Introduction to the REST API – Part I

There is no doubt in my mind that you are aware of the REST API, as for the past several years it has gained tremendous momentum. VMware has spent a great deal of engineering time in making all of VMware's products across the board support and expose a REST API and interface. This is a crucial and critical element in Automation of the SDDC.

In this post and subsequent posts we are going to take a look at the REST API and in particular we are going to cover the following items

- **What is REST All About**
- **REST Elements (Resources)**
- **REST Request & Response Methods**
- **Important Headers and Requests**
- **VMware NSX for vSphere Networking and Security REST API**
- **Authorization and Ports**
- **API Consumption Options**
- **Tools Required**
- **REST Clients**

Some of the key objectives for you in this post are to Identify the value of programmatically automating and controlling the NSX for vSphere platform through the REST API, how to become familiar (or more familiar if you already have experience) with the Basic REST API consumption methods, be able to compare and contrast the various API consumption methods (NSX for vSphere REST API, vCloud API…), understand the workflow process for consuming the NSX for vSphere REST API, and perform the basic REST commands to make configuration changes against the NSX for vSphere Global Configuration, Edge Devices and Distributed Firewall.

Sound good? OK, lets get started then!

What is REST All About?

Firstly, and foremost, **REST** itself does not define where or how the state of resources should be stored, only how it can be retrieved (via GET – we will cover this later) or provided (via PUT and POST). It is also possible to have a read-only resource (only supporting GET). Otherwise, the resource state could be provided by any means such as a filesystem, a combination of other resources, physical sensors, and so on. I always like to think of REST in the following terms :

- Representational State Transfer (REST) is an approach (design pattern) to designing resource oriented web services.
- REST is NOT a Standard.
- REST is NOT a Protocol.
- REST is an Architectural Style for Networked Applications.
- REST defines a set of simple *principles* which are loosely followed by most API implementations.
- Resource (the source of specific information)
- Global Permanent Identifier (every resource is uniquely identified – think HTTP URI)
- Standard Interface (used to exchange the representation of resources – think the HTTP protocol)
- Set of Constraints

Let us take a look at the constraints, which are very important to understand and to which Fielding grouped into six categories :

- – Client-server
- – Uniform interface
- – Layered system
- – Cache
- – Stateless
- – Code-on-demand

Each constraint category is summarized in the following subsections :

Client–Server

The separation of concerns is the core theme of the Web's client-server constraints. The Web is a client-server based system, in which clients and servers have distinct parts to play. They may be implemented and deployed independently, using any language or technology, so long as they conform to the Web's *uniform interface*.

Uniform Interface

The interactions between the Web's components—meaning its clients, servers, and network-based intermediaries—depend on the uniformity of their interfaces. If any of the components stray from the established standards, then the Web's communication system breaks down.

Web components interoperate consistently within the uniform interface's four constraints, which Fielding identified as :

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)

The four interface constraints are summarized in the following subsections.

Identification of resources

Each distinct Web-based concept is known as a *resource* and may be addressed by a unique identifier, such as a URI. For example, a particular home page URI, like *http://www.vmware.com*, uniquely identifies the concept of a specific website's root resource.

Manipulation of resources through representations

Clients manipulate representations of resources. The same exact resource can be represented to different clients in different ways. For example, a document might be represented as HTML to a web browser, and as JSON to an automated program. The key idea here is that the representation is a way to interact with the resource but it is not the resource itself. This conceptual distinction allows the resource to be represented in different ways and formats without ever changing its identifier.

Self-descriptive messages

A resource's *desired* state can be represented within a client's request message. A resource's *current* state may be represented within the response message that comes back from a server. As an example, a wiki page editor client may use a request message to transfer a representation that *suggests* a page update (new state) for a server-managed web page (resource). It is up to the server to accept or deny the client's request.

The self-descriptive messages may include *metadata* to convey additional details regarding the resource state, the representation format and size, and the message itself. An HTTP message provides *headers* to organize the various types of metadata into uniform fields.

Hypermedia as the engine of application state (HATEOAS)

A resource's state representation includes links to related resources. Links are the threads that weave the Web together by allowing users to traverse information and applications in a meaningful and directed manner. The presence, or absence, of a link on a page is an important part of the resource's current state.

