EdgeX Foundry
Fuji
Device Service Functional Requirements

Version: 10
June 25, 2019

Tony Espy <espy@canonical.com>
Jim White <james.white2@dell.com>
Iain Anderson <iain@iotech.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 solely 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. An SDK may provide relevant convenience methods and 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/api/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 Metadata. If a matching instance is found, a DS should update the instance to reflect current configuration. 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.

***Note** - the following subsection is considered out of scope for the Fuji release.*

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 command is 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).

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. These identifier may be fully-qualified, in which case each ProvisionWatcher instance matches a specific device, or they may include glob characters, allowing a single ProvisionWatcher to match multiple devices.

A second, optional, identifier map may exist in the `ProvisionWatcher` and if it is present, devices which are matched by the primary map are subsequently compared against it. Matching on the secondary map results in the new device being ignored; this allows the matching scope of the `ProvisionWatcher` to be narrowed. If a device is matched on a `ProvisionWatcher` 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. The `ProvisionWatcher` instance also carries an `AdminState` value which is applied to the new `Device` instance.

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 bootstrap 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 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 overridden 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. If the registry is enabled but contains no settings for the DS, the DS should read its configuration from TOML and store them in the registry for future use.

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 either log an error, and the change will not be applied until the DS is restarted, or it can 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 its pre-existing device service object in Core Metadata, it must update that object 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, specifically Description, Labels, Host and Port.

Device and DeviceProfile management

An SDK must provide facilities which allow a DS to create, modify, query for and delete Device instances. In addition, the ability to query for Device Profiles, and to create them (for example, from .yaml files) must be provided.

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 new devices being automatically added. Discovery is controlled 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 or otherwise. 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, the device is added using the `DeviceProfile` and `AdminState` settings found in the `ProvisionWatcher`.

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 unanticipated 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 DS's version string; indicates that the service is available

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

System Management support

A DS must implement the REST API endpoints required by the EdgeX system management service.

Configuration check

`/config`

status codes:

200: return the current DS configuration

Example response:


```
{"Clients":{"Data":{"Host":"edgex-core-data","Port":48080"},"Metadata":{"Host":"edgex-core-metadata",
"Port":48081"}},"Logging":{"EnableRemote":false,"File":"-"},"Service":{"Timeout":5000,"CheckInterval":"10s",
"Host":"edgex-device-template","Port":49990,"StartupMsg":"Example template device started","ReadMaxLimit":256,
"ConnectRetries":3,"Labels":["Template"]},"Device":{"DataTransform":true,"Discovery":false,"MaxCmdOps":128,
"MaxCmdResultLen":256,"SendReadingsOnChanged":true}}
```


Metrics

`/metrics`

status codes:

200: return a list of metrics for the running DS

Example response :

{“Memory”:{"Alloc":2420616,"TotalAlloc":2420616,"Sys":6949112,"Mallocs":30506,"Frees":5084,"LiveObjects":25422},"CpuBusyAvg":2.224}

Note - minimum metric availability for the Edinburgh release will be a summary of heap and CPU usage. This can be obtained from the runtime in Go implementations, as shown above. In C, the ability to provide similar information will depend on the operating system and C library in use.

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`. The default behavior of the SDK if one or more readings to be pushed to Core Data contain binary values, is to encode the entire message payload (i.e. an event including one or more readings) as CBOR encoded payload instead of JSON.

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 by default a base64 encoded little-endian of the float32 or float64 value, but the “`floatEncoding`” attribute relating to the value may instead specify “`eNotation`” in which case the representation is a decimal with exponent (eg “1.234e-5”)

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 Fuji 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 endpoints to support device readings:

`/device/{id}/{command}` - issue named query command to the specified (by id) device/sensor
`/device/name/{name}/{command}` - issue named query command to the specified (by name) devices/sensor

parameters:

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

***Note** - the following subsection is considered out of scope for the Fuji release.*

Command Header

Callers should supply an *Accept* header and indicate a supported MIME type. If none is supplied, the SDK shall assume application/json by default, except in the case where one or more Readings is typed as Binary, in which case application/cbor will be the default.

Command Response

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 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`.

Response Header

The SDK should include with its response to a query command, at least the following headers:

- Content-Type

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
mask – a mask with which the reading should be bitwise and-ed
shift – a number of bits by which the reading should be shifted right

Note that the mask and shift operations apply only to `uint*` types, and should not be used together with base, scale or offset.

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 responsibility 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 responsibility 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.

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 difference, measured in nanoseconds, between the current time and midnight, January 1, 1970 UTC. Like other objects, `Readings` also have `modified` and `created` attributes which use similar time semantics as `origin` but are expressed in milliseconds. 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).

