English

Smart API

A smarter way to exchange data

ASEMA

Table of Contents

1. Why is there a need for smarter data?

The purpose of the Smart API specification is to promote understandable data and APIs. What this means is that the design is such that their use is as unambiguous as possible and it takes little if any previous introduction what the data is and how API's function.

The problem is becoming more and more pronounced now with IoT applications and Big Data. Technologies create vast possibilities to collect data from anywhere and everywhere and it is simply hard to justify not collecting data just for the sake of collecting it. More often than not IoT applications collect raw data just in case it may become useful in the future. If the cost of collection and storage is marginal, it is less risky to just collect everything than try to second guess whether something might be useful in some future scenario.

The assumption many times is that it is perfectly fine and risk free to do so. After all it is processed by precise computer software that has been carefully crafted by skilled engineers who know what they are doing.

Oh, really?

Now, while software engineering is a precise science and computers force engineers to be diligent, the ugly truth still is that most software engineers treat data like dirt. The profession teaches and mandates automated tests on software and its functions but the data that is processed is usually left without attention. Which is in a way natural. Software engineers are interested in the functionality, it is their job to implement it. Once done, they move on to the next one. The collection and manipulation of data is left to the users who on the other hand usually neither have the skill nor the visibility to the actual format of data. They really don't know how and even where and when something is stored.

So looking at an old database of data is like finding a dusty yellow sticker note of an employee who used to vacate your office years before. "Mike at 5:25 pm, orion project, app code 67Uw". What is the orion project? To what app is that code for? And, actually, who the hell is Mike???

An old database was probably designed and implemented by an engineer who left the company or was transferred to another business unit years ago. Or worse, it was the summer project of an intern. Data collection is usually the dull part of software development process so engineering teams are notoriously good at allocating that work to temps. Who in turn seldom leave behind proper documentation of their work. The result: quite a pile of data no-one really understands later on.

The solution? A better approach at the data itself. Let's next take a look at the data modeling Smart API offers.

2. Expressing data with Smart API

2.1. Smart API objectives

Smart API has been primarily designed for interaction between systems. For APIs this means that the engineers of an organization that uses the API require minimal support from the engineers of some other organization that supplies the API when systems are interconnected. For that to happen, both the call to the API and the data transferred in the call and a response to it must be understandable.

Smart API makes it possible to define data to such detail that actually no previous interaction is needed to connect systems. No API manuals, no long debugging sessions over Skype. In the optimal case the API can configure itself fully automatically and the data can be interpreted with automatically molding parsers.

As a sidenote: to avoid becoming a hindrance to work and a spec that engineers actually start to hate - let's face it, most still hate the dull design work no matter what - Smart API specification allows the creation of designs with different levels of understandability. It is therefore allowed, although not encouraged, to create "bad" designs i.e. designs that are hard to understand. Smart API development tools include a tester that tests the general understandability of the API and the data and gives it a grade. The better the grade, the easier it is to use the design for system integration. Consequently, poorly defined designs do work technically, but get a bad grade.

In Smart API, good designs are incentivized instead of forced. One of the guiding principles of Smart API has been that Smart API makes it possible for previously unknown parties to find service APIs and connect to them. The better the API is defined, the easier it is to find. For a service provider that makes money out of API use, it is naturally important to design the API in a way that the API is easy to find and use.

2.2. An example of bad data

To actually show how Smart API works, let's have an example. Below is some data you could find in one of those infamous databases someone did a couple of years ago.

```
col1    col2    col3    col4    col5        col6
Rosie   19      76      Bon     42-39-56    eMPjs55hNzQ
```

What on earth could this data be about? Without further info all we can do is guess. If you happen to be French, an interpretation that might pop into mind is that this is an exam score of a 19 year old high school student called Rosie. Her overall score was 76, which was rated as good ("bon") and the individual scores of test exercises were 42, 39 and 56. Her student ID in the school database is `eMPjs55hNzQ`. So, well done Rosie. You're a good girl!

