English

Smart API Data Designer

User manual

ASEMA

Table of Contents

List of Figures

1. Introduction

1.1. The Smart API

Smart API is a technology designed for semantic interoperability of systems. It helps you connect two or more IT systems and make them understand, analyze and convert each other's data effectively and unambiguously.

Smart API comprises three key components

- The Smart API programming library (SDK)
- The Smart API data model
- Smart API services.

The Smart API Data Designer is a tool that helps in using those three components. You can use the Data Designer for browsing the data model, for adding extensions to the model, and for generating code to be used with the Smart API SDK and Smart API services.

This manual gives you an understanding on how to use the Data Designer. For instructions on the other components of Smart API as well as programming with Smart API, please refer to the corresponding manuals. Below is a short intro to each of these components, what you should be using them for, and how the Data Designer can help in their use.

1.1.1. Smart API programming library (SDK)

The Smart API SDK is an open source programming library available in several programming languages. As a free component, you can embed it into your own software project and make that Smart API compatible. The Smart API Data Designer will generate program code you can copy and paste directly into your software for easy transition.

To make connections compatible with each other, Smart API provides an extendable vocabulary of semantic data models and the core data model for describing connections and relations between systems. This model is used by the Smart API SDK to create the actual data traffic that passes between two systems. The process is automatic, all you need is to describe data for your project, and Smart API generates code for you to use in the system integration. Use the Smart API Data Designer to make such descriptions.

1.1.2. Smart API data model

Smart API is designed, as the name implies, as a smart data-oriented API for connections between computers. As designing API's for communication is nothing new, what is it then that makes Smart API a smart and valuable addition to the field? The short answer: Smart API is based on Semantic Web ontologies and standards.

The slightly longer answer is that with Smart API your data becomes knowledge. Not only is the data enriched and well presented with vocabularies, but also structured so that it is suitable for calculation and automatic deduction. Semantic data models are the key in making computers understand things like relationships, classifications and abstractions. With such models, computer programs can deduct or infer things from data without human assistance and do conversions between, for instance, different standard value units. The definitions of data made for this purpose are called ontologies. The Smart API Data Designer gives tools needed to extend those ontologies with concepts needed in your application domain.

1.1.3. Smart API services

Smart API services provide the supportive infrastructure for intelligent data exchange. Their use is voluntary. The API works perfectly fine even without contacting any of the services. But they do greatly help in the exchange by offering services such as a common repository for vocabulary terms and object classes, a secure notary for transactions, and a global search directory for data. And of course the Smart API Data Designer which is hosted as a part of Smart API Talk service.

With the Smart API Data Designer you can create new classes of objects which serve as global search terms in the Smart API Find directory. The new concepts added with the designer are also available for other users through the Smart API Talk server so that other system developers can make their designs compatible with yours.

1.2. The purpose of data design and the Smart API Data Designer

Smart API as a technology makes it possible to use various models and vocabularies for data exchange from software code. No explicit design effort is required from a pure technical point of view. If terms are inserted into the data from the software, it will work perfectly fine. So why would one use a data designer tool in the process?

There are two main goals for the Smart API Data Designer:

1. helping

2. sharing

Let's take a closer look at both.

Firstly, the Smart API Data Designer **helps** in the software development process. It generates sample code stubs that can be copy-pasted into the applications to shorten development time and lower the learning curve for the library. It also helps in grasping the industry knowledge and terminology. Coders don't have to be experts in the domain when the terms are clearly documented and explained by the tool. This helps in the staffing of industry specific applications.

Secondly, and most importantly, Smart API Data Designer makes it possible to **share** data models between organizations. It makes no sense to reinvent the wheel in every organization and it is tedious to make systems compatible if two organizations talk about the same thing but in different language. If one organization calls the rate of movement "speed" and the other calls it "velocity" and if in one organization to third dimension of a box is "length" while for the other it is "depth", system integration for sure bumps into trouble. The Smart API Data Designer is a common location where such language can be shared. The common vocabulary lets everyone browse for the appropriate terms to use. One designer can make a model of the data and share this model to with other parties, who in turn can generate software code that fits their particular system and handles the data without ambiguity.

Keeping that in mind, when designing the data for data exchange, two main questions arise:

- What is the data that is exchanged between the systems?

- What is the data that describes the systems themselves?

Usually some of both is needed. While exchangeable data naturally needs some structure, it is often necessary to know also details about the overall source and destination of the data (i.e. "data about data" or "metadata"). First, such metadata adds a context to the data. Metadata commonly includes for instance some type of classification of the source i.e. is the data from a car, a house or a medical sensor. You'll probably look at ground speed data differently when it originates from a space rocket than when it is from a bicycle. Second, the metadata may contain important technical details that makes the data transfer possible. Such descriptive data may contain for example information about connectivity: how does someone actually connect to the system, what is its address, what type of authentication is needed, etc.

