EdgeX Foundry
California/Dehli
Device Service Functional Requirements

Version: 8
July 3, 2018

Tony Espy <espy@canonical.com>
Jim White <james.white2@dell.com>

Introduction

This document is a high-level functional requirements specification for EdgeX Foundry device services (DS), and as such provides the same purpose for EdgeX SDK implementations. An EdgeX DS is a micro service that forms a bridge between one or more devices on a system, usually of a similar type, and the EdgeX core services (e.g. Core Data, Core Metadata, …). A DS must provide a set of standard REST APIs which allows it to inter operate with the other EdgeX core services.

Note *– while it's possible to run a DS on a separate (i.e. external) system from the rest of an EdgeX instance, doing so is outside the scope of this document, which focuses soley on EdgeX instances where all micro services run on a single system.*

Service operation

This section describes the service-level functionality that a DS needs to implement in order to properly function in an EdgeX system. For more details on the entire EdgeX architecture, including details of other services mentioned in this document, please refer to the EdgeX Foundry wiki:

https://wiki.edgexfoundry.org/display/FA/EdgeX+Foundry+Microservices+Architecture

Startup

An SDK should provide all of the logic described in this section except for device creation, which is the sole responsibility of a DS implementation, however an SDK may provide helper functions.

Please refer to the Core Metadata RAML for details on the Core Metadata objects discussed in this document:

https://github.com/edgexfoundry/edgex-go/blob/master/core/metadata/raml/core-metadata.raml

DeviceService

On startup, a DS must first query the Core Metadata service to determine if a `DeviceService` instance (and associated `Addressable` instance) with its `serviceName` already exists. If the Core Metadata service is unavailable, then the DS must log an error and exit. If a matching `DeviceService` instance is not found, then the DS must create a new `Deviceservice` instance representing itself in Core Metata. If a matching instance is found, a DS may optionally use it to bootstrap itself. If the DS is configured to use the registry (see *Configuration settings* for more details), then once the DS finishes with initialization, it must register as a service to the registry. If registration fails, the DS must log an error and exit. If the DS is not configured to use the registry (i.e. during development) and more than one instance of the DS is already running, the resulting behavior is undefined.

DeviceProfiles

A DS needs at minimum one `DeviceProfile` defined in Core Metadata to represent a class of devices it manages, this profile may be created at startup via import of YAML file, or created on-demand prior to the first device being added to the DS. Device profiles are in turn used by a DS to create one or more `ValueDescriptors` in Core Data. A DS may use pre-existing `DeviceProfile` instances read from Core Metadata and/or create one or more new `DeviceProfile` instances on startup.

Whether new `DeviceProfile` instances are created from static definitions at DS startup, are or created dynamically via some type of sensor discovery mechanism is DS implementation specific. A DS SDK must provide a mechanism a DS can use to add new `DeviceProfiles`**,** regardless of how they're generated.

Devices

A DS may optionally create one or more `Device` instances in Core Metadata at startup. The DS is free to implement its own mechanism (e.g. static white-list, config setting, …) for device creation at startup. Generally this approach is orthogonal to dynamic device discovery, so a DS usually implements one or the other, but not both. A DS may optionally use any existing devices found in Core Metadata with a matching `serviceId`. If this is supported, the DS is responsible for updating the `adminState` and `operatingState` of the devices.

Initialization / Disconnection

A DS SDK must provide automatic triggering of a device initialization command if configured (by settings) and supported by the corresponding `DeviceProfile` (i.e. the specified command exists). Likewise if a device is removed, the same applies to device removal, the SDK must trigger an automatic device disconnect command if configured and the supported by the `DeviceProfile`. These commands are triggered internally by the SDK itself.

operatingState

A DS is responsible for initializing and updating the current `operatingState` of each of its devices in Core Metadata. The criteria used to determine the `operatingState` of a device is DS-specific. A DS may choose to set operatingState to "**DISABLED**" if one or more errors occur while communicating with the device. Likewise, a DS could choose to set `operatingState` to "**ENABLED**" due to asynchronous events from the device, or it might choose to implement some sort of polling to monitor the health of its devices. A DS must return an error (status code = 423) to any REST API call which specifies a disabled device (note, this doesn't apply to REST API calls which apply to all devices).