Layered System

The layered system constraints enable network-based intermediaries such as proxies and gateways to be *transparently* deployed between a client and server using the Web's uniform interface. Generally speaking, a network-based intermediary will intercept client-server communication for a specific purpose. Network-based intermediaries are commonly used for enforcement of security, response caching, and load balancing.

Cache

Caching is one of web architecture's most important constraints. The cache constraints instruct a web server to declare the *cacheability* of each response's data. Caching response data can help to reduce client-perceived latency, increase the overall availability and reliability of an application, and control a web server's load. In a word, caching reduces the overall *cost* of the Web.

A cache may exist anywhere along the network path between the client and server. They can be in an organization's web server network, within specialized content delivery networks (CDNs), or inside a client itself.

Stateless

The stateless constraint dictates that a web server is not required to memorize the state of its client applications. As a result, each client must include all of the contextual information that it considers relevant in each interaction with the web server. Web servers ask clients to manage the complexity of communicating their application state so that the web server can service a much larger number of clients. This trade-off is a key contributor to the scalability of the Web's architectural style.

Code-On-Demand

The Web makes heavy use of code-on-demand, a constraint which enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients. Code-on-demand tends to establish a technology coupling between web servers and their clients, since the client must be able to understand and execute the code that it downloads on-demand from the server. For this reason, code-on-demand is the only constraint of the Web's architectural style that is considered optional. Web browser-hosted technologies like Java applets, JavaScript, and Flash exemplify the code-on-demand constraint.

After reading the above you may have come to understand that :

- The World Wide Web is built on REST (and is meant to be consumed by humans)
- RESTful Web Services APIs are built on REST (and are meant to be consumed by machines)
- Representational State Transfer (REST) (developed by Roy Thomas Fielding: http://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- REST Constraints – Are Grouped as in the aove into Six Categories (Client-Server, Uniform Interface, Layered System, Cache, Stateless, Code-On-Demand)

Let us now turn our attention and take a look at what makes up the REST Elements.

What are the REST Elements?

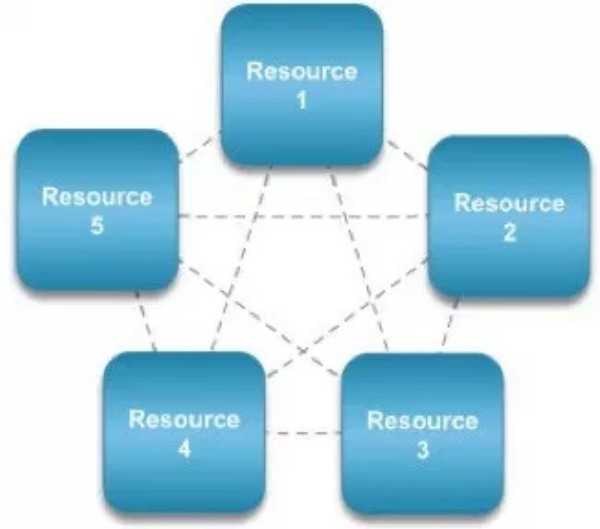
REST elements are comprised of **Resources** (they are the building blocks of the web), and **Resources** are linked together by embedded hyperlinks in HTML documents or URI references.

Let's start our exploration with the most important element : *resources*. They are the building blocks of the web as I stat above. A resource can be anything of interest that an application wants to expose on the network for other applications to use. It can be the balance of a bank account, the current temperature in the UK, a Shakespearean text, a video of a play, a collection of pictures, a blog post, and so on.

One important characteristic of those resources is that they're linked together by embedded hyperlinks in HTML documents or URI references in feeds pointing to recent blog posts. In exactly the way it was envisioned and named by its creator Tim Berners-Lee, the web is a distributed set of resources that can be navigated and discovered dynamically.

What's amazing is that all those resources—retrieved, updated, or deleted by users, humans, or robots—form a complex system that's constantly evolving and growing.

