Document Identifier: DSP2045

Date: 2015-08-04

Version: 1.0.0

Redfish Frequently Asked Questions (FAQ)

Supersedes: None

Document Class: Informative

Document Status: Published

Document Language: en-US

CONTENTS

1. Why a new Interface?
2. Why REST, HTTPS & JSON?
3. Why OData?
4. Why Two Schema formats?
    4.1. Why OData Schema?
    4.2. Why JSON Schema?
5. Why Hypermedia API?
    5.1. Why force the use of "/redfish/v1/" as the service root?
6. Why PATCH and PUT?
7. Why PUSH Eventing only?
8. Why separation of data model from protocol?
9. What's a message registry and why did you select it?
    9.1. Why is the Published Message Registry different than the one in the Mockup?
10. What's a RelatedItem?
    10.1. Why do these values have two different formats and how does an implementer know which is which?
11. Why not CIM & MOF?
12. Why not Ironic?
13. Why is the ETag algorithm left to the implementation?
14. Why does Redfish need to use ETags at all?
15. Why isn't Localization Supported?
16. Why are AES ciphers "should"?
17. Why are PSK GCM AES Ciphers specifically mentioned?
18. ETag modification data leaks private information about resources
19. AccountService roles allow an escalation of privileges
20. Logging of events

Introduction

This document is intended to explain the rationale and reasoning behind some of the architectural decisions in the Redfish API Architecture

1. Why a new Interface?

The market is changing and the current solutions are not adequate to address the multiple dimensions that change is happening on. Below is a bulleted list describing some of the shifts in the market:

- Market shifting to scale-out solutions
  - Massive quantities of simple servers, reliability achieved through software
  - Usage model different than Enterprise (servers as "cattle", not "pets")
  - Customer demand for standards-based, multi-vendor deployments
- Functionality lags in scale-out
  - IPMI feature set is limited to low "common denominator" (e.g. Power On/Off/Reboot, temperature value, text console)
  - Customers increasingly developing their own tools for tight integration
  - Increasing fragmentation of IPMI specification as OEM extensions proliferate
- Customers exhausting basic IPMI functionality
  - Security and encryption support no longer meets customer requirements
  - Increasing occurrence of proprietary extensions fragment the specification
  - SMASH/CIM not an option for these customers (see further section)
- New system architectures cannot be modeled in standard IPMI structure
  - Bit-wise encodings cannot represent complex architectural relationships
  - Aggregation points, multi-node systems
- Customers are asking for a modern interface
  - Expect APIs to use Cloud / Web protocols, structures and security models

2. Why REST, HTTPS & JSON?

The market is quickly evolving to adopt the RESTful/JSON style of interface for a variety of reasons. This bulleted list outlines some of those reasons:

- REST: Rapidly replacing SOAP
  - Ecosystem adopting it (OpenStack, etc.)
  - Much quicker to learn than SOAP/WS-Man
- HTTPS: The Web protocol
  - Well-understood by admins
  - Known security model
  - Known network configuration
  - Being taught in High Schools and Jr Colleges
- JSON: Modern data format
  - Human-readable
  - Simpler than XML
  - Modern language support
  - Rapid growth
- Combination the de-facto choice for public APIs
  - Most Popular Protocols used by Web APIs
  - Most Popular Coding Languages 2014
  - Data Formats used in New APIs

3. Why OData?

Following common OData conventions for describing schema, url conventions, and naming and structure of common properties in a JSON payload, not only encapsulate best practices for RESTful APIs but further enables Redfish services to be consumed by a growing ecosystem of generic OData client libraries, applications, and tools.

So with Redfish, we get the best of both worlds - a direct API accessible by JSON developers as well as the growing number of OData clients.

4. Why Two Schema formats?

We wanted to support both a direct API accessible by JSON developers as well as the growing number of OData clients.

4.1 Why OData Schema?

We are trying to adopt as many existing standards as possible. OData is a ratified standard, on its way to ISO standardization. OData defines a rich schema definition language that describes things in terms of entities (things) and relationships - the way we think about things in the real world. OData's schema language is inherently extensible through annotations. Those extensions can be shared uniquely identified through URIs and shared across services and industries with common semantics as "vocabularies". Schema can be developed very rapidly due to tools and human readability.

4.2 Why JSON Schema?

We are trying to adopt as many existing standards as possible. Since JSON schema is being adopted by the IETF (currently in draft-4) and is already in JSON (thus can be transferred by our protocol), it was one of the few candidates. It is also extensible, so we can add any qualifiers not represented in the base schema. Additionally, because it is JSON, it can be utilized by the existing tool chain. It also helps us develop the schema very rapidly due to tools and human readability. And there is not a JSON version of CSDL (OData schema) available yet.