ValueDescriptors

A DS may create `ValueDescriptor` instances in Core Data for device resources defined in one of its `DeviceProfiles`, but again as with DeviceProfile instances, any existing `ValueDescriptor` instances in Core Data can also be used.

For more details on `ValueDescriptors`, please refer to the Core Data RAML:

https://github.com/edgexfoundry/edgex-go/blob/master/core/data/raml/core-data.raml

Note – as `ValueDescriptor` instances must have a unique name, and are not tagged with a `serviceName` or `serviceId`, cleanup of existing `ValueDescriptor` instances for now is considered outside the scope of this document.

ProvisionWatchers

A DS which supports dynamic device discovery requires one or more `ProvisionWatcher` instances. Like with other Core Metadata objects, these can be pre-exisitng, or may be created by the DS itself.

A `ProvisionWatcher` provides a map of identifiers; the map key being the identifier name. During the periodic device discovery process, newly available devices are matched against the full set of identifiers (generic or DS-specific) defined in each `ProvisionWatcher` instance. If a match is found, and there's no existing `device` instance in the DS, then a new `device` instance is created using the `DeviceProfile` specified by the `ProvisionWatcher` instance.

Note – `ProvisionWatcher` identifiers support basic glob-style filtering, thus a single `ProvisionWatcher` instance can be used to match multiple devices. It's also possible however for a `ProvisionWatcher` to specify fully-qualified identifiers, which creates a one to one mapping to specific devices.

Configuration settings

EdgeX services all support a core set of configuration settings, each with well-known names, which are used to tailor the runtime behavior of the services. In addition, each service should honor boostrap command-line options (which also can be specified as environment variables) which affect how configuration is loaded. The most important is **--registry**, which indicates a service must read all of its configuration settings from the registry. If the registry isn't enabled, then a DS must read its configuration settings from a local TOML[1] formatted file, called `configuration.toml` (see the Base Services design for more details). A DS may also optionally read protocol-specific settings called `protocol.toml`. A DS might also provide some means of overriding the default settings, either by allowing a path to the persistent store to be specified on the command-line, or allowing an individual setting to be overriden on the command-line.

For more details about the bootstrap options and how the affect service startup, please refer to the Base Services specification:

https://wiki.edgexfoundry.org/download/attachments/7602423/ServiceNameDesign-v6.pdf?version=1&modificationDate=1523468903687&api=v2

A DS must provide documentation (preferably included in the DS project's VCS) for all non-generic settings.

If registry usage is enabled, then a DS must query the registry for its settings, which override any local settings, before it registers itself as a service with the registry.

***Note** - the following two sub-sections are considered out of scope for the California release.*

Monitoring settings changes

A DS must also ensure that it monitors the registry for settings, so that any changes to writeable settings are detected and dynamically applied to the DS. A DS is responsible for ensuring that setting(s) changes where applicable, are used to update the corresponding objects in the Core Data or Core Metadata services. A DS can, based on the settings being changed, dynamically adjust itself to work with the new settings. If a DS cannot dynamically apply a changed setting, it must log an error, and the change will not be applied until the DS is restarted.or it can also choose to store the change and use it on restart of the service. This behavior must be specified in the DS settings documentation.

Startup settings changes

On startup, if a DS detects changes in settings which impact pre-existing Core Metadata objects, it must update those objects in Core Metadata before it becomes operational (i.e. registers with Consul). This rule only applies to a small set of settings which should apply across all DSs (e.g. host, labels, port, …).

***TODO**: update **Appendix A** to denote which settings are covered by this rule.*

Device discovery

As mentioned in *Service startup*, a DS can define a static list of devices, however as some devices or sensors may not always be present and/or connected (wired or wireless), a DS may choose to implement dynamic discovery instead of a static device configuration.

What device discovery handles is detection of device availability, which then can trigger white-listed devices being automatically added, or if the device already exists in Core Metadata, updating the device's operatingState to **ENABLED**. White-listing is handled via creation of one or more `provisionwatcher` instances in Core Metadata. A DS SDK may provide a means of configuring one or more `provisionwatcher` instances on startup (see **Appendix A**).