Correct? Not even close. In reality, those numbers are from a song called "Whole Lotta Rosie" by the Australian rock band AC/DC. It tells the tale of an actual event in the band's past. The story goes that in 1976 (or possibly earlier, the story was first told in an interview in 1976), the band's singer Bon Scott met a Tasmanian woman at Freeway Gardens Motel in Melbourne and had a night of wild sex with her. Now, as the song's lyrics say "she ain't exactly pretty, she ain't exactly small", the woman, called "Rosie" in the song, was not one of the smallest of the fans the band members encountered. With a bust of 42", waist at 39", hip at 56", and weighing 19 stones (about 121 kg) you could say that Rosie was a "big mama". And according to Bon Scott, one of the best lovers he had ever had.

The story further goes that after the little session with Bon Scott, Rosie took out her black notebook and put Bon's name there. Then she counted the names and said Bon was her 27th sex partner. In that month. Yes, Rosie really was not one of those good girls.

That cryptic set of letters and numbers in the final column expands into a YouTube URL https://www.youtube.com/watch?v=eMPjs55hNzQ which links to one of the live performances of the song. As a nod

to the French, this particular recording is from a gig in Paris in 1979 where Angus Young broke his guitar in the middle of the solo. The band just kept on rocking while Angus obtained and adjusted a replacement on the fly to finish song with technicians adjusting his strap while he was already on stage ripping the final chords.

Go on, open it and listen to it. The next section of this text is the technical description of how that database could be improved with Smart API and nothing beats a bit of hard rock while reading a dull (or as Australians would say "piss boring") technical spec.

2.3. Improving data design

What makes the design in the previous section bad is that without some supplementary documentation, it is hard to tell what this data is all about. Just proper column names would do a lot, or a text format such as JSON. Let's try that

```
{
 "name": "Rosie",
 "weight": 19,
 "partner": "Bon",
 "year": 76,
 "measurements": "42-39-56",
 "suffix": "eMPjs55hNzQ"
}
```

Much better, but still very ambiguous. At least we know what those columns are about but it is hard to tell how to process this data. 19 what? Grams? Kilograms? Ounches? Is she a "good partner" or what does that "Bon" mean?

Usually in this kind of case, the author of the API releases some API documentation which explains what the content actually means. Here's an improvement, a short "API Doc".

```
Name. Name of the person in case.
Weight. Weight of the person, in stones.
Partner. Another person this person is associated with.
Year. Year of contact with the other person.
Measurements. Hip, waist and bust of that person, in inches, in that order.
Suffix. YouTube video identifier.
```

With the documentation, the situation is naturally is much better, now we understand what those numbers and texts are about. But there still are several problems in it.

- First, someone needs to find the API document, read it and understand it. This requires time and manual work.
- Second, because the definitions are outside of the payload in an external documentation, the spec is rigid and changing the API is difficult. What if in some cases we would actually like to express a measurement in millimeters. This is not possible without releasing a new version of the API. Even then mixing different quantities, i.e. sometimes give the values in centimeters sometimes in inches, would not be possible.
- Third, the spec is separate from data. While admittedly this makes the payload small in size, the importance of such considerations is diminishing due to high speed connections. Of increasing importance starts to be the storage and later use of that data. The assumption behind payload formats is that the recipient will parse the data and further process it immediately. But in many data mining applications this may not be the case. In them the data should be stored in document databases such as MongoDb in their original form to avoid losing any information that might become valuable in the future. But in this case also the documentation would need to be stored, otherwise no-one will understand the data ten years from now.
- And finally, there are ambiguities in the interpretation of the data, especially the part that describes the association of persons. What is the actual association, how does that link to the other person, and how do we find that other person.

So let's take a slightly improved design as an example.

```
{
 "name": {
  "firstName": "Rosie"
 },
 "userAssociation": {
  "userId": "Bon"
  "associationType": "someWildBadaBadaBing",
  "year": 1976
 },
 "measurements": {
  "weight": {
   "unit": "stones",
   "value": 19
  },
  "bust" {
   "unit": "inches",
   "value": 42
  },
  "waist" {
   "unit": "inches",
   "value": 39
  },
  "hip" {
   "unit": "inches",
   "value": 56
  }
 },
 "videoUri": {
  "prefix": "https://www.youtube.com/watch?v=",
  "suffix": "eMPjs55hNzQ"
 }
}
```

This one already embeds a lot more understandable data in it. Actually, the need for API documentation was just reduced radically. For instance, just by looking at the data we know that there measurements of the person, and the units they are expressed in. But while improved, this version is not yet good enough to be automatically processed. A computer would still not understand it. For instance, how would the computer understand the word "unit"? Is that somehow preprogrammed and standardized? If so, how exactly is it actually measured? And, as you've probably spotted, it uses terminology that without a context is rather hard to interpret. So let's further refine the design.