5. Why Hypermedia API?

A single API style that will match the myriad of computing platforms is required. The industry cannot support multiple interface or programming styles. A single interface that spans the market from standalone servers to hyperscale as well as partionable and even virtual systems is needed by customers. A fixed URI API methodology will not suffice to show the various containment relationships between the various forms of sheet metal, the servers within them and the managers associated with them. These means a 1-to-many or many-to-many cardinality of associations is needed.

Additionally, the API must be easy for simple systems and flexible for hyperscale. The use of hrefs to represent associations and collections for similar resources has been proven in hypermedia APIs.

5.1 Why force the use of "/redfish/v1/" as the service root?

It was chosen to represent the version of the protocol. Resources themselves have a separate versioning (the use of the Type property). Protocols have a different lifecycle than the data model they transport so a versioning mechanism is necessary.

The two choices were fixed URI or an X-Header. The RESTful world has adopted the fixed URI methodology and is veering away from non-standard headers (like X-anything). Additionally, headers in general and X headers in specific are difficult to use (though not impossible) in some of the tool sets.

6. Why PATCH and PUT?

PUT has replacement semantics -- properties not specified in a put are reset to default values (generally null). For example, if a resource allows a Setting Data (resource of settings to be applied in the future), PUT provides a way to clear existing properties. But what about when you only want to change a single property? Do you need to send all the properties? What if there are properties you don't know about?

PATCH solves this. While PUT always does a complete replacement, PATCH always does a partial replacement. Deterministic behaviour for the service is achieved without complex logic.

Because it is considered safer (because it doesn't overwrite unintended properties), PATCH is the preferred method for updates. PATCH is gaining wide adoption in the industry, and is already supported by Open Stack and many other APIs and in available in many off the shell web servers.

7. Why PUSH Eventing only?

We need some kind of eventing mechanism to meet comp with SNMP, IPMI and other protocols.

The two methods are "push" and "pull". Push sends events to a destination and pulls create a queue where clients can retrieve events. Pull eventing is resource intensive in a light-weight, small footprint BMC. This is why they already use push style eventing since the data does not need to be retained in precious BMC processing memory with push style eventing.

Note that in future versions of the spec, other eventing methods may be supported.

8. Why separation of data model from protocol?

The data model is expected to change, including extensions added to it, over time. Therefore, churn is expected. However, the protocol portion of the specification should remain relatively stable. So, instead of churning the entire specification, this separation attempts to isolate unnecessary versioning of unaffected specification sections.

Strict forward compatibility rules are in the specification, similar to other data models. The one constant is that the type annotation contains unique, dereferenceable URI to the definition for the resource so, regardless of resource type, the schema can always be obtained.

9. What's a message registry and why did you select it?

Message registries are not a new concept. They allow the Management Processor to retain a short identifier which results in a look up for the space intensive message contained a registry. Mechanisms are in place for parameter substitution on a per-message basis.

Message registries are not only light weight, but make it easy to support internationalization since the management processor does not need to contain the translations. Instead, the registry itself can be translated and the client can then display any translations needed.

One of the things you will notice is that Log Entries, Event Messages, Extended Error Information and Message Registries are all of similar format. We are attempting to optimize both the storage in the service and the usage by Clients in keeping all of this information in as similar a format as possible.

9.1 Why is the Published Message Registry different than the one in the Mockup?

You'll notice that @odata.context and @odata.id are missing from the published message registry. The reason is that this isn't retrieved via an implementation. The message registry in the mockup, however, is returned as if from a genuine implementation. This it has the URI that it came from as well as the path to metadata within the implementation.

10. What's a RelatedItem?

We needed a way to tie together things like sensors with the things they monitor. We also needed a way to show member sets, such as in Redundancy. URLs are the preferred referencing mechanism, but we needed to reference the individual property within a resource and not a whole resource. So we have a RelatedItem, the format of which is possible to be either a JSON Pointer or an OData format.

10.1 Why do these values have two different formats and how does an implementer know which is which?

If the RelatedItem is a JSON Pointer, there will be a '#' character in it. The part before the # will be the OdataID (href) of the resource and the part after will be the property path within that resource. Most broswer/library code acommodates JSON Pointer, so a client should be able to do a GET on the RelatedItem and get just the sub-resource.

If the RelatedItem is an OData format item, or if it is a whole resource, it will not have a '#' character in it. In this case, a client should be able to do a GET on the RelatedItem and get just the sub-resource since that is allowable for "Referencable" properties in OData.

11. Why not CIM & MOF?

We took a different approach for Redfish - we started with the toolset and interface style customers are adopting and leveraged that. We learned what they use instead of forcing them to learn our way and then hope vendors develop tools to go with it.

