Event Models for RESTful APIs

Michael J Koster
May 5th, 2013

Adding event-driven processing to REST APIs is an important concept for the emerging Internet of Things. REST APIs provide for high level system interoperability and software usability, while event driven processing is needed for autonomic capability and efficiency.

An *event* is a significant change in the state of some element of a system. Events are asynchronous, that is they occur at arbitrary times relative to other operations in the system.

REST APIs enable web scale resource interaction and simplify application programming. The state of the system and values of observable properties can be easily discovered and operated on using GET and SET operations in URI addressable end points that have memory-like semantics. Metadata can easily be added to functional data to enable linking resources into composite horizontal applications.

Event driven systems process real time data efficiently. These systems are often based on asynchronous messages sent between end points that invoke software at the end points. Application programming is generally more difficult because of the need to interact using a stateful communication protocol. One example is the difference between programming for sockets and messages (asynchronous) vs. memory-like read/write semantics (synchronous).

The Internet of Things needs to be an event driven system for efficient real time response and processing, and needs REST APIs for resource discovery and programming efficiency, and to support rich metadata for interoperability in horizontally integrated applications.

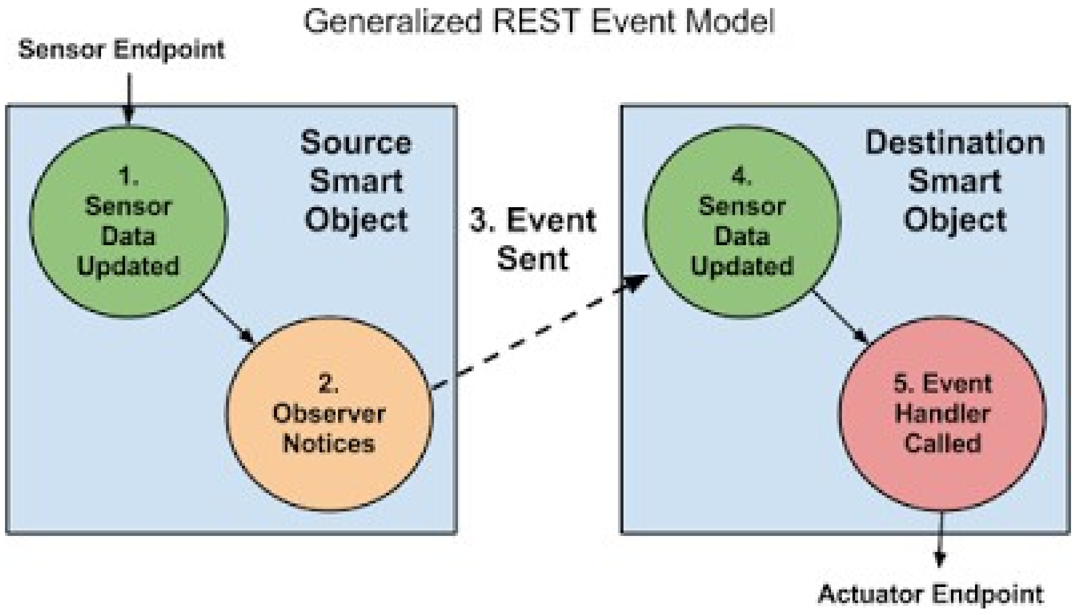
In the context of a REST API, an event occurs when a resource in the API is updated and/or it's value is changed, or changed significantly. Events can be any updates, or filtered updates that meet some criteria, for example temperature too high, temperature too low, or temperature changed by more than 2 degrees since the last update. Typically, Client-Server REST systems don't provide for asynchronous events, instead software would need to perform periodic GET operations to sample the state of a resource.

We propose to add event driven processing to REST by introducing a common set of patterns and abstractions that provide a well defined event model. The event model will enable asynchronous event processing and message oriented communications to be configured and controlled through the REST API.

Many of these patterns and abstractions are familiar and in use today in various forms. For example, CoAP has support for both synchronous and asynchronous REST operations, there are message brokers with REST endpoints (Qest, Xively), and there are hooks for instrumenting RESTful endpoints to add callbacks on updates (Real Time REST hooks). This will summarize the fundamental patterns and build a complete event model based on these patterns.

