English

Smart API Core ontology model

ASEMA

Table of Contents

1. Introduction

1.1. Smart API design philosophy

Smart API is a semantic data model for the Internet of Things (IoT) applications. Such applications typically are built with sensoring and controlling devices ranging from very small and limited power microcontroller-driven devices to embedded Linux gateways and full scale cloud platforms. The application area is broad and the uses and devices very diverse. It is therefore no wonder that the industry has traditionally suffered from a lacking consensus of how data should actually be transmitted for it to be at least somewhat uniform and compatible.

Semantics on the other hand is a field of computer science that tries to bring meaning to data. Instead of just defining what data is, semantics strives to also define what the data stands for. As modern systems push towards more advanced data processing — commonly currently under the umbrella of Artificial Intelligence (AI) — this meaning is of critical importance in automatically processing of the vast amounts of data IoT can produce. Therefore, including semantic thinking into the data processing is vital for making the design — and the data — future-proof.

Having worked with several data design initiatives and various organizations, we've actually found that the concept of IoT is quite hard to grasp from a design point of view. It combines two broad concepts, "Internet" and "Things", and deciding which one is which — and whether there is a reason for making a meaningful difference between them in the first place — is actually quite tricky. The reason for highlighting this discrepancy is that IoT applications seldom, if ever, exist as an isolated island. Quite the contrary, in order for IoT to be valuable in anything, it almost always needs to be combined with some other systems. Due to this, the design tends to deviate from one side to the other and separating "IoT design" from the rest of the design of an application is hard.

To illustrate, let's take a couple of examples. The first one is a classical example of an IoT industry branch: preventive maintenance. It analyzes the failure rate of two motor designs by measuring the oil pressure of remotely operated sample engines with an IoT pressure sensor. So the oil pressure data is our core data for analysis. In this particular use case, it has been decided that if the oil pressure suddenly drops during operation, we conclude that the motor has broken down due to a leak. And we want to find the engine design with the smallest probability of leaks.

Now, let's imagine we have one year's worth of data of two motors, both with their own design. In this one year period, motor 1 has had three occasions when the oil pressure dropped, i.e. the motor became faulty. In the same period, motor 2 failed only once. Based on this data, motor design 2 with only one breakdown is three times more reliable than motor 1 and therefore has a superior design.

Is that the whole truth? Well … no. Looking at another dataset, the usage hours of those motors, we find that motor 1 was used 300 days during the year, 10 hours each day, 3000 hours in total. So it broke once per 1000 hours of operation on average. Motor 2 was started once. It absolutely obliterated itself after 10 minutes and was put back into the warehouse with a sticker "broken" on it and was never taken out again. That's a failure rate of 100%.

Raw IoT data seldom means anything. Only correlation between IoT data and some other data brings business insights. I this case, oil pressure data is quite misleading unless we have the usage hours. Keeping that in mind, the next question is whether IoT data is somehow different from any other data and when is it so? So let's take another example.

In our second example we're measuring the efficiency of window sealants. The aim is to see how well the windows keep the temperature inside a house constant as the temperature outside changes. Now we have a more traditional IoT compatibility case. Let's imagine that the temperature outside the window is measured with one sensor by one manufacturer. The temperature inside is measured with another sensor from another manufacturer. A cross-correlation is run between the two measures. If there is high cross-correlation, i.e. the temperature inside moves in lockstep with the temperature outside, the sealant is poor. If the cross-correlation is low, the sealant is efficient.

Now this is where standardized data would be very useful. Two hardware manufacturers, two sets of data, one data format for temperature measurement hardware. But that is not how real-life tests work. Tests like these are often rolled out into a heterogeneous environment where for instance some houses have a fancy wireless weather station that transmits readings of the outside temperature. Some don't. But no problem, says the engineer. Instead of data from the weather station, we could take readings from the website of the local weather service provider. It might not be absolutely perfect data and the difference in temperature compared to actual may be a degree or two. But that does not matter. To simply judge whether a sealant leaks or not, the accuracy is just fine.

So now we have two sets of data: one from a website and one from a device. From the viewpoint of our data analysis application, what is the difference? How does data from the "Internet" service differ from data from the "Thing" sensor? Is there an actual difference between "Internet" and "Things"? Usually not. This is why most designs that confuse these two become eventually pointless when applied to real applications.

Finally, as our last example let's consider the concept of remote control, a critical part of a wide range of IoT systems. A remote controller is typically called an "actuator". But what is an actuator actually? If a door opens and the opening is the result of a relay at the top of the door mechanism latching, which one is the actuator? The door or the relay? If it is the relay and a design does not include the relay, can we control the door at all? In programming terms: Can we say `door.close()` without being able to say `relay.toggle()`? If controlling some mechanism would mean that we need to know each and every component of a mechanism in order to use them as actuators, then that application would become overly complex. No-one would go through the trouble of providing all the needed blueprints.