If a DS supports this feature, then a periodic call is made to the SDK's discovery endpoint via the Scheduler service per a `ScheduleEvent` (and associated `Schedule`) instance defined in Core Metadata. This REST call triggers an internal SDK call to the DS protocol-specific layer which returns a list of `device` objects, each a map of identifier attributes (e.g. name, MAC, …) for each available device. The SDK must then compare the returned scan list of device objects to the identifiers specified in each `provisionwatcher` instance, and if a match is found, then device is added or updated as described at the start of this section.

If a DS supports device discovery, then it should include a configuration setting called `DeviceDiscovery`, which can be true or false, and controls whether or not device discovery is enabled.

Note – all REST APIs defined in this document actually must include a prefix for versioning, **/api/v1/**, which is left out for brevity's sake.

An SDK must provide the following REST API endpoint for discovery:

`/discovery`

status codes:

200: discovery has been triggered

423: the service is locked (admin state) or disabled (operating state)

500: unknown or unaticipated issues exist

A DS which supports dynamic discovery must provide a language-specific discovery hook, which is called whenever a REST API request is made to the **/discovery** endpoint. A DS is free to use the Scheduling service to create a periodic call to its discovery endpoint.

Note – a DS may choose to trigger sensor auto-discovery in addition to device discovery when this endpoint is called.

Device authentication

EdgeX currently doesn't support any kind of authentication of devices. Any required authentication (e.g. Bluetooth pairing) must be performed out-of-band.

Health checks

A DS must implement a basic health check endpoint. If configured to use the registry, the registry will use this endpoint to determine the availability of the DS. The registry will periodically call this endpoint. If a call fails, the registry will consider the service unavailable, and will no longer return the address of a DS in response to queries from other services.

Note - per the base services design document, a DS no longer needs to update it's operating state in Core Metadata.

An SDK must provide the following REST API endpoint, which by default is configured for use by Consul's health check mechanism to ensure that the DS is functioning properly.

`/ping`

status codes:

200: returns the value "pong"; indicates that the service available

500: unknown or unanticipated issues exist; service should be considered unavailable

Device readings

One of the primary goals of a DS is communicating data (aka readings) from the devices and sensors to Core Data. This may be done on a scheduled basis or may be done via some internal trigger such as a device asynchronously pushing new readings to the DS which it then forwards to Core Data. Optionally, the successful receipt of data from a sensor/device may be used to update the device's operating state (see below for requirement).

Readings are passed from the DS protocol specific logic to the core SDK as native types and then converted to strings before being sent to Core Data, The `PropertyValue` for a device resource (aka object) has a `type` attribute which can be set as follows:

Binary values are specified using the type value `bytes`, and passed as a base64 encoded string.

Boolean values are specified using the type value `bool`, passed as the strings "true" or "false".

String values are specified using the type value `string`.

Floating point values are specified using types as specified by the Go programming language: `float32`, `float64`, which map to single precision or double precision (per IEEE 754 standard). These types should easily be mapped to similar types in

other programming languages. When EdgeX is given (or returns) a float32 or float64 value as a string, the format of the string is a base64 encoded big-endian (i.e. network byte order) of the float32 or float64 value.

Integer values are specified using types as specified by the Go programming language: `uint8` , `uint16` , `uint32` , `uint64` , `int8` , `int16` , `int32` and `int64`. These types should easily be mapped to other programming languages. When EdgeX is given (or returns) an integer value as a string, the format of the string is a simple ASCII numeric string (e.g. "12345").

Note 2 - as the collection of data occurs and the DS communicates these readings to Core Data, a DS may optionally choose to locally cache the readings (see *Query commands*). Caching readings is not a requirement for a DS SDK for the Dehli release.

Query commands

A DS must support query type commands (aka device readings) used to get the latest reading for one or more device resources (as defined by the device's `DeviceProfile`). If one or more valid readings are read from a device/sensor in response to a query command, the DS shall send an `event` object to Core Data containing one or more `reading` objects**,** in addition to returning the readings as a response to the query command.

Optionally, a query that triggers a successful reading from a sensor/device can result in the device's `operatingState` being updated (see *Metadata updates*).

Note - a query handled by a DS may not always result in a request to a single device/sensor. Depending on the device/sensor and mediating software, a query can result in multiple requests to the underlying devices/sensors or even multiple requests to a single device/sensor. In other words, DSs can implement aggregation of devices and/or trigger multiple readings based on a single external query. For example, a single client command "current settings", for the latest cooling and heating set points of a thermostat, may result in two individual requests by the DS to the underlying device to get the device's cooling set point and to get the device's heating set point.