REST endpoints as event sources

An event occurs when some resource is updated or changes. To support events in a REST architecture, there needs to be some means to detect and generate events when resources are updated, a means to propagate the events and direct them to the proper destination, for example to update another REST endpoint, and a means of acting on the event, by storing a record of the event or invoking event handler software to respond to the event.



The diagram above shows an example event processing graph:

(1) sensor data are updated when the sensor endpoint makes an observation and sends an update to the Smart Object Observable Property, which is a REST resource.

(2) an *Observer* resource associated with the Observable Property informs the system to notice updates and forward them according to routing instructions contained in the Observer resource.

(3) The event is routed to the destination resource

(4) The Observable Property at the destination Smart Object is updated

(5) A handler is invoked based on an observer associated with the local resource. This updates a physical actuator endpoint.

HTTP provides for asynchronous operations from Client to Server using POST, PUT, or CREATE. By arranging to have a server endpoint at each resource location, as is the case in with the Smart Object API shown above, asynchronous updates can be propagated using standard HTTP protocol operations POST, PUT, and CREATE.

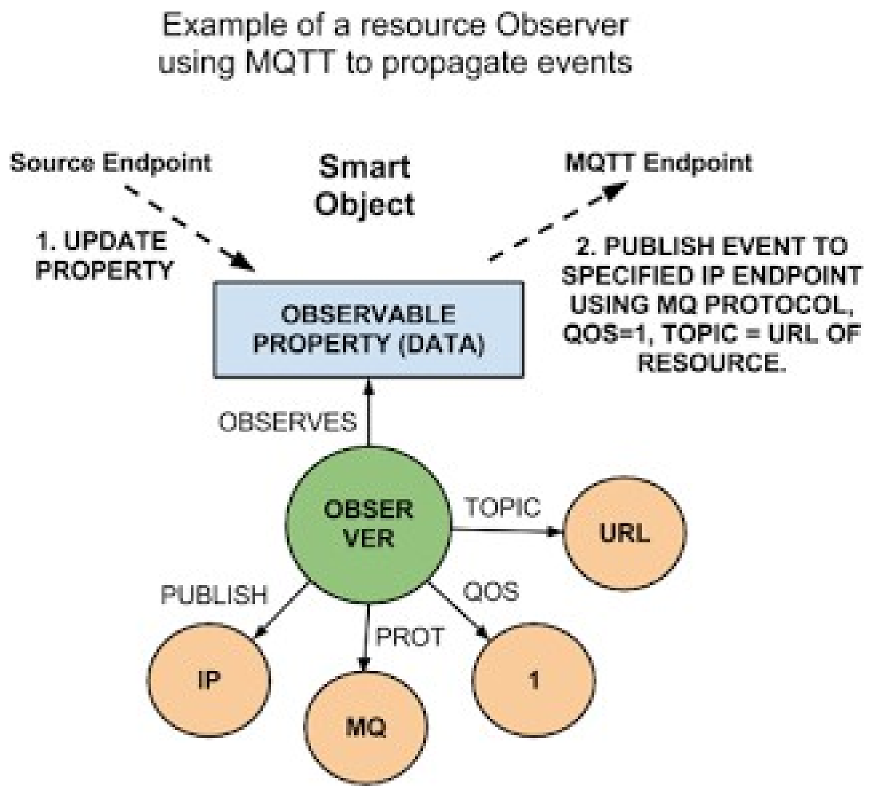
Other protocols like MQTT can also be used to send events between resources. The Observer resource can program the topic, broker, service, QOS, etc. in it's stored representation.

Using MQTT, the topic can be a representation of the URL and the data can be the representation of the data, e.g. JSON. This way, the URLs can be send between endpoints whilst tunneling through MQTT, preserving the semantics of REST updates and, more importantly, preserving the granular resource access control system based on capabilities that REST enables.

We will use MQTT as an example asynchronous protocol for this document, but we could also use HTTP POST, websockets, CoAP, XMPP, AMQP, Dbus, MODBUS, RS-232, WSNs, or any other asynchronous communication mechanism.