So it becomes evident that a good design for IoT actually also caters for the needs of other Internet-based applications for data exchange and control. It also builds on levels of abstraction where one developer sees just a relay in a door, another a door operated by the relay, and the third developer a building with multiple doors. Bringing such understanding to the design is one of the key targets of Smart API.

1.2. Principles of the Smart API design

Designs are used by application software, and software gets programmed by programmers. In the two main IoT applications, that 1) measure, i.e. sensor some physical phenomenon and 2) control, i.e. actuate some physical device, there actually should be no difference between "Internet" and "Things". Both can be virtual or physical, actual or fictional, directly connected with a wire or behind a ton of routers, gateways and firewalls. Those make no difference. This is, in a nutshell, the whole point of Internet of Things.

This is actually the key value proposition of IoT: making remote and autonomous operation of devices possible due to the shared (and mostly free) Internet infrastructure. What used to be an expensive control connection with dedicated wires is now a low-cost connection using the shared Internet infrastructure. What used to have a dedicated control software application with a hefty license cost is now just a browser interface.

The design of Smart API attempts to achieve a holistic view on the data that would fit to the needs of particular IoT applications while taking into account the core value proposition of the overall IoT concept. It needs to serve the multiple needs of various applications while being semantically processable and easily mapped into actual software.

That being said, let's see how the philosophy applies to the actual design.

At its core, Smart API borrows a lot from object-oriented programming. Everything in Smart API is just an object (not even a "thing"). And object can have:

- Properties, i.e. what that object is.
- Abilities, i.e. what that object can and cannot do.

These are really easy to map into software terms. Properties are variables and abilities are methods. Creating an IoT object in software that actually controls or measures something is easy. And so if you want to measure something, you `get` a property. If you want to control something, you `set` a property. There is actually no need for explicitly modeling sensors or actuators. If a property can be read, it is a sensor, if a property can be written, it is an actuator. A read-write is an actuator with sensoring ability.

Next, those properties and abilities must be expressed in a uniform fashion. In Smart API each property of an object, i.e. a data item, is marked with the following variables: a) quantity, b) unit, c) datatype.

When applied to the object modelm this means that: a) everything is modeled as an object and b) each measurable property of that object has quantity, unit and datatype and c) sensoring means reading that property and controlling means writing that property. This simple design covers the basic functionality of all sensoring and controlling applications in one standard fashion.

How about change management then? The systems are not static and neither is the data they manage. Well, once everything is an object, then the good old CRUD-N thinking kicks in very well. Originating from database design, CRUD defines the four basic functions of a persistent storage: Create, Read, Update and Delete. "N", or Notify, is an addition from the autonomous word. An operation that is a signal that something like a change took place.

We've already gone through two of those letters in the acronym: (R)eading properties (sensoring) and (U)pdating properties (controlling). Changes happen when someone (C)reates a property (or a capability) or (D)eletes a property (or a capability). And if you make an autonomous sensoring device (tag, beacon, etc.), when it sends values without a specific read request, that is a (N)otification. CRUD-N therefore covers controlling and sensing in flexible way in situations of change, both with autonomous and non-autonomous IoT applications.

The final piece of property design is the time aspect. When Smart API defines IoT properties, there can be a time span associated with all of them. What the property has been (past), what it should be now (present), what we'd like it to be later (future). These are three time frames of CRUD-N operations; we call them recording (past), setting (present) and targeting (future). And a time can of course refer to a point in time (an instant) or a period of time (the so called Temporal Context).

1.3. Predefined Smart API models

While the object-property modeling tells **how** to model a particular object, it also often pays off to define **what** to model. This reduces administrative time spent on implementing something. While the core of the Smart API model says you can attach any property onto an object (as long as a quantity, unit and datatype is defined), there are several good reasons to standardize how particular things are attached. Predefined property models make communication easier. Also, they are a way to spread industrial knowledge. After all, those who actually implement the systems are software engineers. They are first and foremost specialists of the software engineering domain, not the specific industry domain. So if the industry has a reference model for properties, this makes life in software engineering much easier.

In Smart API such industry specifications are extensions to the core model. Whether an extension is included or not in no way affects whether the core is usable and acceptable. As an example, the Smart API SDK package implements a reference model for physical things. After all, most IoT systems handle physical objects so they all can easily share this spec.

Luckily, a brief look at basic Newtonian physics gives most of the guidelines on how to do a psysical model. Every physical thing has a location in a 3-dimensional space: latitude, longitude and altitude. It also has an orientation: yaw, pitch and roll. It has a weight and some dimensions: height, width and depth (they also have a shape but that is surprisingly rarely interesting). Each of those measures has a past, present and future (e.g. path, location and route) and at least first level deltas (e.g. distance, speed, acceleration). Once those are in, it is easy to model almost any application involving physical things, whether it is a warehouse monitoring system or a system that tracks city bikes.

Of course, a basic extension is never enough for specific applications. Each industry application will have its own requirements for the properties which may range from basic and common things such as temperature or humidity to weird and wonderful ones like permeability, bounciness, or the stickiness of the glued surface at the lower left corner of the related container box. Needless to say, such things cannot be predicted nor defined beforehand so the model needs to be extendable by its users.

