Smart Object API and Architecture

Data Models for the Internet of Things Part 4

Michael J Koster
August 30th, 2012

*The goal is to create a web ecosystem of **sensing**, **reasoning**, and **action** around the Internet of Things.*

In the first post I talked about the current Internet of Things architecture as consisting of sensor nets connected to web services in a mostly vertical fashion. That is, there are systems consisting of sensors, a gateway, and a cloud service where the gateway interacts with the cloud service in a specific or even proprietary fashion. Likewise the API for adding applications to the service is either private to that service or at least a service-specific API. There is a different API for each service.

Yet the interactions between gateway and service, and the affordances the service provides to applications are mostly the same. These gateways interact with the cloud service using web capable RESTful interfaces. There are common patterns emerging for these interfaces and APIs

There are many new sensors, from personal fitness trackers to buddy lamps that turn on and off together halfway around the world. To bring each new idea to market requires a lot of plumbing to be reinvented, mostly from standard parts and patterns. When they're done, they don't talk to each other. There's practically no way to second source a service or to create an application that can arbitrarily mash up data streams. A connector is needed for each service and the only access is to the stored datastream. Cosm, Nimbits, Sen.se, ThingWorx have different APIs that all do the same things. There is no standard way for gateways to interact directly with each other or with user devices like smartphones.

The opportunity is for a simple, open set of constraints and conventions around the existing pattern using sensor nets, gateways, and web services. A standard web API and M2M protocol will allow gateways, smart sensors, services, and user devices to interact with each other at the granular, scale-less level of "things".

The Smart Object is a Semantic Web application for the Internet of Things

In the earlier posts, I describe a Smart Object as a self-describing internet representation of a thing of interest. It's a generalization of the concept of a sensor that can interact directly with services.

The Semantic Web is a set of tools that enable semantic discovery and linkage of web objects pointed to by URLs. These tools can be used to enable content-based discovery and linkage, which is necessary to support interaction at a granular scale within a large system.

In the context of this discussion, a Smart Object is an entity, pointed to by a URL, that encapsulates some Observable Properties along with their Semantic Web descriptions, and application software agents with a RESTful API. An Observable Property can be anything from a sensor reading to an arbitrary document.

It's all about the API

The Smart Object API enables semantic discovery and linkage, access to Observable Properties, and control of application agents.

Observable Properties from one Smart Object to another are linked by a subscribe/notify method that optionally generates callbacks to application handlers.

The API uses named and described elements inside the object to interact with the Smart Object using a simple RESTful interface. In this way an agent or service proxy wishing to interact with an object's Observable Properties can discover enough information to connect Observable Properties to algorithms and actuators.

The Smart Object API looks the same to an agent inside the object as it does on the RESTful web interface. M2M interactions between an agent at one node and an object on another over http consist of simply generating, exchanging, and scanning document fragments. Agent software can in this way be plugged into data streams based on semantic compatibility and linkage criteria.

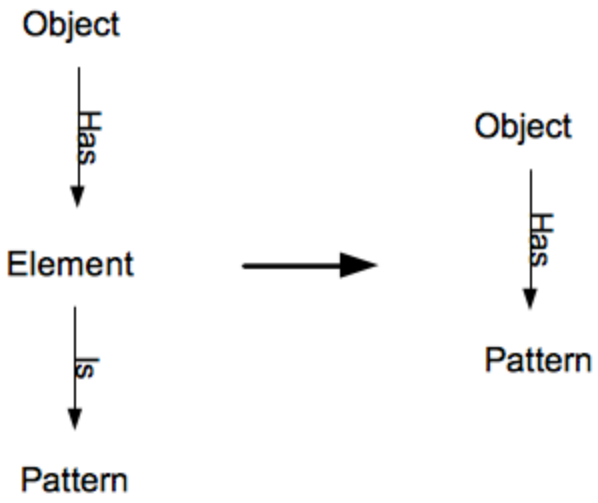
Architected Constraints for a base Smart Object Semantic Model

This is a description of a semantic reference model for a Smart Object, and a list of API methods based on that reference model. This defines the API and reference architecture of a Smart Object.

The model is object oriented and extensible. The elements define their own methods and interfaces. The elements are chosen to allow simple idempotent operations with storage semantics, i.e. a CRUD style interface using REST principles.