Observers and Subscribers in a general REST event model

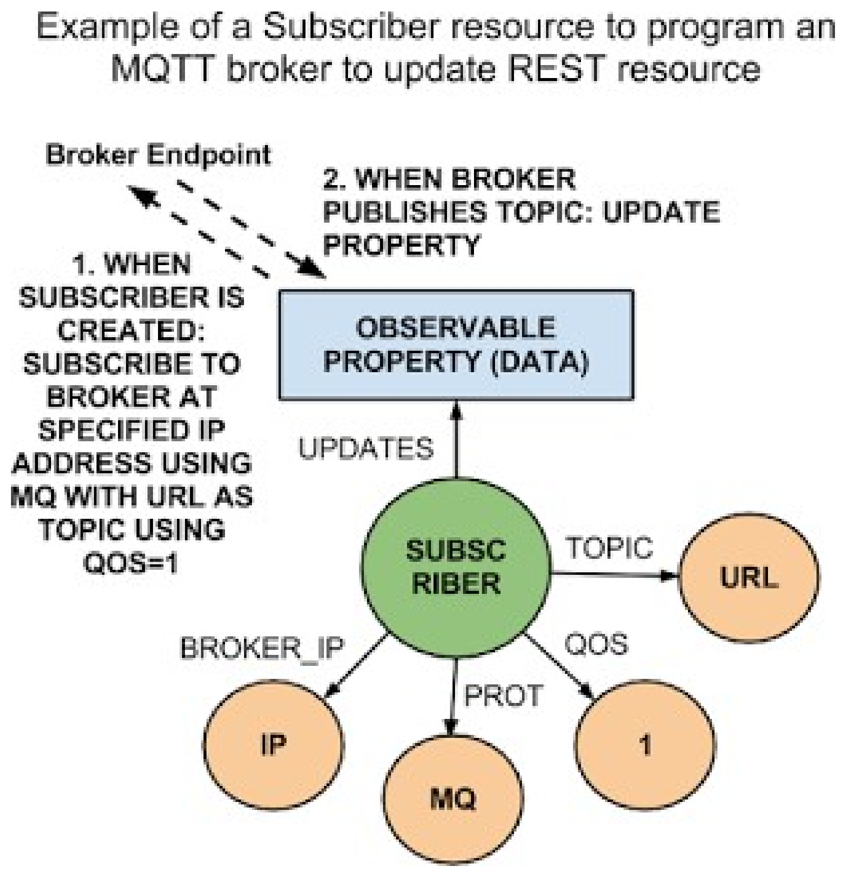
The ***Observer*** is a graph resource that is associated with a data resource. The Observer informs and programs the system to generate events on updates of the associated resource. When the observed resource is updated, the system performs the actions specified by the Observer's graph end points, either propagating a representation of the resource update to another endpoint or invoking a handler. Graph endpoints can be added to describe the action, for example to specify the QOS of an MQTT publish operation.


Example of a resource Observer using MQTT to propagate events
Source Endpoint
Smart Object
MQTT Endpoint
1. UPDATE PROPERTY
2. PUBLISH EVENT TO SPECIFIED IP ENDPOINT USING MQ PROTOCOL, QOS=1, TOPIC = URL OF RESOURCE.
OBSERVABLE PROPERTY (DATA)
OBSERVES
OBSER VER
TOPIC
URL
PUBLISH
QOS
PROT
IP
MQ
1

This example above shows an Observer associated with a REST resource (Observable Property) that programs the endpoint to publish updates to a specified MQTT broker endpoint whenever the Observable Property is updated. The protocol, IP address, QOS, and topic are all specified as local graph relationships that the endpoint uses to program it's behavior when the Observer is created.

Observers can use HTTP POST, websockets, XMPP, or any asynchronous event notification method available in the underlying platform.

The ***Subscriber*** is a graph resource associated with a destination resource endpoint which is to be updated. The Subscriber programs the destination endpoint to subscribe to a data source, such as a MQTT broker or a Smart Object Observable Property that uses HTTP POST. This creates a graph link from destination to source, and could be used to create an Observer resource at the source endpoint.


Example of a Subscriber resource to program an MQTT broker to update REST resource
Broker Endpoint
2. WHEN BROKER PUBLISHES TOPIC: UPDATE PROPERTY
1. WHEN SUBSCRIBER IS CREATED: SUBSCRIBE TO BROKER AT SPECIFIED IP ADDRESS USING MQ WITH URL AS TOPIC USING QOS=1
OBSERVABLE PROPERTY (DATA)
UPDATES
SUBSC RIBER
TOPIC
URL
BROKER_IP
QOS
PROT
IP
MQ
1

This example shows an MQTT subscriber that creates a subscription with a broker to publish updates to the associated REST resource (Observable Property). This creates a graph link back to the data source. This pattern can be used to control HTTP POST, CoAP extended GET, and other asynchronous notification mechanisms from the destination endpoint.