Next, connectivity. IoT applications deal with technical issues such as connection interfaces, ranging from physical (i.e. serial, Bluetooth), network (i.e. IP address, port), protocol (e.g. TCP, UDP), messaging protocol (e.g. HTTP, MQTT, CoAP, 0MQ) to managerial (connects to, is managed by). These are actually quite straightforward to do. The tricky part is **where** to attach each interface. Remember the example of the door with the relay that closes it? Now, let's imagine making an advanced model of that door: a security door. In addition to the closing mechanism, it also has a camera. Using that door in an application would mean something like `door.showVideoStream(); if (person on video is known) door.open(); else door.lock();` For convenience, the video stream is a door property of the bytestream type, mimetype MPEG4. The door open/close mechanism is a boolean property. Are they exposed through the same interface? Well, no. The good old industrial relay uses ModBus. Can ModBus transmit video stream? It cannot. For this purpose the door camera has WiFi. So while the door does have two interfaces, ModBus and WiFi, this information is not enough as we would not know which interface serves which property (i.e. "can we close the door by accessing the WiFi"). Therefore interfaces must be linked to properties.

While technical networking connections are one puzzle to solve, in many real life roll-outs, those issues are still trivial compared to connecting and aligning the policies of organizations where the systems run. Who has the authorization to give access to data? What are the rights to the data afterwards? Is there payment involved? These issues mean that some part of a model must take into account not only technical but also administrative aspects. Luckily most of the relations can be structured with basic components of agreements and trade. Someone gives permission, someone gets permission. Someone gives an order, someone fulfills that order. Someone sells a thing, someone buys a thing. Somewhere there is a made offer, somewhere an accepted offer (i.e. an agreement). Add a bit of logical sugar on top and those links can model most human relations, including the inner workings of an organization. When implemented, this paves the way for automatic (smart) contracts, micro payments, and granular, revocable permissions to information. Smart API therefore has a set of classes and modes to cope with these issues.

Finally, efficiency. Unfortunately, the more data is modeled, the slower it becomes to process, simply due to the overhead caused by the model data. That said, in the majority of implementation cases, the efficiency problems in data transfer boil down to the implementation of lists. Most of big data communicated in IoT is measurements collected into a list as a timeseries. Timeseries is a tricky thing when it comes to semantic technologies. Technically, it is an ordered list of values, and core semantic languages define ordered lists as recursive linked lists. This is a very poor design when it comes to handling megabytes of data. If parsing or serialization requires handling recursive datastructures, most languages fail in parsing due to call stack overflows. Python is particularly bad at this, a list with more than 5000 items is doomed to crash the software. And recursive parsing is painstakingly slow to start with.

Now, because the vast majority of data, as measured by number of bytes transferred is in the time-series format, designing and modeling an efficient semantics-compatible list serialization is absolutely necessary for the data transfer to happen. The container models included in the Smart API design offer efficient, semantic-language compatible implementations of lists and maps.

So, after this intro to the thinking behind the design, let's look at the actual implementation, starting with requirements for the technology and then the components of the model itself.

2. The core of Smart API technology

The Smart API is a technology for making better APIs for remote system and device management applications such as various IoT (Internet of Things) solutions. Technically, the Smart API is an **object centric, semantics enabled, transaction capable and secure method for transferring and storing linked data.** To understand a better what that is and why these features are important, let's open that up a bit:

- **Object centric**. Smart API has been designed for transferring remote objects between systems. "Objects" in this case means both the physical devices the objects represent and the programming abstractions software engineers actually use to model those devices. Objects are not only a natural way of representing physical "things" but also directly map into popular object oriented programming languages. Smart API transparently handles the details of transforming an object in memory into a datastructure transmitted over the network and back again, making the whole process easy and fast for engineers to use.

- **Semantics enabled**. Smart API builds on the principles of semantic web. The purpose of this technology is to make data exchange unambiguous and suitable for computers to deduct things from the data on behalf of users and programmers. The primary obstacle in most data exchange is that while it may work technically just fine, the data is interpreted incorrectly as different people and organizations use different terms and different measurement units for the same thing. So when for one person "length" is the dimension of a box measured in inches, for the other person the same thing is "depth" measured in centimeters. Common vocabularies of the semantics removes this ambiguity and makes it possible to do for instance unit conversion between inches and centimeters automatically as data is received or sent.

- **Transaction capable**. In most applications, there eventually needs to be some way to monetize the operations and data. For a long time it has been common to substitute revenue from the actual data with something else, such as advertising revenue, or apply some fixed pricing model such as a monthly fee on data services. But with the advent of Cloud Computing and Big Data, a more granular pricing model is often desired to pay just for the resources consumed. So you would pay for the amount of bytes transferred, the number of commands sent, or the number of measurements stored. Smart API transaction support is exactly for this purpose. It allows storing such details of data transfers in a way that is cryptographically protected for confidentiality and non-repudiation and serves as a ledger for invoicing.

