IoT Plug & Play
Modeling and Architecture

IoT Plug and Play

Modeling

Digital Twin Definition Language (DTDL)

- Language for describing models and interfaces for an IoT digital twin.
- Open source based on open standards (JSON-LD, RDF).
- Made up of a set of metamodel classes:
  - Two top-level classes, CapabilityModel and Interface
  - Three metamodel classes that describe capabilities: Telemetry, Property and Command.
- Provides semantic type annotations of capabilities.
- Use of the JSON-LD context (the @context statement) to specify the version of DTDL being used.

Key IoT Plug and Play concepts

Device Capability Model

A CapabilityModel describes a device and defines the set of interfaces implemented by the device.

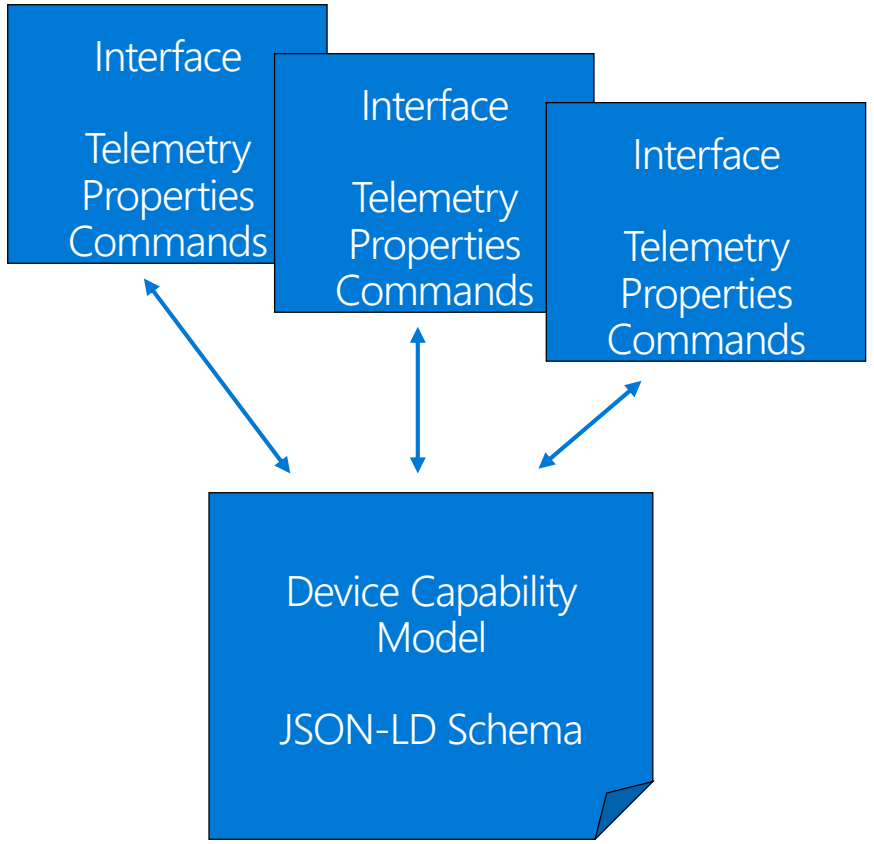
A capability model includes the identifiers of the interfaces that it implements (including the version number)

Interface

A shared contract that uniquely identify the capabilities exposed by a device

Expressed as Properties, Telemetry, and Commands

Interfaces are reusable across different devices and models



Digital Twin Description Language github: DTDL

Device Capability model – authoring consideration (rules)

- A capability model can only implement one instance of each interface.
- A capability model can only implement one version of each interface. A capability model cannot implement two versions of the same interface.
- A newer version of a capability model must include all the interfaces implemented by the previous version

Device Capability model – authoring consideration (properties)

- Required
  - @id - An identifier for the capability model that follows the digital twin identity format.
  - @type - The type of capability model instance.
  - @context - The context to use when processing this capability model.
  - implements - A set of capability model interfaces.