An SDK must implement the following REST GET endpoint to support device readings:

`/device/{id}/{command}` - issue named query command to the specified device/sensor

parameters:

> **id:** the database-generated device ID
> **command:** the command name, defined in the device's corresponding `DeviceProfile`. The command specified must match an existing command, resource, or device resource name from the `DeviceProfile`.

status codes:

200: returns a JSON `event` which contains one or more `Readings`. The setting `MaxCmdOps` defines the maximum number of aggregate (as defined by the `DeviceProfile`) operations per command.

Example response:

```
{"id":"","pushed":0,"device":"XDK01","created":0,"modified":0,"origin":0,"schedule":null,"event":null,"reading
s":[{"id":"","pushed":0,"created":0,"origin":1529592066,"modified":0,"device":"XDK01","name":"AccelX","value":
"0.00747"}]}
```

404: the specified command and/or device doesn't exist

423: the device or service is locked (admin state) or disabled (operating state)

500: unanticipated or unknown issues encountered, this includes a failed assertion for one or more readings

`/device/all/{command}` - issue named query command to all operational devices

parameters:

> **command:** the command name, defined in the device's corresponding `DeviceProfile`. The command specified must match an existing command, resource, or device resource name from the `DeviceProfile`.

status codes:

200: returns a JSON array of `Events` containing `Readings` as returned by the devices/sensors through the DS. Responses to this endpoint must include `DeviceName`.

Example response:


```
{[
{"id":"","pushed":0,"device":"XDK01","created":0,"modified":0,"origin":0,"schedule":null,"event":null,"reading
s":[{"id":"","pushed":0,"created":0,"origin":1529592066,"modified":0,"device":"XDK01","name":"AccelX","value":
"0.00777"}]},
{"id":"","pushed":0,"device":"XDK02","created":0,"modified":0,"origin":0,"schedule":null,"event":null,"reading
s":[{"id":"","pushed":0,"created":0,"origin":1529592066,"modified":0,"device":"XDK02","name":"AccelX","value":
"0.00787"}]}
]}
```


404: no matching command or devices found

423: the service is locked (admin state) or disabled (operating state)

500: unanticipated or unknown issues encountered

A DS must provide both forms (all & device-specific) of GET handlers for each command defined in one of its currently active `DeviceProfiles`.

Data transformation

A DS SDK must implement data transformation logic that converts data readings from devices/sensors to a normalized EdgeX Foundry specific schema before being forwarded to Core Data.

The values that can be read from a device are defined in the `PropertyValues` of device resources as defined by a `DeviceProfile`**.** A `PropertyValue` may specify optional attributes which are used to transform the readings from devices/sensors. These attributes are defined below and are applied to the native data types (`float*` or `int/uint*`) in the following order:

base – a base value which is raised to the power of the reading value
scale – a multiplicative factor applied to the reading
offset – an additive factor applied to the reading

Once the transformations have been applied, the DS SDK converts the native type back to a string representation prior to sending it to Core Data as a JSON `event` object containing one or more JSON `reading` objects.

Since it's possible for overflow errors to occur during data transformation the DS SDK should set the resulting string to "Overflow failed for device resource: <name> " and return a **500** status. If the **all** form of the `/command` endpoint is used an overflow will return a 200 status, and it's the responsibilty of the client to check each reading for overflows.

Assertions

Assertions are another attribute in a device resource's value `PropertyValue` which specify a string value which the result is compared against. If the comparison fails, then the result is set to a string of the form "*Assertion failed for device resource: <name>, with value: <result>*", this also has a side-effect of setting the device `operatingstate` to **DISABLED**. In the case of the single device `/command` endpoint, a 500 status code is also returned. If the **all** form of the `/command` endpoint is used an assertion failure will return a 200 status, and it's the responsibilty of the client to check each reading for assertion failures.

Mappings

Mappings are an attribute in resource operation (see *Appendix C - Device Profiles* for details) and consist of a static list of string result values that contain a fixed mapping. Ex.

["8675309": "911", "2112": "1", "777": "666"]

Mapping are applied after assertions are checked, and are the final transformation before readings are created.