- **Secure**. Smart API is end-to-end crypto enabled. While it is highly recommended and fully supported to use a secure links over technologies such as HTTPS, Smart API messages can further be encrypted and signed per each message. So no matter what data communication links are in between, how the connections terminate and how the data may be stored in intermediate locations, the data remains confidential all the way to the recipient. Smart API also has built in support for authentication using OAuth2 so you get the building blocks of security straight out of the box.

- **Linked data enabled**. Connected systems are all about - surprise - connections. Those connections link one thing to another, then to another and another and possibly back again. Such links create graphs. Understanding and being capable of processing graphs is the core of many IoT analysis applications. In Smart API, things can be linked between objects, within messages, across messages, and in any other configuration available and necessary.

So why and when would you use Smart API? The short answer is of course when you need features listed above. The slightly longer answer is that Smart API is a highly recommended technology when you build remote control and measurement applications. True, simple get / put REST APIs over some JSON structure are the norm and in many applications sufficient - at least in the beginning. But when the application grows and it actually needs to be integrated with some larger entities, when security becomes an issue, and when you need to make sure that data is actually correct before making automation decisions, Smart API becomes an essential tool. It saves software engineers from a ton of headaches caused by the tedious process of building data converters, it always stores data in an understandable format, it creates documentation for both the API and the data automatically, and it can handle data ambiguity in a professional manner. And as Smart API is free and pretty much as easy to use as alternative formats, there really is few reasons why not build a system properly from the beginning.

But there already are other API standards and tools such as Swagger/OpenAPI, why another? Well, the simple answer is: Smart API takes many of the best practices of OpenAPI such as declarative specifications and automatic generation of documents and code, but adds essential features needed by modern Big Data applications on top. Where the design of Open API originates from the API itself, Smart API's core is in the data. In a nutshell: OpenAPI is excellent is telling **how** data is transferred but lacks the functionality to tell **what** the data is. Smart API fills this gap. It supports vocabularies, links and graphs, something that OpenAPI does not. In SmartAPI the definition of data is made with proper ontologies which can be validated and tested. These are the building block for data accuracy needed in critical systems and scientific computing. And ontology definitions are the core of transforming data into knowledge, an essential process with artificial intelligence applications.

3. Basics of objects

In Smart API, an object is something that abstracts a thing or an item. What an object actually represents is pretty much up to the application. An object could be something physical like a vehicle, a person, a part of a machinery or something abstract like a service or a design. All in all an object (or Obj) is the mother class of all other classes in Smart API. Its implementation provides all common technical functionality needed to operate with objects with a given programming language. Such functionality includes, for instance, identification, property assignment, introspection, and security functions.

Because Smart API is designed for the Internet of Things applications, this objective plays a major role in how the inheritance structure below an Obj is organized. Classes that inherit Obj can be roughly divided in three branches:

- **Entities**. These are the recognizable or tangible things operated on with IoT, including devices, services, people, etc. Entities are the targets of processing. Properties of Entities contain the **data to be processed**.
- **Evaluations**. These are the intangible, technical things in the network that act as data carriers, including requests, inputs and outputs. Evaluations are the definers of processing. Their methods contain **instructions on how to process data**.
- **Property containers**. This category includes all data models that give a standardized format to the properties of Entities. These classes ensure that the data is understandable by the parties with the help of **structures and vocabularies**.

So each object is essentially modeled as an Entity, an Evaluation, or something that defines the value of a property of and entity or evaluation. Entities model what is passed, Evaluations how they are passed, and the rest of the classes glue these things together. As a mnemonic, you could say that Entities are the "Things" part of the design, Evaluations the "Internet" part, and the rest is the "of" in between.

Entities are then split into two subclasses:

- **AbstractEntities** (like a service).
- **PhysicalEntities** (like a vehicle).

PhysicalEntities have physical characteristics such as size and weight whereas AbstractEntities do not. This design above is pretty much the core of the object structure when it comes to describing something that is a tangible thing. A PhysicalObject can be further subclassed into say a vehicle, and a vehicle into a motorcycle, but such subclassing has only a minor impact on the actual processing. The important part is to define whether something is an Entity or not (if it is not, then this Obj is probably a characteristic of an Entity) and whether that Entity is physical or abstract.

Evaluations on the other hand are subclassed into:

- **Activities**, i.e. services that process data.
- **Abilities**, i.e. definitions of what a service can and cannot do (including Availabilities, which define when the service is available and Restrictions that say when it is not).
- **Inputs**, i.e. freeform input data to an Activity.
- **Outputs**, i.e. freeform output data of an Activity.
- **Messages**, i.e. structured data passed to an Activity (including Requests, Responses, and Notifications).
- **Provenances**, i.e. definitions of the source and reliability of data.

Most of the network processing takes place by creating an Evaluation for the target party to process and attaching data to it in the form of Entities. The most typical procedure involves creating a Request that identifies the target party, linking an Activity that identifies what the target should do, and packing the processing data into the Activity as a set of Entities.