- Optional
  - Comment - A developer comment.
  - Description - A localizable description for human display.
  - displayName - A localizable name for human display.

Device Capability model – example 1


```
{
   "@id": "urn:example.com:thermostat_T_1000:1",
   "@type": "CapabilityModel",
   "displayName": "Thermostat T-1000",
   "implements": [
      {
         "name": "thermostat",
         "schema": "urn:example:thermostat:1"
      },
      {
         "name": "urn_azureiot_DeviceManagement_DeviceInformation",
         "schema": "urn:azureiot:DeviceManagement:DeviceInformation:1"
      }
   ],
   "@context": "http://azureiot.com/v1/contexts/IoTModel.json"
}
```

Capability Model Interface ("implements" section)

- A Capability Model Interface describes a part of a capability model.
- Capability Model Interface required properties
  - name - The "programming" name of the capability model interface.
  - schema - The interface implemented by the capability model.

Device Capability model – example 2


```
{
  "@id": "urn:example:thermostat_T_1000:1",
  "@type": "CapabilityModel",
  "displayName": "Thermostat T-1000",
  "implements": [
    {
      "name": "thermostat",
      "schema": {
        "@id": "urn:example:thermostat:1",
        "@type": "Interface",
        "displayName": "Thermostat",
        "contents": [
        ],
        "@context": "http://azureiot.com/v1/contexts/IoTModel.json"
      }
    },
    {
      "name": "urn_azureiot_DeviceManagement_DeviceInformation",
      "schema": "urn:azureiot:DeviceManagement:DeviceInformation:1"
    }
  ],
  "@context": "http://azureiot.com/v1/contexts/IoTModel.json"
}
```

Interface – authoring consideration

- An Interface describes related capabilities that are implemented by a device or digital twin.
- Interfaces are reusable and can be reused across different capability models.
- Interface properties
  - Required
    - @id - An identifier for the interface that follows the digital twin identity format.
    - @type - The type of interface object (must refer to the "Interface" metamodel class).
    - @context - The context to use when processing this interface.
  - Optional
    - contents - A set of objects that describe the capabilities (telemetry, property, and/or commands) of this interface.
    - …

Interface - example

```
{
   "@id": "urn:example:thermostat:1",
   "@type": "Interface",
   "displayName": "Thermostat",
   "contents": [
      {
         "@type": "Telemetry",
         "name": "temp",
         "schema": "double"
      },
      {
         "@type": "Property",
         "name": "setPointTemp",
         "writable": true,
         "schema": "double"
      }
   ],
   "@context": "http://azureiot.com/v1/contexts/IoTModel.json"
}
```

Interface - Telemetry

- Telemetry describes the data emitted by a device or digital twin
  - a regular stream of sensor readings
  - or an occasional error
  - or information message.
- "Telemetry" properties
  - Required
    - @type - The type of telemetry object.
    - name - The "programming" name of the telemetry.
    - schema - The data type of the telemetry.
  - Optional
    - unit - The unit type of the telemetry.
    - ..

Telemetry - example

```
{
   "@type": "Telemetry",
   "name": "temp",
   "schema": "double",
   "unit": "celsius"
}
```

Interface - Property

- A Property describes the read-only and read-write state of a device or DT.
  - a device serial number may be a read-only property
  - the temperature set point on a thermostat may be a read-write property.
- "Property" properties
  - Required
    - @type - The type of Property object.
    - name - The "programming" name of the Property.
    - schema - The data type of the Property.
  - Optional
    - writable - A boolean value that indicates whether the property is writable or not. The default value is false (read-only)
    - ..

Property - example

```
{
   "@type": "Property",
   "name": "setPointTemp",
   "schema": "double",
   "writable": true
}
```

Interface - Command

- A command describes a function or operation that can be performed on a device or digital twin.
- "Command" properties
  - Required
    - @type - The type of Command object.
    - name - The "programming" name of the Command.
  - Optional
    - commandType - The type of command execution, either synchronous or asynchronous. The default value is synchronous.
    - request - A description of the input to the command.
    - response - A description of the output of the command.

