the device attribute to be disabled is expected to be found.

The property `http://www.disit.org/km4city/schema#disabled` is set to `true` for the specified attribute.

If the operation completes successfully, a single line plain text is returned in the response body where the URI of the disabled attribute can be found. Otherwise, an empty response body with a *Warning* header is generated.

The current production configuration the *disable* primitive can be found at:

http://www.disit.org/ServiceMap/api/v1/iot/disable-api-cfg.xml

The IoT Device API also provides an *enable* primitive for enabling again device attributes that had been disabled.

For the purpose, it accepts HTTP POST requests to the following endpoint:

https://servicemap.disit.org/WebAppGrafo/api/v1/iot/enable

The request body is expected to be a JSON object with a single property, named *uri*, where the URI of the attribute to be enabled is expected to be found.

The property `http://www.disit.org/km4city/schema#disabled` is set to `false` for the specified attribute.

If the operation completes successfully, a single line plain text is returned in the response body where the URI of the enabled attribute can be found. Otherwise, an empty response body with a *Warning* header is generated.

The current production configuration of the *enable* primitive can be found at:

http://www.disit.org/ServiceMap/api/v1/iot/enable-api-cfg.xml

The IoT Device API also provides a *make-private* primitive for marking a device as *private*.

For the purpose, it accepts HTTP POST requests to the following endpoint:

https://servicemap.disit.org/WebAppGrafo/api/v1/iot/make-private

The request body is expected to be a JSON object with a single property, named *uri*, where the Service URI of the device that must be marked as *private* is expected to be found.

The property `http://www.disit.org/km4city/schema#ownership` is set to `private` for the specified device.

If the operation completes successfully, a single line plain text is returned in the response body where the service URI of the device can be found. Otherwise, an empty response body with a *Warning* header is generated.

The current production configuration of the *enable* primitive can be found at:

http://www.disit.org/ServiceMap/api/v1/iot/make-private-api-cfg.xml

The IoT Device API also provides a *make-public* primitive for marking a device as *public* after that it had been marked as private. Indeed, devices are *public* by default.

For the purpose, it accepts HTTP POST requests to the following endpoint:

https://servicemap.disit.org/WebAppGrafo/api/v1/iot/make-public

The request body is expected to be a JSON object with a single property, named *uri*, where the Service URI of the device that must be marked as *public* is expected to be found.

The property `http://www.disit.org/km4city/schema#ownership` is set to `public` for the specified device.

If the operation completes successfully, a single line plain text is returned in the response body where the service URI of the device can be found. Otherwise, an empty response body with a *Warning* header is generated.

The current production configuration of the *enable* primitive can be found at:

http://www.disit.org/ServiceMap/api/v1/iot/make-public-api-cfg.xml

How devices should be represented in input JSON for the insert and delete primitives

Below here the expected shaping is outlined of the input JSONs that describe the devices, brokers and their attributes that have to be inserted, deleted or updated in the Knowledge Base that is managed by the Virtuoso graph database.

The input JSON is expected to be an object with the following properties:

- **id**, a free text string where a unique identifier for the device can be found (mandatory);
- **type**, a free text string that synthetically describes what the device measures, e.g. whether it is a temperature sensor, a traffic sensor, a presence detector, or what else (mandatory);
- **kind**, a text string that is expected to be set to *sensor*, or *actuator*, for differentiating the two typologies of devices (mandatory);
- **protocol**, a text string that where the communication protocol is indicated through which the broker can be accessed, i.e. *amqp*, *mqtt*, or *ngsi*;