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/api/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 uses 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 to trigger periodic device readings. An SDK should also support the Automatic Events mechanism which does not require the use of REST calls to trigger such readings. A device may have a number of `AutoEvents` associated with it; the `AutoEvent` contains the following fields:

resource: the name of a deviceresource or resourcecommand indicating what to read.
frequency: a string indicating the time to wait between reading events, expressed as an integer followed by units of ms, s, m or h.
onchange: a boolean: if set to true, only generate new events if one or more of the contained readings has changed since the last event.

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

parameters:

id: a database-generated device id *or*
name: the name of the device.
command: the command name, defined in the device's corresponding `DeviceProfile`. The command specified must match an existing 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: The PUT operation was successful.

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

Data transformation

The device service should apply transformations on parameters in the reverse fashion that it does for readings, ie the order of application is reversed, **`offset`** becomes negative, **`scale`** should result in a division and **`base`** requires a calculation of the form `value = log(parameter)/log(`**`base`**`)`.

mask is a special case which allows setting or clearing one or more bits while leaving the rest of the original value untouched. The value written to the device should be `(currentvalue AND (NOT `**`mask`**`)) OR (parameter <<` **`shift`**`)`.

Minimum and Maximum

Minimum and Maximum are attributes in a device resource's value `PropertyValue` which may be used to specify an allowable range for numeric actuation parameters. A device service should validate parameters against these before applying any transformations. An out-of-range request should result in a **400** error.

Mappings

Mappings may be defined for actuation commands just as they are for queries, and if present should be the first transformation applied.

Metadata updates

A DS or SDK must provide the following REST endpoint which supports the PUT, POST and DELETE 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"
"PROFILE"
"PROVISIONWATCHER"

status codes:

200: OK

400: an invalid object or parameter was specified.

The device service is notified of relevant additions (POST), modifications (PUT) and deletions (DELETE) of devices, device profiles and provision watchers, by the core-metadata service. It should update its internal state accordingly.

Device 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 specify the device other than an update which sets the `adminState` to "**UNLOCKED**".

Shutdown

If a device service is configured to use the Registry, it should unregister itself as part of its shutdown process.

Security

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

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

Appendix A - Device service configuration settings

***Note** - Init/RemoveCmd[Args] and SendReadingsOnChanged are considered out of scope for the Edinburgh release.*

```
[Clients]                                           // Only used if the registry is not enabled
    [Clients.Data]
        Host = “localhost”
        Port = 48080

    [Clients.Metadata]
        Host = “localhost”
        Port = 48081

    [Clients.Logging]
        Host = “localhost”
        Port = 48061

[Logging]
    EnableRemote = false                            // set true to use support-logging service
    File = <file path>                              // set to ‘-’ for console output
    LogLevel = “DEBUG”

[Service]
    Host = “localhost”
    Port = 89999
    ConnectRetries = 3                              // maximum attempts to connect to other services
    Timeout = 5000                                  // and time to wait between them
    StartupMsg = “This is the XYZ Device Service”   // message to log on startup
    Labels = [ label1, label2, ...]
    CheckInterval = “30s”                           // interval at which the registry should
                                                    // check our liveliness

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


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


    MaxCmdOps = 128                                 // maximum number of operations per command
    MaxCmdResultLen = 256                           // maximum string length of value results

    InitCmd =                                       // command for device initialization
    InitCmdArgs =
    RemoveCmd =                                     // command for device removal
    RemoveCmdArgs =

    ProfilesDir = “./profile”                       // directory containing device profiles
                                                    // default == './res'

[[Watchers]]
    Name = “Provisioner1”                           // name of the provisionwatcher
    DeviceProfile = “WidgetABC”                     // DeviceProfile of the provisionwatcher
    AdminState = “UNLOCKED”                         // Initial AdminState to apply
    [[Watchers.Identifiers]]
        Key = “MAC”                                 // identifier name (e.g. 'MAC', 'Name', ...)
        MatchString = “*somevalue*”                 // glob pattern used to match the given key
```

Appendix B - JSON schemas

For reference, here is the JSON schema for the `setting` object used to pass values to the DS command endpoints used for actuation:

```
   setting:
'{"type":"object","$schema":"http://json-schema.org/draft-06/schema#","title":"setting","additionalProperties"
:{"type":"string"}}'
```

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

```
   events:
'{"type":"array","$schema":"http://json-schema.org/draft-06/schema#","title":"events","items":{"type":"object"
,"required":false,"$ref":"#/schemas/event"}}'

   event:
'{"type":"object","$schema":"http://json-schema.org/draft-06/schema#","title":"event","properties":{"device":{
"description":"the name of the device","type":"string"}, "origin":{"description":"timestamp indicating when
the readings were taken","type":"integer"},"readings":{"type":"array","items":{"type":"object",
"$ref":"#/schemas/reading"}}}}'

   reading:
'{"type":"object","$schema":"http://json-schema.org/draft-06/schema#","title":"reading","properties":{"name":{
"type":"string"},"value":{"type":"string"}}}'
```

Appendix C - Device profiles

This appendix is meant to be a high-level overview of device profiles 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, optional resources, and optional commands. Commands are added to Core Metadata, and are used to validate REST API requests made to Core Command's **command** endpoint.