The following example adds linked identifiers into the data. This is now one representation of data in RDF, a data specification Smart API uses. There are many formats of RDF, all supported by Smart API. The following format is JSON-LD.

```
{
 "@context": {
  "person": "http://smart-api.org/ontology/custom/person#",
  "rockfan": "http://smart-api.org/ontology/custom/rockfan#",
  "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
  "nasa": "http://data.nasa.gov/qudt/owl/qudt#",
  "unit": "http://data.nasa.gov/qudt/owl/unit#",
  "xsd": "http://www.w3.org/2001/XMLSchema#",
  "vcard": "http://www.w3.org/2006/vcard/ns#",
  "foaf": "http://xmlns.com/foaf/0.1/"
 },

 "@id": "http://acdcfans.example/people/Rosie",
 "@type": "foaf:Person",

 "vcard:given-name": "Rosie",

 "rockfan:connection": {
  "rockfan:connectedPerson" {
   "@type": "foaf:Person",
   "@id": "http://acdcfans.example/people/Bon"
  }
  "rockfan:connectionType": "rockfan:SomeWildBadaBadaBing",
  "unit:Year365Day": { "@value": 1976, "@type": "xsd:int" }
 },

 "person:measurements": {
  "person:weight": {
   "unit:unit": "nasa:Kilogram",
   "rdf:value": { "@value": 121.2, "@type": "xsd:float" }
  },
  "person:bust" {
   "unit:unit": "nasa:Inch",
   "rdf:value": { "@value": 42, "@type": "xsd:int" }
  },
  "person:waist" {
```

```
    "unit:unit": "nasa:Inch",
    "rdf:value": { "@value": 39, "@type": "xsd:int" }
   },
   "person:hip" {
    "unit:unit": "nasa:Inch",
    "rdf:value": { "@value": 56, "@type": "xsd:int" }
   }
  },

  "vcard:hasUrl": "https://www.youtube.com/watch?v=eMPjs55hNzQ"
}
```

Now this one is starting to be much more accurate. First, we've added in the beginning a context. It tells within which framework we are talking about things. For instance weight is expressed not just in kilograms. It is measured in kilograms in the way that NASA scientists have officially defined a kilogram to be. The context says that such a definition can be found from the link that is marked with the prefix `nasa`. Computers can now interpret such things automatically because the expanded URI of the weight is `http://data.nasa.gov/qudt/owl/unit#Kilogram` which is the universal definition of weight expressed in kilograms. So if we would now receive data of the people and both have the weight in `nasa:Kilogram`, we can automatically deduct that having those two people in an elevator would result in some summed load that can be obtained by adding together the fields that have weight expressed in the same `nasa:Kilogram` unit.

Second, items have a unique ID. Rosie's ID is http://acdcfans.example/people/Rosie. With that ID we can point to her and look for data about her. Or create a link to another person, in this case http://acdcfans.example/people/Bon. Further, as you notice, the numbers have a type. This way the data can be accurately converted into programming languages that are strongly typed, like Java. Weight is given as a floating point number while bust size is an integer.

Finally, there are two custom contexts for this data, `person` and `rockfan`. These are so called ontologies of data. In all simplicity, these are the places where we have documented what we mean by some term. In a way they are the "online API doc" of the data. But instead of being just some random document, they actually have a specific format that is computer interpretable. For instance, at the URL http://smart-api.org/ontology/custom/rockfan we can explain what we mean by a "connection". And naturally the ontology would also say that a "SomeWildBadaBadaBing" is "a one-night stand by a skinny drunken rockstar who has a tendency to like large dominant women" (wasn't that obvious?).