- **format**, a free text string that describes the format of the possible output that is produced by the device, e.g. *csv*;
- **macaddress**, a text string where the MAC address of the device can be found;
- **model**, a free text string where the model of the device (the product name provided by the manufacturer) can be found;
- **producer**, a free text string the manufacturer of the device can be found;
- **latitude**, a float number that indicates the (starting) WGS84 latitude of the device. If the device moves over the time, an attribute (see below) is expected to be defined for the device, that contains its real-time latitude (can be missing, but in that case, the broker *must* have the latitude property set and valid);
- **longitude**, a float number that indicates the (starting) WGS84 longitude of the device. If the device moves over the time, an attribute (see below) is expected to be defined for the device, that contains its real-time latitude (can be missing, but in that case, the broker *must* have the longitude property set and valid);
- **frequency**, an integer number that expresses the frequency at which the device produces its outputs, as the time interval in seconds between two consecutive detections;
- **created**, a text string where the date and time when the device has been put in place can be found;
- **broker**, a JSON object that describes the device broker (mandatory, see below);
- **attributes**, a JSON array whose items describe what the device detects (mandatory, see below);
- **uri**, the URI is provided if the device has already been registered in the Knowledge Base (update operations). If it is not provided, an insert operation is assumed, and a new URI is generated based on the broker *name* and device *id*;
- **ownership**, that is expected to be set to *public*, or *private*.

The device *broker*, i.e. the software/hardware component that acts as an interface between the physical devices and the data consumers, is expected to be represented through a JSON object with these properties:

- **name**, a free text string where the broker name can be found (mandatory);
- **type**, a text string in lowercase letters that indicates whether it is a AMQP, MQTT, or NGSI broker (mandatory);
- **ip**, a text string where the IP address of the broker can be found (mandatory);
- **port**, an integer number that indicates the network port that is opened on the broker. Together with the IP address, it makes up the broker *endpoint* (mandatory);
- **login**, a text string that indicates the user name that should be used for requesting data to the broker;
- **password**, a text string that indicates the password that should be used in combination with the user name for requesting data to the broker;
- **latitude**, a float number where the WGS84 latitude of the broker can be found (mandatory if a latitude has not been specified for the device);
- **longitude**, a float number where the WGS84 longitude of the broker can be found (mandatory if a latitude has not been specified for the device);
- **created**, a text string where the date and time when the broker has been put in place can be found.

Each *device attribute* is expected to have these properties:

- **value_name**, a free text string that briefly indicates what type of detection the attribute represents, e.g. *tempXX*, *humXX*, *thievesInLivingRoom* (mandatory);
- **data_type**, a text string that indicates the data type of the values that the device produces in output for the specific type of detections, e.g. *integer*, *float*, *Boolean*, *string*. A list of the allowed values for *data_type* has been defined and can be found in the XML request configuration documents (mandatory);

- **value_type**, a text string where a high-level indication can be found of the type of detections that the attribute represents, that is aimed at enabling searches of all devices that detect the same thing in different locations. A list of the allowed values for *value_type* has been defined, but it cannot be found in the configuration documents. Instead, the list of the allowed values can be found in the destination Knowledge Base, shaped as a set of SSN Property instances (mandatory);
- **value_unit**, the unit of measure of the values that are produced in output as a result of the detections of the type that is represented by the device attribute. A list of the allowed *value_unit* can be found in the XML request configuration documents (mandatory);
- **value_refresh_rate**, an integer number that expresses the frequency at which the device produces outputs of the type that is represented by the device attribute. The frequency is expressed as the time interval in seconds between two consecutive detections (mandatory);
- **healthiness_criteria**, a text string that indicates how it is possible to determine if the device is correctly producing output values as a result of the detections of the type that is represented by the device attribute. The allowed values are: *refresh_rate* (there is a problem if the expected refresh rate is not respected), *different_values* (there is a problem if the device produces in output the same value for more than *consecutive_detections* detections), or within_bounds (there is a problem if the output value does not fall within an interval that is described through the value_bounds property);
- **different_values**, an integer number, see the *healthiness_criteria* for details about its semantic;
- **within_bounds**, a text string with a well-defined syntax, that represents a value interval. See the *healthiness_criteria* for details about its semantic.