Smart API data model provides an overall framework for both of the types of data described above. Because the computer processable data needs to be in some standard format for machines to understand it, the core design of Smart API data must remain constant. However, the model is purposefully designed to be as simplified and generic as possible to allow for maximum flexibility in various applications. You can change the application-specific parts of the data to make the model suit your particular application.

More specifically, with the Smart API Data designer you can:

- Define new terms that describe the data your application is exchanging.
- Author classes that characterize the systems and devices your application uses.
- Collect terms into objects which are shared between Smart API compatible systems.

Each data item in Smart API is an object and each object has properties that carry the values of the data. To modify the data, you design what these properties look like. Each property has three main characteristics: a quantity, a unit and a data type. To design a property, you'll need to pick these three from the already existing terms in the Smart API library or add terms of your own using the Data Designer. Then, with the help of the Data Designer, collect the terms you need and organize them as properties of objects.

The Smart API library has a vast collection of classes but your application may need a totally new class category or a specific extension to an existing class. For instance, the library already contains a thing called "a car" but if your application uses a specific car, let's say "a race car", you can extend the model. Using the Data Designer you add a class and then attach it to the systems as metadata.

Note
There is a lot more tech in the Smart API data model to make all this possible, but that is out of the scope of this document. If you intend to extend the model, it may be beneficial to take a closer look at the design. For more details about the Smart API data design philosophy, please refer to the separate Smart API core ontology model design document.

1.3. Designing the data

Semantic Web technology is the cornerstone of the Smart API. Using semantic data provides three main advantages:

- Self-described data.
- Standard vocabularies.
- Extendability.

A self-described data model allows very rich metadata with no need for additional schema definitions. In practice it means that you can create data that can be understood by other applications without being fully complete. Systems can deduct and fill in the pieces with the help of the data model.

Standard vocabularies ensure consistent interpretation. The vocabulary represents a common understanding of what each data item is all about. The data model includes human readable descriptions of what the data means and computer interpretable relationships that can be used to automate deduction.

The data model and the vocabularies are represented by an ontology which is the representation and formalization of the model. The ontology can be represented as a graph and stored into a file by using some format of RDF (Resource Description Framework). By adding items into these graphs, the model can be extended to suit more purposes.

Smart API natively supports a range of predesigned ontologies. The Semantic Web community has managed to agree on certain ontologies and vocabularies quite well, and thus many of the ontologies,

such as The Dublin Core (DC) and Good Relations (GR) can be considered standards. This is one of the key benefits of semantic data design: a lot of industry knowledge already exists and new designs don't need to start from scratch. Smart API Data Designer and the Smart API Talk server — which hosts the designer tool — allow such cooperation between designers.

When you need to use some concept while developing an application, the recommended first step is to always check whether it has already been defined. If not, you can either import an existing ontology that defines this concept, or you can use Smart API Data Designer to add the concept. The instructions in this manual help you in the various customization options which include:

- Using an existing ontology i.e. something already made by someone else.
- Importing ontology you designed previously.
- Extending ontology i.e. modifying previously created ontology.

1.3.1. Designs for data exchange

A common challenge in providing compatibility between IoT systems is that different people and organizations use different terms and measurement units for the same thing. While 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. The purpose of vocabularies is to remove such ambiguity and, in case different terms are used for the same measurement, offer tools to convert the values from one term to the other.

Now, to make the two systems understand each other in practice, what you'd need to:

1. Create a data model (in the case of our example, let's call it "a box")
2. Add a term to the model (let's call it "boxlength") and an explanation that says this term means the longest dimension of a box.
3. Define the measurement units (for example, inches and centimeters) and conversion factors between them if necessary.

That said, Smart API provides a pile of vocabularies of data models, so it is highly possible that you can just pick up an existing one for your data. For instance, the conversion between inches and centimeters is already there.

1.3.2. Designs for describing systems

When you describe a system, the most basic thing is to assign a class to it. So, instead of being just an anonymous object, the system can be classified as what it represents, measures, or controls. For instance, it could be a thermostat, a car, a house, a factory, etc.

Next, an equally important thing is to attach an identity to it. In Semantic web and in Smart API, identities are represented by URI's and start with the domain of the owner or manager. So a car owned by a taxi firm could be, for instance, http://www.mytaxicorp.com/vehicles/nyc-abc-1234. This identity makes it possible to point to that car from any system without ambiguity.