***Note** - support for mappings is a nice-to-have for Dehli.*

Readings

Once transformation is complete, the DS will create a new `Reading` object, which is comprised of the `ValueDescriptor` name, the transformed value (as a string), and the device name. A `Reading`, like other objects includes an `origin` attribute which indicates the time the `Reading` was created. This time value indicates the the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. Like other objects, `Readings` also have `modified` and `created` attributes which use the same time semantics as `origin`. The `modified` attribute indicates the last time the `Reading` was modified, and the `created` attribute indicates the time the `Reading` was saved to the database (by Core Data). If a DS service wishes to convey its own timings associated with a `Reading`, then the device profile should return an `OriginTimeStamp` attribute (an optional attribute) in the associated `responseresult` JSON object (see Appendix B for more details).

Depending on the `DeviceProfile`, multiple `Readings` can be triggered simultaneously. Please refer to *core-data.raml* for more details:

https://github.com/edgexfoundry/edgex-go/blob/master/core/data/raml/core-data.raml

Events
A DS passes readings to Core Data via `Event` objects. When one or more `Readings` from a single transaction are created, a DS must generate a new `Event`, and send it to Core Data. An `Event` has a single additional field, called `device`, which is dual-purpose and can be a device name or id. Core Data use a new Core Metadata endpoint `/device/check` to determine if a device exists, first by using the given parameter as a name, and if that fails, it retries the lookup using the parameter as a device id. As such, a DS SDK should always populate the Event device field with a device name.

Scheduling

EdgeX provides an optional Scheduler support service that may be used by a DS to trigger periodic device readings. An SDK must provide helper functions (via library or package) for interacting with the Scheduler service. An SDK may also provide alternative schedule mechanisms that don't require the use of REST calls to trigger schedule events. The actual configuration of scheduling device readings is the sole responsibility of the DS implementation. A DS might choose to create a schedule for reading the same device resource for all of it's devices every minute, or it could choose to create individual schedules to read the same resource for each device it manages at different intervals, or a DS could choose not to schedule any regular readings, but instead only push readings when received asynchronously from its devices or more simply just respond to device reading queries from Core Command.

Note – see **Appendix A** for details on `Schedule` and `ScheduleEvent` configurations.

Actuation commands

A DS must handle PUT method requests (aka commands) from other services (primarily from Core Command) to set (aka Put) or actuate a device or sensor. The differences between GET and PUT methods are:

- a PUT method must provide an HTTP message body
- if there are no errors, a GET request must respond with data whereas a PUT does not have to respond with any data (except in the case of some exception/error conditions)
- a GET method implies that the client is requesting or "getting" a reading from the associated device/sensor
- a PUT method implies that the client making the request is sending or "putting" a value or some state change to the device/sensor.

An SDK must provide the following REST API PUT method endpoints:

`/device/{id}/{command}` – issue named command to the specified device

parameters:

> **id:** a database-generated device id
> **command:** the command name, defined in the device's corresponding `DeviceProfile`. The command specified must match an existing command, resource, or device resource name from the `DeviceProfile`.

body:

An application/json `cmdargs`, which is an array of key/value pairs where the keys are valid `ValueDescriptor` names**,** and provides the command arguments for each device**.**

Ex.

```
'{"VDS-CurrentTemperature": "32.52"}, {"VDS-CurrentHumidity": "1.0"}'
```

status codes:

200: returns a JSON `event` containing the `readings` as returned by the device/sensor through the DS

Example results:

```
{"id":"","pushed":0,"device":"VDC-0033","created":0,"modified":0,"origin":0,"schedule":null,"event":null,"read
ings":[{"id":"","pushed":0,"created":0,"origin":1529592066,"modified":0,"device":"VDC-0033","name":"VDS-Curren
tTemperature","value":"32.52"},{"id":"","pushed":0,"created":0,"origin":1529592066,"modified":0,"device":"VDC-
0033","name":"VDS-CurrentHumidity","value":"1.0"}]}
```

400: the provided request body is empty or contains invalid JSON

413: the provided `commandargs` object is larger than that allowed by the `MaxCmdOps` setting

404: the specified command and/or device doesn't exist

423: the device or service is locked (admin state) or disabled (operating state)

500: unanticipated or unknown issues encountered

Metadata updates

For the Dehli release a DS SDK must only support updates to a single device's adminState. All other updates from Core Metadata are ignored.