How to browse the N-Quads that are produced in the graph database

The device is stored in the Knowledge Base as an instance of one of the two following classes:

- `http://www.disit.org/km4city/schema#IoTSensor`
- `http://www.disit.org/km4city/schema#IoTActuator`

The device is assigned a URI that is built conforming to the following template:

`http://www.disit.org/km4city/resource/iot/{1}/{2}`

where:

1. the *name* property of the broker JSON object that has been received in input;
2. the *id* property of the root (device) JSON object that has been received in input.

If the *uri* property is set for the device, that URI is used, without composing it from the device and broker properties.

On the *IoTSensor* or *IoTActuator* instances, appropriate properties can be found for describing the device, its attributes (for each attribute, a `http://www.disit.org/km4city/schema#DeviceAttribute` is generated and is referenced from the device through the `http://www.disit.org/km4city/schema#hasAttribute` property), and its broker (the `http://purl.oclc.org/NET/UNIS/fiware/iot-lite#exposedBy` property).

Software Architecture Overview

A *Provider* retrieves data from a *Repos*. The specialized Request Provider is implemented that provides raw request body and request parameters. Indeed, the HTTP request body and parameters make up a *Repos* that is automatically configured for all requests. A *Builder* builds *Data*. A minimal set of specialized *Builders* is implemented at today, including the *Lookup Builder* that builds *Data* querying a *Provider*, and the self-explanatory *Const*, *Switch*, *Template, Alternative and Bean Builders*. Each *Data* has an enforced user-defined data type mapped to a Java class, and a set of values. Validators check the data quality and take actions. The *Basic Validator* that is implemented at today is suitable for checking cardinalities, string matches, and numeric thresholds, and can interrupt the process, or strip away the dirty data, and/or print a log message.

Loaders load *Data* to a *Repos*. The *Virtuoso Loader* is implemented at now, and it can also be configured to work in delete/insert (the SPARQL way for transactional updates). The computation process is outlined in the request configuration, and it is made up of a set of *Data* build stages. At each stage a *Data* instance is produced, possibly marked for being sent as output at the end of the process, possibly validated, and possibly loaded to a Repos. Each request has its own request management configuration file, whose location can be read in the Deployment Descriptor.

Request Fulfilment

Two requests are managed by the application at today: *insert* and *delete*. The execution flow is the same for the two requests: the only difference is that when inserting, the *load* method is invoked on the loader, while when deleting, the *unload* method is invoked. All other aspects are managed through the request configurations. The execution flow can be summarized this way:

1. The request raw body and parameters, and the request configuration, are parsed;
2. The configured process is executed, made up of stages, each decomposable this way:
    a. The *Data* instance is built through the intended *Builder*, configured for the purpose;
    b. The *Data* instance is validated/cleaned based on a configured validation process, that could involve the employment of several validators and validation criteria;
    c. If the validation is passed, non-obvious configured dependencies from other *Data* instances are retrieved, evaluated, and stored in the Data instance, indeed a *Data* instance can be configured as *triggered* by a different *Data* instance, so that if the latter is not available, the former is skipped from being sent in output and loaded, even if they could be built independently of each other;
    d. The flag that specifies if the *Data* instance must be sent in output at the end of the process is retrieved from the configuration, and it is stored in the *Data* instance;
    e. The Data instance is loaded in memory (through the virtual *volatile* loader), or to a *Repos*, throw an implemented specialized *Loader*, such as the *Virtuoso Loader*;
3. The loaders are disconnected (and the loaded data is flushed in case of transactional loads);
4. The response is produced and delivered to the user agent.

Request Configuration

Some details are provided here about how to configure the management of a request. An initialization parameter can be found in the deployment descriptor that specifies where the configuration file can be found, for each of the exposed primitives (i.e. implemented servlets), with respect to the user home.

Admin

The Admin section is only expected to contain at today a *xlogs* section, where the application logging is configured. The *xlogs destination* attribute specifies where the XML logs must be put. It is expected to be a folder, and the path is expected to be relative to the user home. For each request, a XML log file is produced, if something exist that must be logged. The *xlog-level* element configures the logging level at class level. What happens in the unconfigured classes is not logged.