Figure **– The web as a graph of potentially hyperlinked resources**



Illustrates this and shows the potential relationships between resources via hyperlinks. In practice, only a few relationships will be used and may sometimes even be broken. We all know the 404 "Not found" error pages that are annoying but that also prove the liveliness of the web.

To allow users to retrieve them from anywhere in the world, each resource is given a unique name called a Uniform Resource Identifier or URI (for example, http://www.example.uk/weather), enabling identification and remote access on the internet.

Resources can expose the state via representations containing both metadata (such as size, media type, or character set) and content (binary image or text document). As shown in this figure below a resource can expose its state via representations containing both metadata (such as size, media type, or character set) and content (binary image or text document). The representation of a confirmation of purchase on eBay could be an HTML document; for a wedding picture it could be a JPEG binary stream; for a contact in an address book web service it could be an XML fragment; and so on.

Figure **– State Representation**

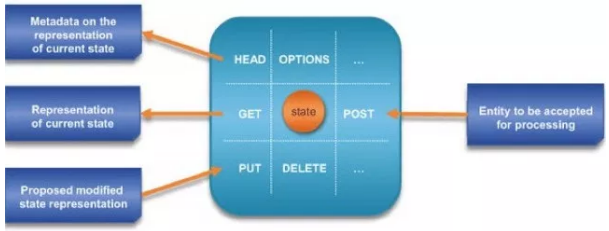
Now let's look at what I call the Anatomy of a Resource!

As you can see the figure below, the circle delimits the resource from its external environment, which interacts with it.

The *resource* is composed of some state depicted at the center, managed in any way that makes sense (like in a database, in a file, or computed dynamically) and of standard methods (like GET, PUT, DELETE, or POST explained right after) that together define its *uniform interface*.

To give you an idea of the importance of *resources*, you can compare them to objects in the *object-oriented* paradigm. They are the backbone of your web applications, exposing both state and behaviour. Compared to the object-oriented paradigm, the standard resource methods (usually HTTP methods) are similar to object methods, with a key difference: they're limited in number and have a behaviour that's predefined by a protocol such as HTTP. Because applications use predefined methods, instead of defining their own, to express interactions over the network, the network infrastructure can understand to a certain degree the semantics of the interactions it's carrying on. From this understanding comes the ability to provide a number of key services such as caching and automatic message reemission, things an infrastructure can't do blindly without understanding some of the application-level semantics of the interactions.

Figure **– The Anatomy of a Resource**



Still confused or lost?

REST stands for "Representational State Transfer" which just serves to make things less clear, I think!

The key concept is of a stateless client-server architecture where clients send requests to servers which in return send back responses. Pretty simple, eh? Rather than the client and the server maintaining state, the state is transferred in the requests and responses. This is what "representational" means: there's some representation of the state exchanged between client and server. There are a number of things that REST architectures impose. If you adhere to these constraints (as explained above), your architecture is considered RESTful. It just means that your architecture obeys the REST constraints.

REST is not new, but it has become in-vogue. Basically the entire web is built on REST concepts. Every time you click a button on a web page in a web browser (client) you're sending a request to a web server, which in turn sends back a response. This does not mean that all web services are REST-based, of course, just that they share some of the characteristics. A good example of a RESTful application is Google Mail. When you log into Google Mail, it sends a request to get the contents of your inbox (actually the latest n messages). The Google Mail server sends back a ***representation*** (the R in REST) of your inbox, which the client displays. You can then click on a message to read it, which sends a request to the server for the contents of the message. Actually, it's a little more complicated than this, since Google Mail can be doing things behind the scenes while you are reading messages. The point is that the Google Mail server has no idea which set of messages you're reading or which message is currently displayed. It keeps no state (i.e., it's "stateless".)

If it helps, you can consider the client being "at rest" when it's not actively engaged in communication with the server.

Also note that RESTful architectures do nor have to be based on HTTP protocol. You could implement them over any protocol you like. It's just that typically, HTTP is what RESTful implementations are based on.

There's a nice explanation in wikipedia:

http://en.wikipedia.org/wiki/Representational_State_Transfer

Note carefully the list of 6 Constraints.

You might find the following interesting reading to:

A Brief Introduction to REST

http://www.infoq.com/articles/rest-introduction