2.4. Data compatibility and customization

To make data compatible between systems, users of Smart API are encouraged to use the same terms as everyone else does. So if someone adds a uri for additional info, it would be recommeded to use the `hasUrl` property of the vCard ontology consistently across systems as it is predefined and already widely used.

But for sure in most applications engineers bump into data and definitions that no-one has previously thought of, let alone defined. So while the wise engineers at NASA have carefully thought through every detail about kilograms and meters, it is doubtful they have considered of the need for some good old badabadabing. Or maybe they have, it is said engineers think about it on average once every six minutes. But maybe they did not write it down. Who knows. At least we couldn't find it. Which we found rather disappointing.

Anyways, for the purpose of adding missing bits and pieces, the Smart API design includes the possibility to include those ad hoc, non-defined values into the data or to extend the pre-designed definitions with custom definitions. They can be added on the go while programming or with online tools. These specs are then shared with everyone.

If you would receive this data with Smart API and issue the command `explain` with the Smart API SDK, Smart API would actually fetch all those ontologies, combine them with the data and explain to you exactly what each of the fields means in detail. Neat. Or you can feed the data to a code generator which automatically generates software code for interpreting the data as objects in a programming language. Damn neat. So neat that if it was any neater, the engineers would find their little badabada go "bing!" pretty soon.

3. Working with Smart API

3.1. Combining data accuracy with development speed

Now, defining things in the accuracy that was in the final example of the previous chapter of course takes some extra work in the beginning. There simply is a lot more to think about than just typing in some random identifiers. But the effort quickly pays off. You can either spend one hour making the API and two hours documenting and explaining it, or you could spend two hours creating something that requires no further explanation.

Smart API comes with an SDK and online tools that help in creating the data. While the format of the data of Smart API is human-readable, in the vast majority of cases it is still computer generated and interpreted. Programmers don't actually write the data that was shown, it is created by the SDK. When using the SDK, tools that integrate straight into the favorite programming tools used in software engineering guide the process. So if someone needs to type in a property that uses "kilogram", the tool automatically suggests the appropriate terms and autofills the code.

That said, there are still cases where going through all the hassle of defining everything with scientific accuracy simply does not pay off. For instance there could be a helper API that takes two lists A and B and adds the numbers of them together. Documenting this in the form "A is the first list", "B is the second list" is not only tedious but for the most part useless as it really does not add any further info. Because of this, Smart API supports very standard data structures, lists, dictionaries (maps) and just literal parameter values that require no further explanation. Programmers need to do nothing extra to use them. The native format in the programming language they have, say Python or C#, can be used to enter values into methods. The Smart API SDK does conversion automatically back and forth. So the extra effort for using a "smart" way of data to a traditional one is literally zero.

So let's take a couple of further examples in the next section. All of them are legal according to the Smart API spec, but use different means for expressing data. For clarity, the payloads are written in Turtle, another format for RDF. You can automatically switch between Turtle, JSON-LD and other serializations of RDF. They are equivalent in expression and the Smart API SDK can automatically convert data between them.

3.2. Expressing properties as parameters

Parameters are a flexible way to define custom, application specific inputs and outputs in valid RDF format. They emulate function variables so it is easy to map an existing function call to the payload. Parameters can be strongly typed i.e. strictly define the type of input/output and can also emulate data structures, including dictionaries (hash maps) and lists.

Let's write some data in Turtle, here the property `roomTemperature` is modeled as a parameter.

```
@prefix smartapi: <http://smart-api.org/ontology/1.0#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix nasa: <http://data.nasa.gov/qudt/owl/qudt> .
@prefix unit: <http://data.nasa.gov/qudt/owl/unit#> .
@prefix quantity: <http://data.nasa.gov/qudt/owl/quantity#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

[] a smartapi:Input ;
    smartapi:weight [
        nasa:unit  unit:Kilogram;
        rdf:value  3^^xsd:int ],
    smartapi:height [
        nasa:unit  unit:Centimeter;
        rdf:value  4^^xsd:float;
    smartapi:parameter [
        a smartapi:Parameter ;
        smartapi:key "roomTemperature" ;
        rdf:value    34 ].
```

That one is valid RDF. The benefit of this design is that we've been able to define the two standard inputs, weight and height, in a very strict and understandable format but have had full freedom in defining some additional proprietary input also. The downside is that if we ask the API to describe the data, it will only say it is "a parameter" but cannot tell any further info about it. What would be important is to know at least the unit of number 34 so we'd know whether it is expressed in celsius and rather hot or fahrenheit and quite cold.