Repos

It is where data repositories are configured. Each repository is expected to have assigned a unique id. Parameters can be configured for a repository. Each parameter is expected to be a key value pair. Parameters are specific to the repository, and they are expected to be meaningful to the software components (providers, loaders) that access the repository. The {request} repository is implicitly defined in every configuration, and it is the virtual Repos from where request body and parameters can be retrieved.

Providers

It is where data providers are configured. They are the software components that retrieve the input data and make it available for that it could be processed for fulfilling the request. Each provider is expected to have assigned a unique id. Each provider is expected to be implemented by a class, whose qualified name must be specified in the configuration through the *class* attribute. The repository from where the provider extracts the data is expected to be specified in the *repo* attribute. The value of the *repo* attribute must match the unique id of a repository. Providers are *pluggable* components. Further details below.

Builders

It is where builders are declared. Each builder must have a unique id assigned. For each builder the implementing class must be provided through the *class* attribute. Builders are *pluggable* components. Further details below.

Datatypes

Each datatype is expected to have an arbitrary unique id assigned. Also, each datatype must be bound to its implementing Java class through the *class* attribute. Each datatype implementing class is expected to be a wrapper that include: (i) the wrapped typed value; (ii) the *fromString* method, with a String input parameter, where datatype specific data validations are expected to be performed before setting the value; (iii) a possible override of the toString method. This is functional to the datatype enforcement at *Data* build time.

Loaders

It is where the software components that persist *Data* are configured. The implicit *volatile* loader is always available, and it stores the *Data* instance in the volatile (RAM) memory. For each loader, a unique id must be provided. Also, the implementing class must be specified through the *class* attribute. Also, the repository where the loader stores the *Data* must be specified through the *repo* attribute that must match a repository unique identifier. If the same software component is leveraged for accessing two different repositories, two repositories are expected to be configured with different *repo* and same *class*. In the *loader* subtree, further loader-specific configurations can be set.

Data quality

It is where all data checks are configured. In the *validators* subtree, the leveraged validating components are declared. Each of them is expected to have an id assigned, and the implementing *class* specified.

In the *validators* subtree, a set of *validate* subtrees can be found, each targeting a specific *Data* instance through the *ref* attribute. Within each *validate* subtree, one or more *validator* subtrees are expected to be found, each referring one of the declared *validators* through the *ref* attribute. In each *validator* subtree, *validator-specific* XML is expected to be found, that describes the checks that must be performed. Nevertheless, for each configured check, a unique id is expected to be provided for logging purposes, and the severity is expected to be provided through a *lev* attribute (SEVERE for interrupting the process, lower levels for a simple logging). The cleaning action (strip away the invalid values without interrupting the process) can also be configured setting the *clean* attribute to *yes* at data check level.

Process

It is where the computation process is outlined. At now, data builds are executed in the order in which they appear, but a future development is to parallelize the parallelizable, so the sorting of the data elements in the process subtree must be considered not to be enforced. For each Data build (*data* element), the following are expected through appropriate attributes: (i) unique identifier of the Data instance to be built; (ii) data type id; (iii) builder id, i.e. the software component that must be employed for producing the *Data* instance; (iv) loader id, that is the software component that has to be employed for persisting the Data instance; (v) the possible *output* flag; (vi) the possible triggering (parent) *Data* instance. In each data subtree, *builder-specific* configurations are expected to be found.

Pluggable Components

In this section, the implemented pluggable components are described. They are seven builders (i.e. Alternative, Bean, Const, JsonParser, Lookup, Switch, and Template), one provider (the Request Provider), one loader (the Virtuoso Loader), and one validator (the Basic Validator).