3.1. Properties

Each object can hold some data and do something with the data. Held data is stored into Properties. A Property can have a value which is either represented by a literal or an object. This object can be a single object or a list of objects. An object can be represented by a freeform data structure (dictionary) of the Map type or it can be one of the property containers.

3.2. Identified versus generic properties

A property is attached to an object with a predicate. Smart API standardizes in its vocabulary a wide range of predicates to ensure all systems speak the same language. That said, the predicate can also be a custom one simply set by the code or an extension to an ontology.

A property can be attached with or without an identifier. This procedure sets the semantic meaning of the property beytween something that is generic and something that is specific. Without an identifier, the property is understood as a generic member of that property type, say "weight". A value of this property would represent some weight of the entity in question. Attaching an identifier to it, let's say "certified weight" would say that this is a particular weight of the entity, in this case the certified one. If we add two values to the property weight, without an identifier we would not know which one is the certified one.

4. Models for tangible things

4.1. ValueObjects; Quantities, Units and Datatypes

As Smart API is primarily designed for IoT, the properties of Entities usually represent some physical phenomenon that can be measured or changed, such as pressure or height.

The standard container of such measurement is a `ValueObject`. As the name implies, it is an object that carries a value. But in addition to the literal value of the property, the `ValueObject` describes a value in detail, giving it the quantity, the unit, the datatype, and the time (instant) it was recorded as well as limiting factors such as minimum and maximum.

4.2. PhysicalEntities

Physical entities are things of physical nature; cars, buildings, boats, computers, and so forth. It is common to extend the Smart API model by subclassing this class into specific classes representing those objects. However, from the standpoint of the functionality of the API, this is usually unnecessary. The type of physical entity is defined by the RDF Type property and setting it does not require a subclass as such. So if you set the type of an Entity to be "a car", the other parties should understand that that Entity is a car. The features of the car are stored in the properties with the vocabulary of a car. Setting the class and the properties does not require an explicit subclass.

There are subclasses of PhysicalEntities defined in the Smart API such as Devices and Persons but those classes in actuality work as programming shortcuts to the properties, not a further structural models of data. They don't add a deeper structure into the model, only offer getters and setters that use the standardized vocabulary to set for instance the first name or surname properties to a Person. The resulting RDF can be parsed and understood even without the actual subclass. So in this respect the Smart API object model is "flat".

This is an important point of the data model to understand: the classification of an object is orthogonal to the property model of the object. Even though an electric car can be classified along to a hierarchy (say, physical object -> vehicle -> car -> electric car), its speed property will still be just a simple property attached to the entity.

4.2.1. Items that describe physical entities

There are certain property types that do require a further structure to be implemented. Usually these are properties where the value is not a single literal but a collection of multiple literals. For example, location is defined by a latitude, longitude, altitude, and instant (3D coordinates in space-time). Orientation is a similar multi-literal property, comprising yaw, pitch and roll. Additionally, there are structures that are lists by nature (such as a route that is a list of waypoints).

To make life with such multi-literal definitions easier and to ensure everyone does it in a similar manner, Smart API defines a set of such properties in its model. You can find such standard physical item containers in Smart API in the form of Velocity, Size, Orientation, Direction, Route, Coordinates, and Waypoints.

4.3. AbstractEntities

AbstractEntities are Entities without direct physical nature, such as a network service. As with PhysicalEntities, the subclasses of AbstractEntities don't actually add any further structure to the model. They simply are programming classes that ensure that the class gets the correct RDF Type without the programmer having to remember the vocabulary.

4.4. Items that describe what Entities can do

Smart API defines a set of Evaluations that describe the abilities of objects. These Evaluations are used in publishing the object to other parties. They help in finding objects and preparing the network connections with them. So for instance, if you would like to find a rental car in Tokyo for a weekend, the location property of a car object would say whether it is in Tokyo while an Availability attached to the car would say whether it is available on that weekend.

4.4.1. Abilities

An Ability defines what types of actions some Entity is able to perform. Because Smart API works on objects and their properties, an Ability effectively lists which properties an Entity can process and how (read, write, etc).

4.4.2. Controllabilities

Controllabilities define the remote control properties of an Entity. If an Ability says what an Entity can do **by itself**, then a Controllability says what others can **do to it**.

4.4.3. Availabilities

An Availability says when an Entity is available i.e. when it can be accessed in some manner. This can be defined for instance with a time, distance, or access rights.

4.4.4. Restrictions

A Restriction is essentially a negation of an Availability. It isa formalized way of saying "I'd be happy to do this, but...". So the Restriction describes to the receiver how access to an action or data is limited. It could be limited for instance to only certain users or accessible only at a certain time period. The client can then decide to request the data in some other way (or at some other time) when the Restriction is not in force.

4.4.5. Provenances

A Provenance tells what the origin of the data is. They can be used to convey information about how reliable the data in question is, how it has been generated, or what is its source. A Provenance is therefore an indication on how well an Entity performs some task.