Finally, as Smart API is meant for IoT applications, you should say what the particular object can do and how it can be contacted. This involves the definition of

- `activities`(i.e. what types of services the object hosts and what it can do for you).
- `properties`(i.e. what type of data the object and its actions may take as input or give as output).
- `interfaces`(i.e. how in practice the object may be contacted in order to access an activity).
- `relations`(i.e. what this system is in relation to other systems e.g. does it host or manage some other system).

The descriptive data can be used to register a system to Smart API Find. This directory builds a map of various systems and checks how they link together. The resulting graph can be queried to find IoT objects with using advanced search such as connections, geography, capabilities, etc.

Registration code needed to do such registration of your system can be generated with the Smart API Data Designer.

Learn more about working with data models in 4, Designing data.

1.4. Generating code for the system integration

Once the data models are defined, you can connect systems together.

If you use Asema IoT software, you can create a schema file. A schema defines how to retrieve property values from the data flow received from a device or another data source. Also, a schema defines how to convert the received data. Once you download a schema file, go to the admin interface of Asema IoT Central, create a new object and when defining the data schema for it, choose to import the definition. Once imported, you have a Smart API compatible object to use in IoT applications.

If you program your own application, you can generate sample code for read-write requests to use it in Smart API connections. The Smart API Programmer manual tells you all the details on how to use such code so that is a recommended read if you haven't taken a look at it already.

For a shorter tutorial on how to generate the code samples and an intro on what they do, go to the code example sections.

2. Terminology

2.1. Data model

A data model is a set of data that describes an object, for example "vehicle". It provides definition, describes properties (such as "speed", "length") and units in which the properties are measured (for example, "kilometers per hour", "meters").

A data model can include the following concepts:

- A class of things that are characterized by the data. For example, devices, services, activities, messages. Classes may have subclasses.

  A class or subclass name always starts with a capital letter. Each class or a subclass is accompanied with a description.

- Properties and quantities of a class or a subclass. Quantity is a measurable property. For example, air temperature and car speed are quantities.

- Units in which quantities are measured (centimeters, meters per hour, degrees, and so on).

2.2. Ontology

An ontology is a data model that represents a set of concepts within a domain and the relationships among those concepts. Relationships between the ontology concepts are expressed in triples "<subject> <predicate> <object>". For example, "kilometer per hour is a linear velocity unit", "a car is a vehicle", "a vehicle uses a road" and so on. Notice the format of those definitions where there is always some subject ("a vehicle"), some predicate ("uses"), and some object ("a road").

In the case of everyday Smart API data modeling, ontologies are most commonly needed to describe object quantities and the units in which they are measured. For example, if you use Smart API for an app that collects information about car speed, you would need an ontology that describes the car speed and the units in which it is measured (for example, kilometers per hour). An ontology can also indicate the conversion rate, so that your app could use it to transform kilometers per hour in, let's say, meters per hour if needed.

That said, Smart API also contains a vast array of definitions where relationships are of other nature than just quantities and units. But that is the topic of another manual.

2.3. Quantity

A quantity is a measurable property of a particular object. It is always associated with the context of measurement: the thing measured, the measured value, the accuracy of measurement, etc. For example, speed of light in a vacuum, car speed, blood pressure and atmospheric pressure are quantities.

A quantity is characterized by quantity kind. Quantity kind is a more abstract notion, it identifies the physical nature or type of measured quantity such as speed, pressure, length, frequency. Different quantities can be of the same physical nature i.e. of the same kind. For instance, speed can be measured in relation to something so there can be for instance quantities ground speed (movement in relation to earth surface) and air speed (movement in relation to air mass) which are of the same nature i.e. kind. A quantity is usually measured in the same units as its quantity kind.

2.4. Unit

A unit of measure is a quantity value chosen as a scale for measuring other quantities the same kind. For example, the meter is a quantity of length. Any measurement of the length can be expressed as a number multiplied by the unit meter.

2.5. Concept

Concept is a generic term for classes, properties, quantities, units etc. — everything that is defined in the data model.

2.6. Class

Classes are groups of things that are described by data models. They are metadata that describes data. For example, vehicles, devices, houses, and weather are classes.

Classes can have a hierarchy and therefore inheritance and multi-inheritance. For instance a rose is a flower, a flower is a plant, and a plant is a living organism.

2.7. Vocabulary

A vocabulary is a set of definitions of variables you use for your data. It solves a common problem: if you have some variable you want to put into your data, how would you name it so that everyone else understands it? Let's say for example that you want to represent the speed at which a vehicle drives. Would you call this "speed" or "velocity" or "groundSpeed" or something else?