Command – Example


```
{
   "@type": "Command",
   "name": "reboot",
   "commandType": "asynchronous",
   "request": {
      "name": "rebootTime",
      "displayName": "Reboot Time",
      "description": "Requested time to reboot the device.",
      "schema": "dateTime"
   },
   "response": {
      "name": "scheduledTime",
      "schema": "dateTime"
   }
}
```

DTDL - Schemas

- Schemas describe the on-the-wire or serialized format of the data in a digital twin interface.
- A full set of primitive data types are provided, along with support for a variety of complex schemas in the forms of Arrays, Enums, Maps, and Objects.
- compatible with popular serialization formats, including JSON, Avro, Protobuf, and others.

Primitive schemas

Digital twin primitive schema	Description
boolean	A boolean value.
date	A date in ISO 8601 format.
datetime	A date and time in ISO 8601 format.
double	An IEEE 8-byte floating point number.
duration	A duration in ISO 8601 format.
float	An IEEE 4-byte floating point number.
integer	A signed 4-byte integer.
long	A signed 8-byte integer.
string	A UTF8 string.
time	A time in ISO 8601 format.

Complex schemas - Arrays

- An Array describes an indexable data type where each element is of the same schema.
- The schema of an array element can itself be a primitive or complex schema.
- Array required properties
  - @type - The type of array object.
  - elementSchema - The data type of the array elements.
- Array example

```
{
  "@type": "Telemetry",
  "name": "ledState",
  "schema": {
    "@type": "Array",
    "elementSchema": "boolean"
  }
}
```

Complex schemas - Enum

- An Enum describes a data type with a set of named labels that map to values.
- Enum required properties
  - @type - The type of enum object.
  - enumValues - A set of enum value and label mappings.
  - valueSchema - The data type for the enum values.

Enum example

```
{
  "@type": "Telemetry",
  "name": "state",
  "schema": {
    "@type": "Enum",
    "valueSchema": "integer",
    "enumValues": [
      {
        "name": "offline",
        "displayName": "Offline",
        "enumValue": 1
      },
      {
        "name": "online",
        "displayName": "Online",
        "enumValue": 2
      }
    ]
  }
}
```

Complex schemas - Object

- An Object describes a data type made up of named fields (like a struct in C).
- The fields in an object map can be primitive or complex schemas.
- Object required properties
  - @type - The type of object.
  - fields - A set of field descriptions, one for each field in the object.
- "fields" required properties
  - Name - The "programming" name of the field.
  - Schema - The data type of the field.

Object - example

```
{
   "@type": "Telemetry",
   "name": "accelerometer",
   "schema": {
      "@type": "Object",
      "fields": [
         {
            "name": "x",
            "schema": "double"
         },
         {
            "name": "y",
            "schema": "double"
         },
         {
            "name": "z",
            "schema": "double"
         }
      ]
   }
}
```

Model versioning - consideration

- In DTDL, capability models and interfaces are versioned by a single version number (positive integer) in the last segment of their identifiers.

- DTDL provides two ways to create new versions of capabilities models and interfaces.
  1. For major changes, entirely new capability models and interfaces can be created
  2. For minor changes, new versions of capability models and interfaces can be created

Model authoring - Additional concerns

- Digital Twin identifier format
  - A valid identifier has at least four segments.
    - The "urn" segment.
    - A namespace segment. This segment may be made up of one or more segments.
    - The name segment (second-to-last segment).
    - The version segment (last segment).
  - Example

urn:nivasseu:GPSTracker:MXChip2000:3

Model authoring - Additional concerns

- Display string localization

```
{
  "@id": "urn:example:thermostat:1",
  "@type": "Interface",
  "displayName": {
    "en": "Thermostat",
    "it": "Termostato"
  }
}
```

- Context
  - When writing a digital twin definition, it's necessary to specify the version of DTDL being used.
  - For this version of DTDL, the context is exactly http://azureiot.com/v1/contexts/IoTModel.json.

IoT Plug and Play

Architecture overview

Architecture Overview