This data can however be improved easily. Let's define `roomTemperature` with a concept called a ValueObject which lets us attach specific units and quantities to it:

```
@prefix smartapi: <http://smart-api.org/ontology/1.0#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix nasa: <http://data.nasa.gov/qudt/owl/qudt> .
@prefix unit: <http://data.nasa.gov/qudt/owl/unit#> .
@prefix quantity: <http://data.nasa.gov/qudt/owl/quantity#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

[] a smartapi:Input ;
    smartapi:weight [
        nasa:unit  unit:Kilogram;
        rdf:value  3^^xsd:int ],
    smartapi:height [
        nasa:unit  unit:Centimeter;
        rdf:value  4^^xsd:float;
    smartapi:parameter [
        a smartapi:Parameter ;
        smartapi:key "roomTemperature" ;
        rdf:value [
            a smartapi:ValueObject;
            nasa:unit  unit:Celsius;
            rdf:value  34.02^^xsd:float; ] ].
```

Good. Now we know, without further documentation, that the room temperature is 34.02 degrees centigrade.

3.3. Data structures and nested models

Smart API ontology supports objects within datastructures such as lists and maps (dictionaries). With them it is possible to create combinations that fulfill the requirements of nearly any data format. Let's take a look at an example (in Turtle again)

```
@prefix smartapi: <http://smart-api.org/ontology/1.0#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix nasa: <http://data.nasa.gov/qudt/owl/qudt> .
@prefix unit: <http://data.nasa.gov/qudt/owl/unit#> .
@prefix quantity: <http://data.nasa.gov/qudt/owl/quantity#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

[] a smartapi:Input ;
    smartapi:parameter [ a smartapi:Parameter ;
        smartapi:key "myBoxes" ;
        rdf:value ( [ a smartapi:Map ;
            smartapi:entry
              [ smartapi:key "NameAtTheBottom" ;
                rdf:value "Box 1" ],
              [ smartapi:key smartapi:weight ;
                rdf:value [
                    a smartapi:ValueObject;
                    nasa:unit  unit:Kilogram;
                    rdf:value  3^^xsd:int
                ]
              ],
              [ smartapi:key smartapi:height ;
                rdf:value [
                    a smartapi:ValueObject;
                    nasa:unit  unit:Centimeter;
                    rdf:value  10^^xsd:int
                ]
              ]
            ] ]
            [ a smartapi:Map ;
            smartapi:entry
              [ smartapi:key "NameAtTheBottom" ;
                rdf:value "Box 2" ],
              [ smartapi:key smartapi:weight ;
                rdf:value [
                    a smartapi:ValueObject;
```

```
                nasa:unit  unit:Kilogram;
                rdf:value  8^^xsd:int
              ]
            ],
            [ smartapi:key smartapi:height ;
              rdf:value [
                a smartapi:ValueObject;
                nasa:unit  unit:Centimeter;
                rdf:value  13^^xsd:int
              ]
            ]
          ] ]
      ) ] .
```

Here we have an input to an API that takes as its input a list of boxes. Each box is defined in a map. The keys in the map are: the name at the bottom of the box, the height of the box and the weight of the box. All important variables are specified accurately with units and contexts in proper places. In this case the input has two boxes, "Box 1" weighing 3 kilos and "Box 2" at 8 kilos.

With all these definitions in place, the spec starts to look not only extensive but admittedly somewhat cumbersome for humans to write. The good thing is all of this can be autogenerated. The Smart API SDK will do all the writing for you. That list would be programmed using a standard programming language like Python or Java like using native data structures (e.g. ArrayList in Java). The library will output the correct format automatically and can also parse it back to the original object.

That's pretty much what you need to know for now. What we can recommend as a next move is to actually test Smart API by downloading the SDK and playing with it.

Anyways, congrats for reading through the document all the way here. That was pretty ballsy. So as a reward for all the attention and a job well done... https://www.youtube.com/watch?v=_W-fIn2QZgg.