I'm using a form of declarative constraints to define the model where the basic form is a graph showing linked entity-relationship triples to define patterns. Here's an example:

Declarative Object Pattern



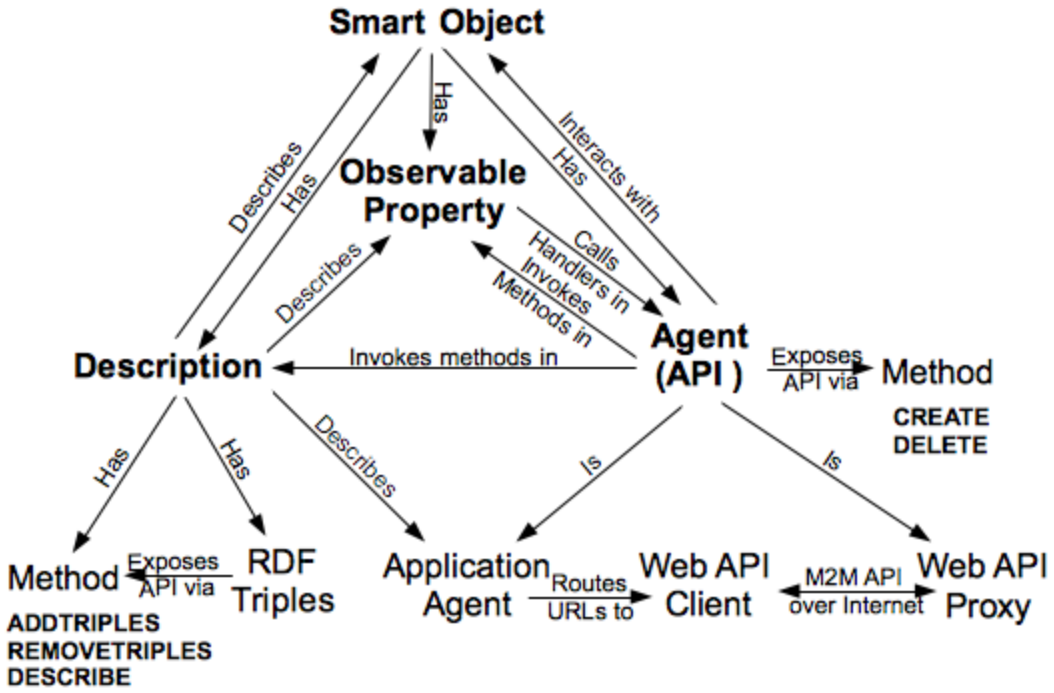
An object "Has" a element (one or more) which "Is" a Pattern is more precisely "An object **has** contained within it a top level named element, which **is** an instance of class "Pattern"

A shorthand is Object **has** Pattern meaning one or more instances of a Pattern.

Smart Object Pattern

Here is the current proposed Smart Object Pattern:

Smart Object Pattern



This shows the constrained relationships between the SmartObject's top level elements. The Smart Object is pointed to by a URL and a path to each method is exposed e.g. :

```
URL = Object.Description.ADDTRIPLES(triples);
```

The Observable Properties are the representations of individual things at any granular scale level, from say a temperature reading to an entire document. They are also self-describing objects.

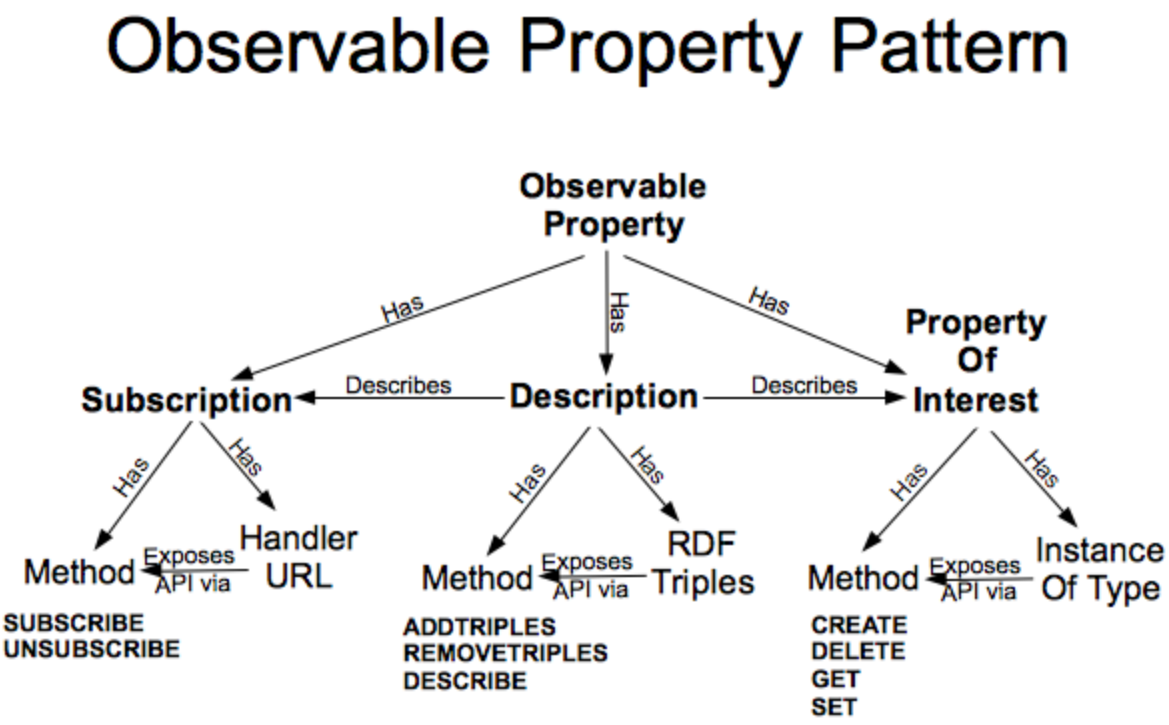
The Description contains RDF triples describing the Object's top level elements; The Observable Properties and Agents, and overall description of the Object itself.