An DS or SDK must provide the following REST endpoint which supports the PUT methods for Core Metadata updates:

`/callback`

parameters:

`id:` `a database-generated device id`
`actionType``: one of the following strings representing the object to be acted upon:`

"`DEVICE`"

status codes:

200: OK

400: an valid object or parameter was specified.

Device

The only attribute of a `Device` which can be updated is `adminState`. Any attempts to change any other device attributes will result in an HTTP BadRequest (400) error.

adminState

If a device update changes `adminState` to "**LOCKED"**, the DS must return an error (status code = 423) to any subsequent REST API calls that specifiy the device other than an update which sets the `adminState` to "**UNLOCKED**".

Security

Any additional security related headers and/or tokens while not precluded by this document, are considered out of scope.

For the California release, access to DS REST APIs defined in this document will be gated by a reverse proxy service.

Appendix A - Device service configuration settings

```
[Consul]
    Host = 'localhost'
    Port = 8500

[Clients]
    [Clients.Data]
        Host = 'localhost'
        Port =

    [Client.Metadata]
        Host = 'localhost'
        Port =

[Logging]
    RemoteURL = ‘’                                  // if set to ‘’, remote logging disabled
    File = <file path>

[Service]
    Host = 'localhost'
    Port = 89999
    ConnectRetries = 3                                      // maximum number of attempts to register
                                                            // with Core Metadata
    HealthCheck = 'http://localhost:49999/api/v1/ping'
                                                            // returned from protocol logic
    OpenMsg = ‘This is the XYZ Device Service’
    ReadMaxLimit = 256                                      // maximum number of allowed results from
                                                            // REST calls to other services


                                                            // sent to Core Data if value has changed
    Timeout = 5000

    Labels = [ label1, label2, ...]

[Devices]
    DataTransform = true                                    // whether data transform is performed on
                                                            // readings or actuation commands


    Discovery = false                                       // an optional setting which can be used
                                                            // to globally disable device discovery


    InitCmd =                                               // command for device initialization
    InitCmdArgs =
    MaxCmdOps = 128                                         // maximum number of operations per command
    MaxCmdResultLen = 256                                   // maximum string length of value results

    RemoveCmd =                                             // command for device removal
    RemoveCmdArgs =
    ProfilesDir = './profile'                               // directory containing device profile
                                                            // default == './res'
    SendReadingsOnChanged = true                            // only send include readings in event

[Schedules]
  [ScheduleName1]                                           // name of the schedule
    Frequency = PT15S                                       // frequencey of the schedule
```

```
[ScheduleEvents]
  [ScheduleEventName1]                                  // name of schedule event
    Schedule = 'ScheduleName1'                          // owning schedule
    Path = /api/v1/discovery                            // REST endpoint to call when event fires

[Watchers]

  [WatcherName1]                                        // name of the provisionwatcher

    DeviceProfile =                                     // DeviceProfile of the provisionwatcher
    Key =                                               // identifier name (e.g. 'MAC', 'Name', ...)
    Identifiers = ['id1', 'id2', …]                     // whitelist of identifiers
    MatchString = '*somevalue*'                         // glob pattern used to match the given key
  [WatcherName2]
```

Appendix B - JSON schemas

For reference, here are the JSON schemas for the `responseobjects` used to pass values to the DS command endpoints used for actuation:

```
responseobjects:
'{"type":"array","$schema":"http://json-schema.org/draft-03/schema#",
"title":"responseobjects","items":{"type":"object","required":false,"$ref":"#/schemas/responseobject"}}'

responseobject:
'{"type":"object","$schema":"http://json-schema.org/draft-03/schema#",
"title":"responseobject","properties":{
"DeviceName": {"type":"string","required":false,"title":"DeviceName"},
"OriginTimeStamp":{"type":"integer","required":false,"title":"OriginTimeStamp"},
"ValueDescriptorName":{"type":"object","required":false,"title":"ValueDescriptorName"}}}'
```