A widely used solution for the problem is to pick some term that sounds sufficiently appropriate for the case and then create a document that explains your choice. In APIs this is commonly part of an API doc. So you have some separate document that says there is a property called "velocity" which measures the speed of a vehicle in relation to the road. But this is quite tedious. First, you need to distribute that API doc. And second, there is no way to automate the mapping and processing. Which means that everyone who uses your terms needs to read the documents term by term and manually build various converters. And once you're past that hurdle, comes the question of units. Is that speed expressed in mph, kph, m/s or what is it?

With a vocabulary, we can jointly agree on such terms. So when someone measures speed, let's all call it "groundSpeed" and measure it in meters per second. If everyone sticks to that vocabulary, there is no need for separate documentation and conversion. But that sounds utopistic to say the least. There is no way we could a priori agree on a fully generic language that covers everything we'd like to define in the future. And then make absolutely everyone follow that. This is why vocabularies are dynamic and expandable. They can be tailored and customized by each party to fit various use cases.

Now that is an improvement and makes vocabularies a generic technology for solving compatibility issues. This approach does create further issues to solve. First, what if there are two parties that do call vehicle movement groundSpeed but use it with different meanings and different units? And second, that if two parties want to talk in completely different levels of accuracy. One needs to know ground speed, air speed, and descend rate while for the other just some velocity would be fine.

To solve the issues, a proper vocabulary first separates different definitions by domain. Technically this means that the term is expressed with a URL, say http://www.commonterms.org/velocity#groundSpeed. That full URL reduces the risk of conflicting definitions. The beginning of the URI, in this case http://www.commonterms.org/velocity is the so-called namespace. If two names are in different spaces, they do not conflict.

The URL is usually replaced by a prefix to make the representation shorter. For example `commonTerms:groundSpeed`. And if you have a differing meaning for it, that would be `myTerms:groundSpeed`. The useful thing about URL's is that you can actually point with them somewhere, in the case of vocabularies it points to the definition of the term. So if someone browses http://www.commonterms.org/velocity#groundSpeed, that link would return text that explains what groundSpeed is. No need for separate API docs, everything is conveniently online.

Further, vocabularies can do inheritance and classification. This is very similar to inheritance in object-oriented programming. Let's for instance assume you are making objects that are all boxes. So the basic type would be for example `myproducts:Box`. Some of these boxes are designed to be useless

toys, some useful tools. Which means the box has two subcategories `myproducts:UselessBox` and `myproducts:Useful`. Let's further divide the useless boxes to `myproducts:Toybox` and `myproducts:DecorativeBox` and the useful boxes have a subcategory `myproducts:Toolbox`. In plan data exchange such classifications have limited meaning but when it comes to search, they are invaluable. For instance if you want to find all useless boxes, you don't need to search for both `myproducts:Toybox` and `myproducts:DecorativeBox` but instead can perform one search with `myproducts:UselessBox`. Automatically processing such hierarchies is the core of semantic tools, they automatically understand the classifications and can greatly help in data management.

3. Signing up and logging in

Smart API Data Designer is a web-based tool. To use it, open your favorite browser and go to http://talk.smart-api.io/develop/

You can use Smart API workbench without creating an account but it is recommended to register and log in to be able to save your work.

To create an account, fill in the registration form. You'll find the form behind the "signup" link in the top toolbar of the Data Designer or at the entry page of the Smart API developer portal.

When a new user account is created, it has to be linked to a namespace prefix. Because data descriptions can differ between organizations, a prefix is needed to mark the data models you create. If you already have such prefix, enter it in the Prefix field. Otherwise, provide your organization name and the prefix will be generated for you.

4. Designing data

4.1. Using predefined vocabularies

The existing set of data models in Smart API covers most common needs for defining data. To avoid duplication, we recommend searching for the existing concepts from the models that are predefined in the service before creating a new concept.

The list of vocabularies currently available in the service:

- The Dublin Core (DC)
- WGS84 Geo Positioning (GEO)
- Good Relations (GR)
- The OWL 2 Schema vocabulary (OWL)
- The RDF Concepts Vocabulary (RDF)
- The RDF Schema vocabulary (RDFS)
- The Smart API ontology (SMARTAPI)
- The NASA QUDT Units Ontology (UNIT)
- The NASA Quantity - Unit - Dimension Ontology (QUDT)
- The NASA QUDT Quantity Ontology (QUANTITY)
- Ontology for vCard (VCARD)

4.1.1. Searching predefined concepts

To search among the existing data models and view their details:

1. Open the data design workbench: http://talk.smart-api.io/develop/designworkbench.
2. Start typing the key word in the search field. You can then choose one of suggested options.
3. Click the info icon to view the concept details.
4. To add these terms to your workbench for later use in applications, click the plus icon.