The Agent pattern shows how Application Agents and the Web API agent (web proxy) see the same underlying methods, thus have the same API, local or remote. An Agent can be either the Web API proxy for Web clients, plus any number of separate Application Agents that can be created within the scope of the Smart Object.

Agents have a Web client they use for interacting with Smart Objects on other services, gateways, user devices, etc. using a RESTful M2M protocol over http. The client and proxy exchange document fragments representing the remote API method invocations and responses.

Observable Properties and Agents can be dynamically created and deleted after the base instance of a Smart Object is created.

The Observable Property is itself a self-describing Object with the following structural constraints:



The Observable Property has the same Description pattern as the top level object.

The Property of Interest is the container for the actual data or data stream, and allows custom implementations and methods to be defined by each Property of Interest. There is exactly one instance of Property Of Interest for each Observable Property, but it can be of any type including compound types or documents.

The Subscription is how data push on change or other asynchronous notification is performed. SUBSCRIBE adds a URL to a set of handlers to call back on updates to the Property of Interest. The handlers can be to the local agent, or to a URL pointing to an Observable Property on some other Smart Object to push data or notification to.

API Summary

Base Smart Object API Methods

Methods to manage ObservableProperties

- **CREATE(newOP)** – Make a new ObservableProperty or Agent Element
- **DELETE(OP)** – Remove Matching ObservableProperty or Agent Element

Methods to interact with the PropertyOfInterest:

- **GET()** – Returns ObservableProperty.PropertyOfInterest
- **SET(NewPOI)** – Replace or Update the PropertyOfInterest
- **SUBSCRIBE(URL)** – Request notification-on-change of PropertyOfInterest to URL
- **UNSUBSCRIBE(URL)** – Remove Push-On-Change Subscription matching URL

Methods to interact with Descriptions:

- **DESCRIBE(RDF)** - Returns RDF from element.Description() matching RDFspec
- **ADDTRIPLES(RDF)** – Add RDF triples to Description
- **REMOVETRIPLES(RDF)** – Remove triples matching RDFspec

Mapping to CRUD methods

Since each element has a unique path to it, e.g.

Object.ObservablePropertyX.GET()
Object.Description.DESCRIBE()

A form of the GET verb can be used for both without confusion:

Object.ObservablePropertyX.GET()
Object.Description.GET()

ADDTRIPLES and REMOVETRIPLES can map to CREATE and DELETE respectively.

Likewise, SUBSCRIBE can be though of as the CREATE of a subscription, and UNSUBSCRIBE is a DELETE.

Each method in the Smart Object API can be mapped to one of the four CRUD methods relative to it's element.

Summary

The Smart Object is a prototype Semantic Web API and application framework for the Internet of Things.

It is a common data model and API for Internet Gateways, Web Services, and User Devices to use for interacting with each other spontaneously at the granular scale of connected Things.

Semantic discovery and linkage is enabled by an interface that attaches semantic tags dynamically to named resources within the object.

Every element instance is self-describing. The "data" element resolves to a self-describing instance of a type, allowing complete flexibility in data representation.

An instance of a Smart Object is pointed to by a URL and includes the object's Observable Properties, it's Description, and it's Application Agents.

Observable properties of a Smart Object can be Subscribed to by other other Smart Objects for Notify-On-Change or Data Push operation.

The Smart Object API looks the same to a local Agent as to a Web-connected Agent, allowing transparency of location. Only the URLs are differently routed.

Web services can be second-sourced, or mashed up with other services at the data algorithm level.

Gateways and Smart Sensors can interact directly with actuators and user devices when it makes sense, using the same agents and APIs. Smart Objects and their services can be migrated and replicated easily for total system ruggedness.

Next up

As I build a prototype Smart Object and deploy a test service, I'll be looking at the high level ecosystem again. A general tools framework is needed around Smart Objects to make plumbing a new web service as easy as making an instance of a class.

I also will be looking at prototyping Arduino sensors, super-structed gateways, user informing devices, and smartphone/tablet controls.