5. Models for network communication and processing

Network communication is of particular interest in Smart API as it is designed for devices communicating over a network, namely the Internet (not forgetting the lower level protocols employed before the actual Internet connection). The primary purpose of this modeling is to make it possible to autoconfigure devices and streamline the connection processes. Further, the data transfer should support passing messages over several protocols and connection points, effectively providing an end-to-end messaging protocol that is resilient to delays and connection timeouts.

5.1. Interfaces and Heartbeats

An InterfaceAddress is the basic model for defining a network connection interface of a device. The InterfaceAddress defines things such as port numbers, IP addresses and similar. Announcing or exposing the InterfaceAddress data allows other services to take the connection to a particular device.

In addition to interfaces, Smart API models connections between Entities. This is done with the managedBy relations. These can be set in both ways, i.e. Entity 1 **manages** Entity 2 or Entity 2 is **managedBy** Entity 1. The management links make it possible to model things such as gateways. From the linking definitions, a control system can deduct that if say device A does not have a public interface but it is managed by device B, then controlling device A may be possible by sending a command to device B (the gateway).

A Heartbeat is the basic form of debugging messaging to make sure a connection works. The Heartbeat is similar to a network ping. It is sent and received by request or at regular intervals to ensure that a connection is still alive.

5.2. Requests, Responses and Notifications

As Smart API is designed as a data format for APIs, the top level format of Smart API messages follows the API messaging convention: sending and receiving data. There are three basic means for communication between systems defined in Smart API:

- **Requests**. This is something a "client" sends to a service.
- **Responses**. This is something a "server" uses to respond to requests.
- **Notifications**. This is a "fire and forget" message send to someone without expecting a reply.

Requests, Responses and Notifications are the wrapper for all other data. They contain the basic information of the data exchange, including the sender identity and the message creation time.

The data recipient is always expected to try to parse the data into one of these three objects. Class definitions in the data will tell the receiver which one of the three it actually is. Once this parsing and type recognition succeeds at the receiver end, the receiver is guaranteed to have something to act upon.

5.3. Activities

When a request is sent, some action is expected to happen. This "action" is represented by Activity. For instance, if a service that receives data can process it in, say, two ways, for instance `add` and `subtract`, then it should have two Activities; one for add and one for subtract. When the service is called, the caller will address the type of action it wants to be performed with the ID of the Activity.

Activities can work with five basic methods that confirm to the classic CRUD-N model:

- **Read**. Return some data.
- **Write** (or update). Modify some data.

- **Create**. Create some new data or objects.
- **Delete**. Delete data.
- **Notify**. Tell those interested that something has happened.

When one of the five method types is invoked, data forwarded to the Activity can be given with two ways

- In **object oriented** mode by passing it Entities.
- In **functional** mode by passing it Inputs.

5.3.1. Object-oriented calls

When an Activity receives Entities, it is assumed to look for objects that match the descriptive data of the Entities in case and perform the action on those. The response value of an object-oriented call is always either an Error, a Status or an Entity (or the same in plural).

- An **Error** is sent in the response if the requested Activity cannot be performed (not found, server error, incorrect data, etc).
- A **Status** is sent in the response if the requested Activity will be performed but not now (delayed, asynchronous execution). A Notification may later be sent to inform that the process is actually completed.
- An **Entity** is sent in the response if the requested Activity has been performed. Data in the Entity reflects its state after the operation.

Note that an Activity can process an Entity in stages and inform the requester of progress as it happens. In such case the initial response to the request is the current state of the Entity and the later Notifications are updates to that state. For example, let's imagine we are spinning up an engine and send a command that the targeted speed is 4000 RPM. If the engine is not running, the system that controls it would respond to the request with an Entity that has its RPM property set to zero. This is a confirmation to the request stating that the command has been received, now starting from value zero. As the engine spins up, regular Notifications can be sent with the RPM value set to the speed at that point in time; 500 RPM, 1000 RPM, 2200 RPM, etc.

Sent Entities should be understood as templates i.e. "this is the way I'd like the Entity to be". In a write operation, the recipient takes the template and tries to change the corresponding local object to match the template. In read operations, those properties that have an assigned value act as search filters. So, for instance, if a service receives a read call with an Entity with property "color" set to "red", it should try to find all Entities that are marked as red. If a property is completely omitted from the Entity (not set at all, as opposed to set without a value), it acts as a wildcard as is deemed appropriate by the receiver.

Once the receiver finds the Entities as filtered according to the template, it should perform specified operation on them. If the method is read, the service should return all those Entities that match the search. If it is delete, they should be deleted. In create however, only one Entity is created no matter what types of wildcards are present.

In a read operation, those properties that are set but do not have a value, mark the properties that should be returned with values. So, for instance, an Entity with type property set to "Car", color property set to "red", and speed property set to nothing (i.e. a ValueObject with no value property), should be interpreted as a request to return the speed of all red cars. The ValueObject can still contain Quantities and Units. So in this case if the ValueObject of the speed property has quantity set as miles per hour, it means that the requester wants to receive the speed values of all red cars expressed in miles per hour.