Once you have a set of semantic concepts, you can choose the approach forward: autogenerate source code to use in your application, manually write code for the concepts to your Asema IoT object definition or generate a schema file to be imported into Asema IoT.

4.1.2. Filtering predefined concepts

To limit the concepts displayed, use the search filters (the filter icon next to the search field):

- By vocabulary.
- By the type of concept (quantity, unit, class, property).

4.1.3. Browsing all predefined concepts

To browse all existing semantic concepts, open the data design workbench: http://talk.smart-api.io/develop/designworkbench and click the cube icon and then choose what you want to view:

- All classes (such as vehicles, pumps and so on).
- All properties (such as depth, speed and so on).
- All items, that is classes and properties in one list.

To add a concept to your data design, click the plus icon next to it.

4.2. Creating a custom data model

Smart API is designed to be a technology that helps in standard communication between parties. You can customize it to fit a particular application but there are still some rules. Because Smart API aims for standardization, there are certain principles that must be applied to maintain the standard. While there is considerable freedom in ontology handling, the system cannot allow simply everything as standard communication cannot happen without structure.

The first rule in customization is that the Smart API Talk server should be kept as the common repository for all designs. The tools offered by the server ensure that conflicts don't happen, ontologies can be developed jointly, and developers get the definitions for their software in a unified manner. Now, it is technically possible to keep your ontology off the server and just link it. That is what the URI's in ontologies are meant for. However, if the ontology is not at Smart API Talk, the ontology is not included in the search and IDE tools nor checked for conflicts. This will make using it more difficult for others. So while it is perfectly fine and often advisable that you can design and develop an ontology outside the server, always upload the latest version into the service.

Second, users should pay attention to namespaces as they may not conflict. When you edit an ontology **in the Smart API service** you will automatically be assigned a prefix and a namespace. All assigned namespaces start with http://smart-api.io/ontology/ and end with the prefix. So for instance, if you'd make an ontology for Acme Inc, the prefix assigned (if not yet taken) would be "acme" and your namespace http://smart-api.io/ontology/acme. Similarly, someone working for Coyote Corp would use the namespace http://smart-api.io/ontology/coyote. Note that when you are assigned a namespace, you also automatically become the administrator of that namespace. The administrator has the rights to accept and reject changes in that namespace.

4.2.1. Rules applied to imports

When you **import** an ontology, the ontology is checked and the triples in it are analyzed for namespace conflicts. A namespace conflict occurs when:

- The namespace you try to import matches a namespace already defined by some fixed ontology in the Smart API standard.
- The namespace you try to import is the same as a namespace that has already been assigned by the Smart API Talk service to some other user (irrespective of whether this namespace is empty or not).
- The namespace you try to import matches a namespace that is included in an ontology imported by some other user.

If an imported triple **does not conflict** with anything, it is added into the main graph of the system under that namespace. If the namespace does not exist yet, it is created and you will be assigned as the administrator of that namespace. So for instance, if there is an ontology for cars, say http://vehicles.org/ontology/cars and you create an ontology for motorcycles http://vehicles.org/ontology/motorcycles, you will become the administrator of all edits to http://vehicles.org/ontology/motorcycles.

Needless to say, if you are not working on a very generic ontology that you are sure will benefit everyone else, you should keep the namespace in a domain with your organization name in it. For example, something like http://mikesbikes.com/ontology/motorcycles.

If an imported triple **conflicts** you will be prompted to make a choice before the import is finalized:

- Accept the import, in which case the conflicting terms will be assigned to a new namespace.
- Cancel the import, in which case nothing will be imported (not even the non-conflicting terms).

If you accept the import, the namespace that is assigned will be the same namespace that is assigned to you by the Smart API service. If this causes conflicts with the concepts you edited previously in the service or with some other ontology you have already imported, the import will be canceled.

In case the import is canceled due to you declining the prefix assignment or because of further conflicts, you will need to modify the ontology file to remove the conflicts or remove previous imported ontologies or edited concepts.

So, for example, there already is an ontology in the system for motorcycles, say http://vehicles.com/ontology/motorcycles and you try to submit a new item with the same namespace, say, http://vehicles.com/ontology/motorcycles#Bugatti this will either rejected or put into your namespace. If in this case you'd work for our example organization Acme Inc, that term would become http://smart-api.io/ontology/acme#Bugatti.

4.2.2. Rules applied to online edits

When you **online edit** an ontology using the tools, all your edits will be placed under your namespace. Because the namespace is unique, this guarantees that there are no conflicts in the namespace. When you save the edit, it will be checked for namespace conflicts against the data you yourself have either imported or edited previously.