Request Provider

The Request Provider accesses the reserved repository *{request}*, and provides the raw request body, or the value of a specific request parameter, based on the query. The provider is leveraged in the context of a *Lookup Builder*. More specifically, for extracting, let's say, the raw request body, one should put something like this in the *process* subtree of the configuration document.

```
<data id="request-body" type="string" builder="lookup" loader="volatile">
<lookup><priority>1</priority><provider>request</provider><query>{raw}</query></lookup>
</data>
```

Instead, for retrieving a specific argument, the argument name should be put in the query element without braces. The query can also be retrieved from a previously built data instead of being hardcoded in the lookup configuration, adding a *ref* attribute to the query element, whose value is the unique identifier of a *Data* instance. *Type, builder and loader attribute values are expected to be unique identifiers of components declared in appropriate sections of the configuration doc, and the same applies to all builders.*

Virtuoso Provider

The Virtuoso Provider queries a Virtuoso database instance and fills the *data* instance that must be built with a (set of) serialized JSON objects, one for each row that is returned by the query.

This sample configuration fragment:

```
<data id="dest-data" type="string" builder="lookup" loader="volatile"><lookup>
<priority>1</priority><provider>virtuoso-provider</provider><query ref="query-txt" />
</lookup></data>
```

retrieves the query that must be performed reading from the (previously built) `query-txt` variable and writes to the `dest-data` array of values, generating an item (string, serialized JSON) for each row of the query result set, where a property can be found for each column that appears in the query result set.

The query text could also be provided putting it between the query opening and closing tag, instead of referencing a previously built variable.

The `provider` element contains the identifier of an appropriate provider that is expected to be defined in the `providers` section of the configuration document.

An implementation of a Virtuoso Provider has been produced while developing the IoT Device API, and can be found in the `org.disit.iotdeviceapi.providers.virtuoso` package.

Alternative Builder

It is a builder that builds a new *Data* instance evaluating in sequence a set of *Data* instances and setting the values of the new instance equal to the values of the first non-empty instance that can be found in the set. Alternatives not only can be expressed in the form of a referenced *Data* instance to be evaluated, but they also can be hardcoded in the data build configuration, putting the value as simple text within the alternative element opening and closing. This way, the *Alternative Builder* can be used for setting default values. As an example, this fragment builds a Data instance attempting to retrieve the latitude from the device, and if it is not available, from the broker. Device and broker latitudes have been previously put in dedicated, separated, *Data* instances that are referred here through the ref attribute.

```
<data id="latitude" type="float" builder="alternative" loader="volatile">
<alternative ref="device-latitude"/>
```

```
<alternative ref="broker-latitude"/>
<alternative>-1</alternative>
</data>
```

Bean Builder

The *Bean Builder* builds a bean instance. Assuming you have a data type that is implemented through a class that has a set of members, you can use the *Bean Builder* to produce an instance of such a class, setting the values of its members equal to *Data* instances that you have previously produced. As an example, if you have a class that models a N-Quad with graph, subject, property, and filler, you can add a data build configuration like this for producing a specific Quad instance:

```
<data id="graph-uri.service-uri.is-a.device-type" type="quad" builder="bean"
loader="virtuoso"><member name="graph" ref="graph-uri"/><member name="subject"
ref="service-uri"/><member name="property" ref="is-a"/><member name="filler" ref="device-
type"/></data>
```

It could be a future development the enforcement of the member datatypes, through appropriate configuration of the bean data types, and the introduction of data type checks within the *Bean Builder*.

Const Builder

The Const Builder produces a Data Instance and initialize its values with a single value hardcoded in the data build configuration. Briefly, it initializes a *Data* instance with a constant. A future development could be the opening to the possibility of initializing *arrays*. The sample fragment builds the Data instance named *graph*, with a hardcoded URL.

```
<data id="graph" type="uri" builder="const" loader="volatile">http://graph.xyz/</data>
```

JSON Parser

The *JSON Parser Builder* builds a *Data* instance through the parsing of a JSON that can be taken from a previously produced *Data* instances, or put hardcoded in the *Data* build configuration, and the extraction of a specific (set of) values. The source JSON is provided through the *source* element. The JSON can be put as plain text within the tag opening and closing, or the *ref* attribute can be used for addressing a Data instance where the JSON can be found. The query can also be hardcoded in the data build configuration or taken from a previously build Data instance through the *ref* attribute. The query syntax is: a path made up of object/property names separated by single dots without any leading or trailing character.

```
<data id="broker-latitude" type="float" builder="jsonp" loader="volatile">
<source ref="request-body"/><query>broker.latitude</query></data>
```