For reference, here are the JSON schemas for the `events and readings`, JSON objects used to return values in respose to actuation or query REST calls to the DS command endpoints :

```
event:
'{"type":"object","$schema":"http://json-schema.org/draft-03/schema#",
"description":"Core device/sensor
event","title":"event","properties":{"id":{"type":"string","required":false,"title":"id"},"created":{"type":"i
nteger","required":false,"title":"created"},"modified":{"type":"integer","required":false,"title":"modified"},
"origin":{"type":"integer","required":false,"title":"origin"},"pushed":{"type":"integer","required":false,"tit
le":"pushed"},"device":{"type":"string","required":false,"title":"device"},"readings":{"type":"array","require
d":false,"title":"readings","items":{"type":"object","$ref":"#/schemas/reading"},"uniqueItems":false}}}'

reading: '{"type":"object","$schema":"http://json-schema.org/draft-03/schema#","description":"Core
device/sensor
reading","title":"reading","properties":{"id":{"type":"string","required":false,"title":"id"},"created":{"type
":"integer","required":false,"title":"created"},"modified":{"type":"integer","required":false,"title":"modifie
d"},"origin":{"type":"integer","required":false,"title":"origin"},"pushed":{"type":"integer","required":false,
"title":"pushed"},"name":{"type":"string","required":false,"title":"name"},"value":{"type":"string","required"
:false,"title":"value"}}}'
```

Appendix C - Device profiles

This appendix is meant to be a high-level overview of `DeviceProfiles` with the intention of describing the high-level use cases they drive with respect to readings and command actuation.

Device profiles describe the attributes of a common class of devices managed by a specific DS. A device profile is made up of one or more device resources (aka objects), optional resource commands, and top-level command definitions. All device profiles also share a common set of top-level attributes:

- `name` (required) - the device profile name; this must be unique within an EdgeX instance.
- `manufacturer` (optional) - a manufacturer name.
- `model` (optional) - a model name.
- `labels` (optional) - a list of string labels which can be used when searching device profiles.
- `description` (optional) - a short description of the device profile.
- `objects` (optional) - a list of key-value string pairs. *Not used by any existing device profile. Considered deprecated unless use case is found…*
- `deviceResources` (required) - one or more `DeviceObject` instances which represent atomic values of a device which can be read of written.
- `resources` (optional) - one or more resource commands which describe aggregate commands for reading/writing to one or more resource or device resources.
- `commands` (optional) - one or more commands which describe command data, with the purpose of supporting command introspection by clients.

DeviceObject

A `DeviceProfile` must at minimum define a single `DeviceObject`, althought typically multiple are defined. A `DeviceObject` defines an atomic resource of a device which can be read or written. The attributes of a `DeviceObject` are:

- `name` (required) - the name of the object. A `DeviceObject` name is a valid command parameter for the DS command endpoint, and is the simplest command understood by the command endpoint.
- `description` (optional) - a short description of the object.
- attributes (optional) - a list of key/value pairs describing device/protocol specific attributes (e.g. BLE `uuid`).
- `properties`:
    - `value` (required) - a list of attributes that both describe and define rules governing the value of the object. See *Appendix D - PropertyUnits, PropertyValues, and ValueDescriptors* for more details.
    - `unit` (optional) - a list of attributes describing the units associated with this object. See *Appendix D - PropertyUnits, PropertyValues, and ValueDescriptors* for more details.

Resource

A `DeviceProfile` may optionally define one or more resource objects which define commands which can aggregate one or more resouce operations. The attributes of a `Resource` object are:

- `name` (required) - the name of the resource. A resource name is a valid command parameter for the DS command endpoint.
- `get` (optional) - a list of one or more get type resource operations.
- `set` (optional) - a list of one or more set type resource operations.

`ResourceOperation`

A `ResourceOperation` defines a get or set operations for another resource or a specific device resource (`object`). A resource must specify at least one get or set resource operation. The attributes of a `ResourceOperation` are:

- `index` (required) - an integer value which should be incremented for each operation in the list
- `operation` (required) - the value `get` or `set`; should match the parent list name
- `resource` (optional) - specifies another resource name; can be used for aggregation
- `object` (optional) - specifies the name of a device resource
- `property` (optional) - required if `object` is specified; must be set to the string `value`.
- `parameter` (optional) - for a set operation, the name of the parameter that must be provided on an actuation (e.g. PUT) command endpoint call. For a get operation this must be the same as the `object` attribute.
- `mappings` (optional) - a list of key/value pairs. If a result for an operation matches one of the keys after transformation and conversion to a string value, then the result is replaced with the value specified.