While namespaces don't conflict, you may be trying to add a term that is already in the data you have previously added with online tools or imported. If a conflict is detected, you will be prompted with options on how to proceed.

4.2.3. Importing an ontology

If you have a data model saved in a file formatted as Turtle, JSON-LD or N-Triples, you can upload it to Smart API Data Designer. Click the cloud icon and choose the file.

If the imported data models don't duplicate the existing ones, they are added under your prefix to the list of concepts.

Once the ontology is imported, you can use it in your code, pass it through the validators and generate documentation based on the ontology. In the beginning of your code, add the custom namespace into the namespace manager and then refer to it in the code. For example, in Java:

```
NS.addPrefix("acme", "http://smart-api.io/ontology/acme#");

ValueObject outTemp = new ValueObject();
outTemp.setQuantity("quantity:ThermoDynamicTemperature");
outTemp.setUnit("unit:DegreeFahrenheit");
outTemp.setValue(82.5);
dev.add("acme:outsideTemperature", outTemp);
ValueObject inTemp = new ValueObject();
inTemp.setQuantity("quantity:ThermoDynamicTemperature");
inTemp.setUnit("unit:DegreeFahrenheit");
inTemp.setValue(70);
dev.add("acme:insideTemperature", inTemp);
```

4.2.4. Creating a concept

To create a custom concept online, click the branch icon and fill in the fields depending on the type of concept.

Use new concepts in your code and remember to add prefix mapping for your custom concept. For example in Java, your code would look something like this:

```
NS.addPrefix("acme", "http://smart-api.io/ontology/acme#");

ValueObject outTemp = new ValueObject();
outTemp.setQuantity("acme:OutsideTemperature");
outTemp.setUnit("unit:DegreeFahrenheit");
outTemp.setValue(82.5);
dev.addValueObject(outTemp);
ValueObject inTemp = new ValueObject();
inTemp.setQuantity("acme:InsideTemperature");
inTemp.setUnit("unit:DegreeFahrenheit");
inTemp.setValue(70);
dev.addValueObject(inTemp);
```

4.2.4.1. Creating a Quantity

A quantity is a measurable property of an object. It is associated with the unit of measure. For example, length, speed, temperature are quantities.

To create a new quantity, fill in the fields:

- Name

  The quantity name. The name must be unique and descriptive.

  According to the naming convention , quantities (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "LinearVelocity", "LiquidVolume", and so on.

- Label

  A label to use for the quantity in the application.

- Comment

  Description of the quantity.

- Quantity category (quantityKind)

  The type of quantity (select it from the drop-down list). For example, for the volume quantity the quantity category would be qudt:SpaceAndTimeQuantityKind.

- Parent quantity

  A parent quantity is a generalization of the child quantity. Choose the parent quantity from the drop-down list. Make sure that the chosen parent quantity can be measured in the same units as the child quantity. For example, for the liquid volume quantity, the correct parent quantity would be "Volume".

4.2.4.2. Creating a Unit

Units are used to measure quantities. For example, kilometers, yards, liters, kilograms are units.

To create a unit, fill in the fields:

- Name

  The unit name. The name must be unique and descriptive. .

According to the naming convention , unit names (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "CentimeterPerSecond".

- Label

  A label to use for the unit in the application.

- Comment

  Description of the unit.

- Abbreviation

- Used with quantity

  Quantity associated with the unit.

- Unit Category

  The type of unit (select it from the drop-down list). For example, for the "Liter" unit, the unit category would be "VolumeUnit".

- Conversion multiplier

  A multiplier to convert the current unit to the corresponding SI unit.

- Conversion offset

  An offset to convert the current unit to the corresponding SI unit. For example, to convert Celsius to Kelvin, the offset is 273.15.

4.2.4.3. Creating a Device type

Type of device.

- Name

  The device type name. The name must be unique and descriptive. .

  According to the naming convention , device types (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "WaterPump".

- Label

  A label to use for the device type in the application.

- Comment

  Device type description.

- Parent class

  Select the parent class for the device from the drop-down list.

4.2.4.4. Creating a Service type

Type of service.

- Name

The service type name. The name must be unique and descriptive. .

According to the naming convention , service types (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "BikeRent".

- Label

  A label to use for the service type in the application.

- Comment

  Service type description.

- Parent class

4.2.4.5. Creating a Data type property

Data type, for example string, integer, float number.

- Name

  The data type name. The name must be unique and descriptive. .

  According to the naming convention , service types (as all properties, i.e. something that goes into a predicate) are written in mixed camel case. For example, "indexedArray".

- Label

  A label to use for the data type in the application.