Lookup Builder

The *Lookup Builder* builds a *Data* instance querying a *Provider*. More alternative queries are allowed, each with a given priority, that are executed until one of them returns some result. Query priority, target provider, and query, can be taken from existing *Data* instances through the *ref* attribute, or can be hardcoded in the *Data* build configuration. A sample *Data* build where the *Request Provider* is queried for retrieving the raw request body (reserved parameter name *{raw}*), is proposed below here.

```
<data id="request-body" type="string" builder="lookup" loader="volatile"><lookup>
<priority>1</priority><provider>request</provider><query>{raw}</query></lookup></data>
```

Switch Builder

The *Switch Builder* builds a *Data* instance implementing a logic that is like the logic of a Java switch. The *switch* element is expected to have a *ref* attribute where the unique id of the *Data* instance to be inspected can be found. The *case* elements are all evaluated. Each value of the reference Data instance is compared to each value of the *if* Data instance (or to the string value enclosed in the *if* element), and if at least one match can be found, the values of the *then* Data instance (or the string value enclosed in the *then* element) is

appended to the set of the values. A sample *Data* build configuration is proposed below here where the *Switch Builder* is leveraged.

```
<data id="device-type" type="uri" builder="switch" loader="volatile"><switch ref="kind">
<case><if>sensor</if><then ref="iot-sensor-uri"/></case><case><if>actuator</if>
<then ref="iot-actuator-uri"/></case></switch></data>
```

Template Builder

The *Template Builder* builds a *Data* instance based on a template string that can be hardcoded in the data build configuration or retrieved from an existing *Data* instance through the ref attribute. The template string is expected to include one or more parameter placeholders, each in the form of an integer number enclosed in braces, starting from zero, and raising without discontinuities. The *param* elements define where the parameter values must be taken from, and where they must be put in the template string, respectively through the *ref* and the *index* attributes. A sample *Data* build where the *Template Builder* is leveraged is proposed below here.

```
<data id="broker-endpoint" type="string" builder="template" loader="volatile">
<template>{0}:{1}</template><param index="0" value="broker-ip"/>
<param index="1" value="broker-port"/></data>
```

Basic Validator

Differently from the *Builders*, that are leveraged through the configuring of *Data* builds that are enclosed in the *process* section of the configuration document, the *Basic Validator*, and all possible future *Validators*, are leveraged through the configuring of validations that can be found in the *data-quality* section of the configuration document. The *Basic Validator* can be used for checking that one or more of the following is satisfied:

(i) the checked *Data* instance has a minimum cardinality (minimum number of values in its array of values), through the *min-cardinality* element;
(ii) the checked *Data* instance has a maximum cardinality, through the *max-cardinality* element;
(iii) the checked *Data* instance has the same cardinality of a specified (different) *Data* instance, through the *same-cardinality* element;
(iv) each value in the checked *Data* instance matches one at least of the values that can be found in a different *Data* instance, or a specific hardcoded value, through the *match* element;
(v) each value in the checked *Data* instance is lower than the values that can be found in a different *Data* instance, or a specific hardcoded value, through the *lower-than* element;
(vi) each value in the checked *Data* instance is greater than the values that can be found in a different *Data* instance, or a specific hardcoded value, through the *greater-than* element.

Reference values can be hardcoded (enclosed in their respective element tags) or retrieved from existing *Data* instances.

The definition of a set of alternative validating conditions can also be set, through the *op* attribute of the *pick-validator* element under which the set of the alternative conditions is represented. In this case, the *op* attribute must be set to *or*. Otherwise, it must be set to *and*, or omitted.