In a write operation, the properties that mark a value (are of type ValueObject) are not used in the wildcard but are used as values to write, whereas the wildcard properties are never written. So for instance, if a write is invoked with an object that has some property expressed with a ValueObject (which in turn has quantity and value set), that ValueObject is not used as a wildcard. Instead the service could find all objects that match some other property, say type. Then the ValueObject is applied to the corresponding property of all the found objects.

> **Note**
> an Entity that has an identifier set is never used as a wildcard, no matter what information is contained in it or is missing from it.

5.3.2. Functional calls

When an Activity receives Inputs, it takes these as similar inputs as any function call would. The service should find some function in its logic that matches the identifier of the Activity and feed the input to it. Once the function is invoked, any function output is put into Output objects and returned. Thus the response value of a functional call is always either an Error, a Status or an Output (or the same in plural).

- An **Error** is sent in the reponse if the requested Activity cannot be performed (not found, server error, incorrect data, etc).
- A **Status** is sent in the reponse if the requested Activity will be performed but just not now (delayed or asynchronous execution). A Notification may later be sent to inform that the process actually completed.
- An **Output** is sent in the reponse if the requested Activity has been performed. The Output(s) carry the return values of the function.

Inputs and Outputs are simply meant as semantic carriers of any freeform inputs and outputs of methods. Smart API does not standardize their content in any way to ensure that effectively any function call can be modeled semantically. This means that there is no need to have some separate definition of non-semantic or legacy calls, anything can be expressed with Smart API if needed.

The carriers of Input and Output data are typically Variants. Please see later in this document further explanation on Variants.

5.4. Items limiting Activities

When Activities operate, they can have some type of span. The Activity can span over time or over a geographical area.

5.4.1. TemporalContexts

A TemporalContext defines a time or timespan (depending on the content of the context). When applied to an Activity, it limits the Activity to data that matches that time period. For example, if a property of an Entity is requested, the TemporalContext will instruct the Activity to find all values of that property within the timespan specified in the context.

> **Note**
> If an Activity in a request has TemporalContext defined but a data item does not have any timestamp (or the timestamp value is null), then this data item should not be included.

5.4.2. GeospatialContexts

If a TemporalContext says **when** something should take place, a GeospatialContext says **where** it should take place. For instance, if there is a GeospatialContext attached to an Activity and that context has a center point and a radius, then the Activity will be applied to all objects within the radius from the center point.

> **Note**
> If an Activity in a request has GeospatialContext defined but an object does not have any position information (its coordinate is null), then this object will be excluded.

6. Models for contracting and trading

When data is exchanged in a commercial environment, it usually involves some form of compensation or at least an agreement on the exchange terms, including the rights to data. Such agreements are usually made "offline", i.e. with a traditional legal and contractual process. But in IoT, automated contracting and micropayments are often a tool that helps in high volume applications especially if the number of contractual parties is large (say a service that enables sharing personal property like cars or bicycles).

Smart API includes models to add such contractual terms into the data and the technical solutions for electronically signing and confirming the contracts. Parties in an exchange have certificates to prove their identities and public/private key pairs to perform signing operations. The Smart API library contains a crypto module to operate with them and automatic recognition and processing of security data. Smart API also defines a message format that enables hashing and signing of semantic data that otherwise would change its representative form in data transmission, making it impossible to verify using a byte-by-byte comparison, as demanded by signature algorithms.

6.1. Offerings

An offering is as the name implies something that is offered to another party in return of payment. Any Entity can be attached with an Offering, implying that a party wants to form a contract before the transaction concerning that Entity can continue. A contract is formed by signing the Entity that contains the Offering.

In a typical exchange, the requester asks for an Entity for which the responder wants a price. So instead of returning the Entity with its data, the responder returns the Entity with properties having no values and with the Offering attached. This means that the responder wants to have a price specified in the Offering for the data in the properties. The requester then either discards the Offering, implying that they do not want to pay, or signs the Entity, together with the Offering, confirming that they are willing to pay for the set of properties as presented. The signed Entity is then used for a second round of requests. If the responder now accepts the signature, the Entity with its full data is returned.

The offering should always be attached to the thing that needs a price attached to it. For instance, if a client requests two Entities and one of them is free, the other needs payment, the server should return the first Entity with its full data and the second Entity with empty properties in place of those requested and an Offering attached. This means that the properties presented (and empty) will cost whatever is stated in the Offering.

6.2. Prices

A PriceSpecification is a structure that describes a price of something. The price can be a single value or a set of prices that together form a price list. A price can further be a simple value or a dependent value. It is therefore possible to model prices for example in the form of conditions e.g. the price of A is P1 at distance D1 but P2 at distance D2. And so forth.

6.3. Contracts and Licenses

A Contract is something that represents a previously made agreement on some transaction. A Contract typically has a validity (valid from time A until time B), a price, and a set of items that govern the contact.