- Comment

  Data type description

- Range

  Possible values of the data type. For instance, a positive number has only values from 0 upwards. Negative numbers are not allowed.

- Domain

  The domain to which the data type belongs.

4.2.4.6. Creating an Object property

A property that describes relationship between concepts, for example, "isClassOf" is a property that states that something is a subclass of a class. Another example is "opens", a property stating that something opens at the specified time.

- Name

  The object property name. The name must be unique and descriptive. .

  According to the naming convention , object properties (as all properties, i.e. something that goes into a predicate) are written in mixed camel case. For example, "hasBrand", "open".

- Label

  A label to use for the object property in the application.

- Comment

  Object property description.

- Range

  Possible values of the object property.

- Domain

  The domain to which the object property belongs.

4.2.4.7. Creating a Unit category

Unit category that describes a unit, for example "LinearVelocityUnit", "VolumeUnit"

- Name

  The unit category name. The name must be unique and descriptive.

  According to the naming convention , unit categories (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "TemperatureUnit".

- Label

  A label to use for the unit category in the application

- Comment

  Unit category description.

- Parent unit category

4.2.4.8. Creating a Quantity category

Quantity category that describes a quantity (see 2.3, “Quantity”).

- Name

  The quantity category name. The name must be unique and descriptive. .

  According to the naming convention , quantity categories (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "SpaceAndTimeQuantityKind".

- Label

  Label to use with the quantity category in the application.

- Comment

  Quantity category description.

- Parent quantity category

4.2.4.9. Creating a General class

General class that includes subclasses. For example "food" is a general class that can include "eat" and "drink" classes.

- Name

The class name. The name must be unique and descriptive. .

According to the naming convention , classes (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "FoodAndDrinks".

- Label

  Label to use for the class in the application.

- Comment

  Class description.

- Parent class

4.2.4.10. Creating a Class member

Class member is a resource that has been placed into that class (also sometimes called an instance of the class).

- Name

  The class member name. The name must be unique and descriptive. .

  According to the naming convention , class instances (as all resources, i.e. something that is a subject or object) are written in upper camel case. For example, "MyBugattiMotorcycle".

- Label

  Label to use for the resource in the application.

- Comment

  Resource description.

- Class member of

  The class of the resource.

4.2.5. Edit a concept

Sometimes when you try to use some existing ontology, it seems that it would be perfect for your application — except for some missing detail in it. If only the designers had added that thing in it, that would be everything needed. So for instance replacing the whole thing sounds like a waste of time — and often is. However, you may not edit an ontology unless you are its administrator. The assigned administrator is always the party who first claimed the namespace(s) used in it. To get rights to edit, you'll need to contact the author and ask for a permission. Tools for this can be found in the developer portal. The current administrator can choose to proceed in two ways: giving parallel rights to edit the ontology on their behalf or transfer the management rights fully.

If you don't have an ontology file or would simply want to add a couple of new terms into the system, you can do that with online forms found in the developer tools.

4.2.6. Request a custom feature

The Smart API follows a certain structure and logic, especially when it concerns `Requests`, `Responses`, and `Activities` inside them. These are critical for network communication to work. As `Entities`

can be abstracted to just identifiers and bundles of properties, this gives quite a bit of flexibility for application design. But the flexibility comes at the expense of structure and clarity when it comes to very specific applications.

This lack of clarity can usually be remedied with a custom ontology, nothing prevents from modeling a given property in some certain way. So this is the primary means to add structure. That said, however, there may be various reasons why the current structure of Smart API is simply not enough for some application. This is why the technology is being continuously developed and improved.

If it is clear that your application would need a change in the core structure of Smart API, you can always request it. To do this, open the Submit a model feature request form from the main icon bar of the data design workbench. Then simply submit your request as a feature ticket. It will be recorded into the feature ticketing system and assigned a schedule if the request is feasible.

5. Deploying the model to applications

5.1. Schemas

The primary target for the data model is software code implemented with the Smart API SDK. When integrated to a software application, the SDK's libraries will make the target system "talk" in Smart API. But even if you are not a programmer who wants to use such tools, worry not. Asema IoT software suite supports semantics out of the box and you can import your model to the software. This is done with schemas.

An object schema in Asema IoT basically defines the structure and functioning of an object. The philosophy there unsurprisingly follows the design of Smart API: all data is represented by objects and their properties. So as the Smart API Data Designer can be used just for that, designing objects and their properties, moving the design to Asema IoT is straightforward.

To get the schema file:

1. Open the design of the object in case.
2. On the left toolbar of the Samrt API Data Designer, open the guilabelView menu and choose the Schemas icon.
3. Copy the example to your clipboard.