For each validation, a level must be provided through the *lev* attribute: *SEVERE* leads to the process interruption in the case the check fails, while other levels lead to the delivering of a log message of the specified level. In those cases where a set of alternative conditions is defined, the *lev* attribute is also expected to be set for the *pick-validator* element, so that it could be clear what should be done if none of the alternative conditions is satisfied. If and where the log message will appear depends of the configured level in the admin section of the configuration file, for the *BasicValidator* class.

In the case of non-*SEVERE* checks, the dirty values can be stripped away from the checked *Data* instance, setting the *clean* attribute to *yes* at single check level.

For the *match*, *lower-than*, and *greater-than* checks, an *op* attribute is also expected that specifies if the condition must be valid for all values that can be found in the reference Data instance (attribute value: *and*) or it is enough that the condition is valid for one of the values (attribute value: *or*).

A sample configuration of a validation of the *Data* instance named (unique id) *kind* is proposed below here.

```
<validate ref="kind"><pick-validator ref="basic">
<min-cardinality id="kind.min-cardinality" lev="SEVERE">1</min-cardinality>
<max-cardinality id="kind.max-cardinality" lev="SEVERE">1</max-cardinality>
<match id="kind.match" op="or" lev="SEVERE"><match-value>sensor</match-value>
<match-value>actuator</match-value></match></pick-validator></validate>
```

Virtuoso Loader

The Virtuoso Loader loads the Data instances to a Virtuoso graph database instance. The configuration of the repository (in the *repos* section of the configuration document) where the loader loads (and unloads) the triples, have a key role. A sample configuration follows:

```
<repo id="virtuoso">
<param key="virtuosoEndpoint" value="localhost:1111"/>
<param key="virtuosoUser" value="dba"/>
<param key="virtuosoPwd" value="dba"/>
<param key="virtuosoTimeout" value="600"/>
<param key="virtuosoDeleteInsert" value="subject"/>
</repo>
```

The non-obvious parameter is the *virtuosoDeleteInsert*. Three values are admitted for it: *subject*, *property*, or *disabled*. Subject means that before loading a quad, all N-Quads with the same graph and subject of the one that is going to be loaded, are unloaded. In other words, subjects are cleared before every insert. It is reasonable in those cases where insert requests are used indifferently for new inserts and for full updates, and we wish to handle the two cases indifferently. Similarly, if it is set to property, it means that before inserting, all N-Quads that have the same graph, subject and property of the one(s) that is going to be loaded, are deleted. The disabled value disables deletions at insert time. Remarkably, if the delete/insert is enabled at any level, the loading becomes transactional: N-Quads that have to be loaded/unloaded are kept in memory, and the actual operations are performed on the Knowledge Base only when the loader is disconnected, at the end of the process, and in the only case when its status is valid (no errors have occurred during N-Quads loading and unloading along the process). In insert/update modality, *untriggered* N-Quads (quads that are wrapped in untriggered Data) are used for determining what has to be possibly deleted but are not inserted. The data type enforcement is achieved at the loading time through a configuration that is expected to be provided in the *loaders* section of the configuration document. Something like this, where the *ref* attribute of *formatting* is the unique identifier of a configured data type, and the text enclosed in the formatting tag is a template where it is indicated how the value must be written in the SPARQL query when it has that specific data type:

```
<loader id="virtuoso" class="org.disit.iotdeviceapi.loaders.virtuoso.VirtuosoLoader"
repo="virtuoso"><formattings><formatting ref="string">&amp;quot;{0}&amp;quot;</formatting>
<formatting ref="integer">
&amp;quot;{0}&amp;quot;^^&amp;lt;http://www.w3.org/2001/XMLSchema#integer&amp;gt;
</formatting><formatting ref="float">
&amp;quot;{0}&amp;quot;^^&amp;lt;http://www.w3.org/2001/XMLSchema#float&amp;gt;
</formatting><formatting ref="uri">&amp;lt;{0}&amp;gt;</formatting>
<formatting ref="geometry">
&amp;quot;{0}&amp;quot;^^&amp;lt;http://www.openlinksw.com/schemas/virtrdf#Geometry&amp;gt;
</formatting></formattings></loader>
```