A License is a Contract that governs something that can be licensed, such as information or software. Because Smart API is made for transmitting data, Licenses have a special role in Smart API. Technically they contain a license key (API key), which is a typical access model applied by a majority of data services on the Internet.

6.4. Accounts and Transactions

When payments are made, their value needs to be stored somewhere for bookkeeping. This is what Accounts are for. Accounts manage how much money can be spent, how much must be deposited, whether credit can be granted, etc. The balance of the Account itself is a collection of AccountTransactions. Whenever something that affects the balance of the Account occurs, a transaction is created. The balance of the Account is then the sum off all its AccountTransactions.

7. Object grouping and access control

7.1. Zones

The purpose of zones is to group entities together. A Zone is an abstract concept that can describe a map area (geographical zone), a grouping of objects for access (administrative zone), or some other confined area or connection that binds the objects together (physical zone).

In access control, Zones define which objects belong together in terms of having similar access rights. For example, all sensors in "Building A" could belong to a zone names "Building A". Granting access to this zone would open access to all the sensors in Building A.

7.2. Authorizations

An Authorization is something granted to a person to access a resource. An empty Authorization means that an Authorization is needed for the resource, a filled in Authorization that the access has been granted to the Person attached to it.

8. Data modeling classes

In addition to the structured modeling for data semantics, there are a couple of helper classes that are necessary for fluent programming with the Smart API library in various languages. Their purpose is to give each programming language a similar programming API that can be mapped into RDF so that implementations across languages look as similar as possible given the syntax constraints of each programming language. These include Variants, Parametes, Lists and Maps.

8.1. Variants

A Variant is simply a value that can hold any type of value. It gives flexibility of programming to strongly typed languages and can return values in the original type when requested. In a loosely typed language such as Python it is not really needed as anything can be assigned to anything on language level. But in strongly typed languages (Java, C++, C#) the Variant is necessary to be able to assign values to properties flexibly.

8.2. Parameters

A Parameter is a semantic wrapper for a non-semantic property name. The syntax of RDF requires that each predicate has a proper URI format, starting with http://... In some custom programming applications this may be an overkill if, for instance, an input parameter is fully custom and never used for public API processing, thus not really requiring a vocabulary.

A Parameter solves this problem by wrapping the non-conforming parameter format inside valid RDF and parses/serializes it back and forth so that programming code never has to encounter this problem. In the Smart API library the Parameters are set transparently to the coder and programmers therefore they seldom have to explicitly worry about them.

8.3. Lists and dictionaries

In most applications with data, lists is something you'll be handling a lot. In essence all large datasets are processed as lists at some point in time. Not surprisingly, Smart API also has comprehensive list support.

The challenge with lists is that there are many possible ways to represent them. They can be linked, indexed, ordered, etc. Each method of representing a list has its pros and cons. One may be fast but non-ordered, the other ordered but large to transfer. What is the most suitable format is largely dependent on application and developer choice.

Smart API offers the following list types:

- `LinkedList`
- `OrderedList`
- `ItemizedList`
- `NudeList`

`LinkedList` stores serialized data items by linking the previous item directly to the next. This creates a hierarchical data structure where the depth of recursion needed to reach the final item is the same as the number of items in the list. `LinkedList` has the benefit of having a serialization that is 100% compliant with the RDF standard. The downside is poor performance in most platforms, both in terms of processing time and speed, and the risk of overflowing the call stack due to deep recursion. You can use this type of list for small lists containing less than 1000 items. Using a LinkedList for bigger lists results in delays due to slow serialization and parsing, and may fail due overflow caused by the depth of the data hierarchy.

`OrderedList` stores serialized data items in an indexed array. Because it does not create a deep hierarchical structure, it performs much better with large datasets than `LinkedList`. `OrderedList` is recommended to be used with lists that need to be ordered and contain more than 1000 but less than 100000 items.

`ItemizedList` stores serialized data items in an array. The structure is similar with the `OrderedList` but the items in the `ItemizedList` do not have an index number. `ItemizedList` is recommended to be used with datasets where the order of the items does not matter, and they should work fine even when the dataset is large.

`NudeList` stores serialized data items in a custom performance-optimized non-semantic format (based on JSON). Because most programming languages and libraries have binary optimized code for processing JSON a `NudeList` is by far the fastest of the implementations and consumes the least amount of memory. It is optimal for large amounts of data where the data is just literal values (i.e. ints, doubles, dates, strings). The downside of this list is that is performs badly when transferring non-literal values i.e. full objects. `NudeList` is the recommended format when the data is not objects (just literals), and you need to process efficiently a large amount of data (up to several million entries).

To simplify development, all lists provide an identical API. This essentially means that you can change list type "on the fly" without modifications to the rest of the code. The recipient of data automatically detects the incoming list type and parses it accordingly. When you do change the list type, what you'll see is differences in performance (speed and memory use) and in the size and format of the data that is transferred over network.

8.4. Conditions

Conditions emulate Boolean algebra operations and can be used to input multiple property values in a form of a conditional formula instead of a fixed value.

Asema Electronics Ltd
Copyright © 2011-2019