Once you have the schema in your clipboard, go to the admin interface of Asema IoT Central, choose the type of object you are looking to create and then open the form to create a new schema for this object type. Now, just paste the content of the clipboard into the schema editor and save. Attach the schema to the new object(s) that you create. You now have a Smart API compatible, semantics enabled object to use in IoT applications.

5.2. Code for Smart API SDK

To make an application Smart API-compatible, you can program an API interface with the Smart API library. For this purpose, you can either generate a sample code stub with the Smart API Data Designer and start from there or simply author the code from scratch by yourself.

> **Note**
>
> For instructions on how to program with Smart API SDK and embed the code into your application, please refer to the Smart API Programmer Manual.

To generate sample code:

1. Open the design of the object in case.
2. On the left toolbar of the Smart API Data Designer open the View menu and choose the Code examples icon.
3. The code examples tool opens with options in the top bar.
4. From the language tabs on the left you can choose in which programming language the sample is displayed.
5. From the dropdown on the right you can choose which type of sample is displayed.

> **Note**
> The type of samples will depend on the workbench type. The server workbench displays server samples, the client workbench client samples and the notifier workbench the notifier samples.

5.2.1. Server code examples

Server side code runs as the name implies on a server and responds to requests from a client. The sample code shows how to parse the incoming request and pick the Activities from it. Each `Activity` represents a "service" on your server and this will determine how to actually process the data.

The sample cannot naturally display the logic of processing the data as this will depend on you application. But it does show how to extract parts of the request and handle the objects that represent the incoming data.

If there is a convenient small class that represents of HTTP service (such as Pythons WebPy), the sample contains code to start the server and is therefore self-contained, readily runnable code which you can start. In most environments this is however not the case and you will be running a specific web server / application server. In such case just take the request handler as an example and apply it to the particular technology you are using.

5.2.2. Client code examples

The client code examples show how to create a Smart API `Request` and place and `Activity` in it. In Smart API the `Request` represents a call to a server and the `Activity` a service inside the server.

The samples then show how to manipulate or read objects in the service. Each is represented by an `Entity` which is an abstraction of some item in the service. The design of Smart API is based on the idea of templates and methods. If you read an `Entity`, you add the properties you want to read into the `Entity` but without values. This is a "template" which tells the service to fill in those values. If you give no properties at all, the service should try to fill in all the properties that are known.

Writes of `Entities` work in a similar fashion: to write a particular property, include it in the `Entity` with a value. That value will be set to the `Entity` if applicable.

Adding a `TemporalContext` to the read of write operation means that you want to work on a series of values and that series is determined by the parameters of the `TemporalContext`. For instance a `TemporalContext` that defines a date range from January to June in a read operation will return values of a property from January to June.

5.2.3. Notifier code examples

Notifier code examples show the functioning of two sides of notification logic: publishing (sending) notifications and subscribing to (receiving) notifications. Unlike a client-server model, a notifier is single directional. A notification is simply sent and the subscribers receive it but do not respond.

The easiest way to work with notifications is to use the `EventAgent` helper class of Smart API. This hides all the details of the underlying pub/sub protocol (by default MQTT) and offers a simple callback structure that processes notifications when they arrive. The sample code shows how to extract the Activities from the incoming notifications and then take the data contained inside the Activities. On the publisher side the samples show how to pack Entities into the notifications and send them.

In notifications, Smart API follows the same templating idea as in client-server writes. The publisher includes into the notification an `Entity` and fills in the details of those properties it considers worth notifying. The `Entity` may also have other properties but if there is nothing to notify (e.g. no change in value), these are left out.

5.2.4. Code testing

While the Smart API SDK, and especially the Agent classes, construct many of the necessary communication structures automatically and the semantic format helps in interpreting data even when it is

incomplete, it is still possible to have various errors and omissions in a Smart API implementation. This is why testing is essential.

To test the standards compliance of a Smart API implementation, the Smart API Data Designer also features a testing service. You enter this tester from the toolbar icon at the very top left corner of a Data Designer workbench.

The tester acts as a test endpoint to various types of code. It can act as a server for client code, as a client for server code, and as a subscriber for notifier code.

To use the tester, open the tester from the icon and then follow on-screen instructions to activate the test.

Because of the semantic model and the ability to interpret incomplete data, the tester will not give you just a works / does not work result. Instead, the code is graded. If the communication follows the best practices and has all the recommended fields, the result will be close to an "A" whereas a code missing these will get a grade closer to an "F".

Appendix A. Quantity-Unit relationship in QUDT ontologies

Figure A.1. Links from units to quantities

Figure A.2. Adding custom units and quantities