Note - there's a single namespace for device resource names across a single EdgeX instance.

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
- `DeviceResources` (required) - one or more device resources which represent discrete values of a device which can be read or written
- `Resources` (optional) - one or more resource (commands), each of which contains two arrays of resource operations, one for read (get) commands, and one for write (put) commands
- `Commands` (optional) - one or more commands, each of which describe the expected values, parameter names, and possible HTTP response codes for each command supported by the Core Command service's **/command** REST endpoint. In essence, this section gives definition to the device service's public API as supported by Core Command.

DeviceResource

A device profile must at minimum define a single device resource, although typically multiple are defined. A device resource defines a discrete value of a device which can be read or written. The attributes of a device resource are:

- `Description` (optional) - a short description of the device resource
- `Name` (required) - the name of the device resource
  A device resource name is a valid command parameter for the DS **/device** endpoint, and thus is the simplest command understood by the device service's **device** REST endpoint.
- `Tag` (optional) - an optional string which can be used for categorization of device resources
- `Properties` - two maps of properties that describe the device resource:
  - `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.
- `Attributes` (optional) - a list of key/value pairs describing device/protocol specific attributes (e.g. BLE `uuid`)

DeviceCommand

A device profile may optionally define one or more device commands which can aggregate one or more resource operations which most of the time perform reads or writes to device resources. The attributes of a device command are:

- `Name` (required) - the name of the command
  A command name is a valid command parameter for the device service's **/device** endpoint.
- `Get` (optional) - a list of one or more read operations
- `Set` (optional) - a list of one or more write operations

Note - which list gets used is determined by the HTTP method of the incoming request to the **/device** endpoint. If the request method is GET, then the list of get operations is used, if the method is PUT, then the set operations are used. Although in the past releases based on the original Java version of EdgeX it was possible to include get operations in the set list and visa versa, this is no longer supported as of the Fuji release.

ResourceOperation

A resource operation defines a get or set resource operations that each specify a device resource or another resource (command). A resource must specify at least one get or set resource operation. The attributes of a resource operation are:

- `Index` (required) - an integer value which should be incremented for each operation in the list
- `Operation` (required) - the value **`get`** or **`set`**
- `DeviceCommand` (optional) - specifies another DeviceCommand name; can be used for aggregation. A resource operation must either specify a DeviceCommand or a DeviceResource, which are mutually exclusive.
- `DeviceResource` (optional) - specifies the name of a device resource. A resource operation must either specify a DeviceCommand or a DeviceResource, which are mutually exclusive.
- `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.

CoreCommand

A device profile may optionally define one or more commands which map one to one to resource (commands). Commands are used by the Core Command service to define which command names are accepted by it's **/command** REST endpoint. Core Command provides REST endpoints for clients to query commands supported per device profile. The results of these queries include may include 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.

Note - like with device resources, there's a single namespace for "commands" across a single EdgeX instance.

Appendix D - PropertyUnit, PropertyValue, and ValueDescriptor

This appendix describes property unit and property values which are components of a device resource. It also describes value descriptors which are Core Data objects derived from property values.

PropertyUnit

The attributes of a property unit are:

- `Type` (required) - a string indicating the type of 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 property value
  Examples include "deg/s", "degreesFarenheit", "G", or "% Relative Humidity".

PropertyValue

The attributes of a property value are:

- `Type` (required) - describes the native type of the 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 value descriptor `type` attribute. The supported native types, which map easily to C, Go, or other languages are:

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

- `ReadWrite` (required) - a string value indicating if the value can be read ("`R`"), written ("`W`"), or both ("`RW`")
- `Minimum` (optional) - A minimum permitted value for a numeric actuation parameter
- `Maximum` (optional) - A maximum permitted value for a numeric actuation parameter

- *DefaultValue (optional) - not used*
- MediaType - a string value used to indicate the type of binary data if *Type=binary*
- FloatEncoding (optional) - a string value which indicates the type of encoding used for floating point values (**float32**|**float64**). If set to **eNotation**, the floating point value is encoded using the standard C-style printf floating point notation (i.e. "%f"), with enough numerals to preserve precision. If set to **base64** (or by default) the floating point value is encoded using base64.
- Mask (optional) - a string-based mask to be applied to the (integer) reading
- Shift (optional) - a string-based shift factor applied to the (integer) reading
- Scale (optional) - a string-based numeric multiplicative factor applied to the reading
- Offset (optional) - a string-based numeric additive factor applied to the reading
- Base (optional) - a string-based numeric value which is raised to the power of the reading value
- Assertion (optional) - 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>".
- Precision *(deprecated) - **should be removed***