Note that Observers and Subscribers are ways of controlling asynchronous updates from either the source or the destination endpoints, or both. If both are used between 2 endpoints, there is a 2 way resource graph connection established.

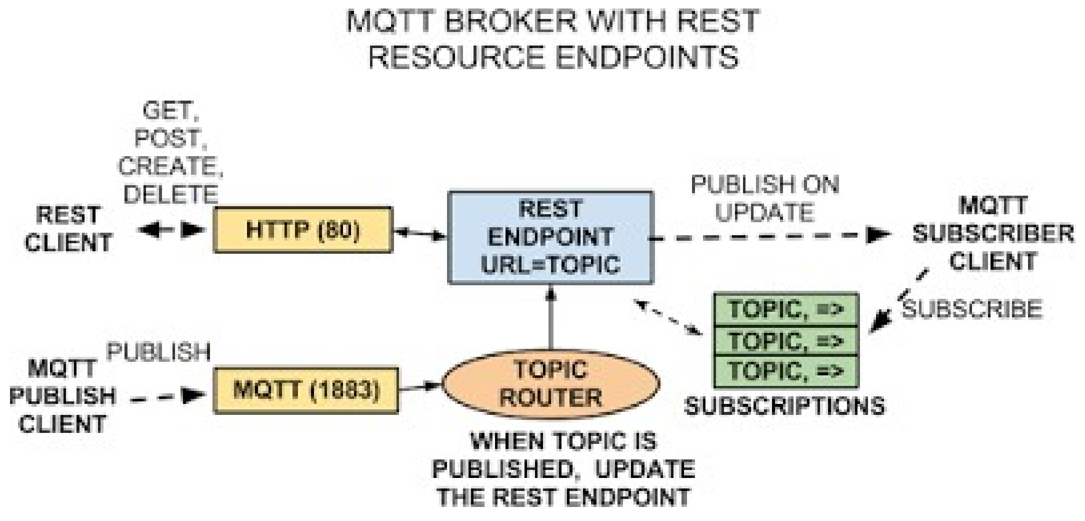
Message broker with REST endpoints

Another set of abstractions is to enable the configuration and of existing message systems such as MQTT using REST endpoints. There are two interfaces, a message interface and a REST interface, to shared stored resources.

There are two message operations supported, to enable clients or brokers to publish updates to REST endpoints through a message interface, and to publish updates of REST endpoints to brokers or clients subscribing through a message interface. There is also a subscribe operation using the message interface. Incoming messages update resources, and resource updates send outgoing messages.

The convention is to use URIs as topics that allow the system to construct URLs for the REST resource endpoints. The payload consists of the resource representation, e.g. JSON. This handles incoming and outgoing publish operations and acts as a message broker with REST access to topics and, if there is storage, topic history storage.

REST Message Broker allows message clients to publish to REST endpoints and subscribe to REST endpoints. An application can monitor an MQTT endpoint by observing the REST resource it publishes to, and an application can publish to an MQTT client by updating to the REST endpoint the MQTT endpoint is subscribed to. For example, sensors could connect with MQTT and publish updates for application software to read, and application software could update a REST endpoint, resulting in the system publishing the update to a sensor that is subscribed to the URL as topic.



The above example shows how an MQTT broker endpoint can be added to a REST server to enable REST clients and MQTT clients to share data by mapping URLs to topics. When either a REST client updates a resource an MQTT client publishes to a topic, the corresponding resource update is published to any subscribers.

A General Event Model for REST APIs and architectures

In summary, a complete event model is presented for adding event driven real time capability to RESTful architectures. The event model enables RESTful abstractions for event sources and destinations, message broker endpoints, client and server endpoints, and asynchronous software event handlers.

Through the *Observer*, *Subscriber*, and *REST Message Broker* abstractions, and associated graph resource definitions, a general facility is created that can abstract the functions of common event and message passing systems to provide high level REST API control of the low level protocols, for example QOS control for MQTT endpoints. These high level models provide a consistent API for software interoperability, while allowing diverse protocols to be used in the underlying platform.