Command

A DeviceProfile may optionally define one or more command objects which map one to one to a resource operation. The command objects are provided as a means for client introspection to determine required parameters for actuation commands, expected values for query commands, and possible HTTP return status codes. They are otherwise not used internally by a DS or DS SDK.

Appendix D - PropertyUnit, PropertyValue, and ValueDescriptor

This appendix describes `PropertyUnit`, `PropertyValue` and `ValueDescriptor`objects, which are a required component of a `DeviceProfiles` that describe the type of value that can be read/written to a `DeviceObject`. A `PropertyValue` describes the type of a `DeviceObject`, as well as other attributes which may be used to transform a reading.

PropertyUnit

The attributes of a `PropertyUnit` are:

- `type` (required) - a string indicating the type pf unit, currently the only value supported is "`String`".
- `readWrite` (required) - a string value indicating if the unit can be read ("`R`"), written ("`W`"), or both ("`RW`"). Currently on "`R`" is supported.
- `defaultValue` (required) - a string which describes the measurement units associated with a `PropertyValue`. Examples include "deg/s", "degreesFarenheit", "G", or "% Relative Humidity".

PropertyValue

The attributes of a `PropertyValue` are:

- `type` (required) - describes the native type of a `DeviceObject` (aka device resource). Values are passed from a DS implementation to the SDK as native types, and then transformed by the SDK and serialized as strings before being pushed to Core Data. The `type` attribute is also used to populate the associated `ValueDescriptor` `type` attribute. The supported native types, which map easily to C, Go, or other languages are:

  ```
  bytes | bool | float32 | float64 | int8-64 | string | uint8-64
  ```

- `readWrite` (required) - a string value indicating if the unit can be read ("`R`"), written ("`W`"), or both ("`RW`").
- `minimum` (optional) - *not used in Dehli*
- `maximum` (optional) - *not used in Dehli*
- `size` *(optional)* -
- `precision` *(deprecated)*
- `word` - *not used in Dehli*
- `LSB` *(deprecated) - not used in Dehli*
- `mask` - *not used in Dehli*
- `shift` - *not used in Dehli*
- `base` - a string-based numeric value which is raised to the power of the reading value
- `scale` - a string-based numeric multiplicative factor applied to the reading
- `offset` - a string base numeric additive factor applied to the reading
- `assertion` -a string based value which is compared to a reading post-transform. If the values do not match, then the result is replaced the string "Assertion failed with value: <post-transform-result>".

ValueDescriptor

A `ValueDescriptor` is a Core Data object which is created by a DS SDK while importing device profiles. A `ValueDescriptor` maps to a single device resource within a device profile, although other device profiles specifying a device resource with the same name, will end up sharing the associated `ValueDescriptor`.

The attributes of a `ValueDescriptor` are:

- `id` - a unique id assigned to the `ValueDescriptor` on creation

- `created` - an `int64` value which indicates when the `ValueDescriptor` was saved to persistent storage. See the Data transformation - Readings sub-section for more details on the time format.
- `description` -
- `modified` - an `int64` value which indicates when the `ValueDescriptor` was last modified. See the *Data transformation - Readings* sub-section for more details on the time format.
- `origin` - an `int64` value which indicates when the ValueDescriptor was created. See the Data transformation - Readings sub-section for more details on the time format.
- `name` - a unique name of the ValueDescriptor, taken from the name of the first `PropertyValue` which triggered creation of the `ValueDescriptor`.
- `min` - a string based numeric value representing a minimum value for the reading associated with the `ValueDescriptor`. *Not used in Dehli.*
- `max` - - a string based numeric value representing a maximum value for the reading associated with the `ValueDescriptor`. *Not used in Dehli.*
- `defaultValue` - a string value representing the units associated with a reading associated with the `valueDescriptor`.
- `type` - a string value which indicates the type of a transformed reading associated with this ValueDescriptor, taken from the type of the first `PropertyValue` which triggered creation of the `ValueDescriptor`.
- `uomLabel` -
- `formatting` - a string representing how to format the result; uses printf conventions. Not currently used.
- `labels` - one or more strings which may be used to search for the `ValueDescriptor`.