CIM is very normalized, even with View classes. There are so many classes and they are not purpose built for the customers wanting this technology.

The MOF files are not in JSON. Even if we translated them to JSON, we would need to prune them and turn all of the associations into hrefs in each class. Then we would have to create view classes. We would also have to rationalize the values which do not reflect true server/hyperscale values (look at the state and status fields alone, or look at how power is done) (or networking - 17 resources to show what Redfish can do in a single object within a resource).

Looking at a 4 processor System with 4DIMMs, gathering just the CPU/DIMM information would take over 90 IOs. A view class approach would take from 6 to 14 IOs. Redfish can do all of this and more in just 1 IO.

Next, consider the tool set - HTTP & JSON have a rich set of tools and programming languages with existing support. MOF/CIM is limited and vendor specific when it comes to tools. Many of the tools hoped for never materialized.

Every BMC today has a web server and most of them surface a web page that has a JavaScript driven GUI. So all of the components for an implementation are already in the firmware footprint. CIM/WBEM/MOF would require additional footprint.

The amount of documentation that it takes to understand CIM and the protocols takes reams and reams of paper, resulting in a stack well over a foot high. Our goal is to have someone read our spec in the morning, play with the data model an hour later and be coding by the afternoon - and that person should not be required to have a college education, understand cardinality, etc.

Lastly is customer demand. The customers asking for a RESTful API see CIM & MOF as "old", "Windows centric", "storage centric", "OS centric", "too hard" and other negative feedback. They are asking for something simple that meets their "10 commandlets".

12. Why not Ironic?

Ironic is the hardware management API for OpenStack. It is OpenStack specific and doesn't have the functionality of a full BMC - only what OpenStack needs. It is means as a translation layer between OpenStack and IPMI, SNMP and other protocols. It does not have all of the features we need and is very much tied to OpenStack. The market for a RESTful API is greater than just OpenStack. That being said, we expect a Redfish shim for Ironic to be developed.

13. Why is the ETag algorithm left to the implementation?

Considering the different ETag models (Weak/Strong) mentioned in the spec it seems that leaving the model decision up to the server implementation will leave clients not knowing what has been implemented and therefore not knowing how to interpret the ETag.

In practice it doesn't matter since the value of the ETag will only be as good as the server is capable of providing for. The client should treat ETags as opaque anyway and use them to determine if data has changed. This is the adopted practice in all web servers and is what RFC2616 allows.

14. Why does Redfish need to use ETags at all?

It allows the use of the If-Match header for replacement of resources. It also allows the clients to determine if anything has changed before doing a large fetch (a Server collection in Moonshot is hundreds of servers).

It also allows the server the option of providing something more robust than a simple modified time.

15. Why isn't Localization Supported?

For version 1.0, priority was given to features, functions and scalability. But some thought was given to Client-side localization rather than service side localization.

Schema-supplied display strings may be localized as necessary by clients and there are also JSON Schema file values for localized Schema, but any Schema file may only contain one language. Alternate language schemas may be published and available to Redfish clients, but need not be provided via the Redfish implementation.

Property names defined within a Redfish schema are never localized. User-supplied string-valued property values such as an asset tag may be localized. Localizable string valued properties could be annotated with the OData IsLanguageDependent annotation term.

Registries, likewise, may also be localized and lend themselves to translation. The MessageID can be used to look up values in a translated registry as easily as in a non-localized version.

While thought was given to other localization mechanisms, it was determined to consider localization fully in later versions.

Security

16. Why are AES ciphers "should"?

These ciphers provide a requisite level of security, efficient hardware implementations exist and are widely available to support even low cost implementations. They are what is recommended now. But we know security requirements change over time, so they are "shoulds" not "shalls". And they may not be right for every situation but they are what we believe implementers should support at the time of the initial release of the 1.0 Redfish specification.

17. Why are PSK GCM AES Ciphers specifically mentioned?

While TLS is an excellent solution for security, it depends on a full PKI implementation for basic identity verification. PSK GCM ciphers provide both identity verification and encryption, allowing for a bi-directionally trusted connection during initial setup or factory default configurations on the basis of a known initial password. This information is also relative to the initial release of the 1.0 Redfish specification.

18. ETag modification data leaks private information about resources

It is possible for an otherwise invisible change to an object - e.g. changing a password - to cause an ETag to change, potentially providing useful information. ETag implementations are left to the vendor, many current ETag implementations show this metadata and accept the risk.

19. AccountService roles allow an escalation of privileges

It is possible for a user with the System Administrator role to create user accounts which are assigned to roles which the creating user does not themselves possess. This escalation of privileges is well understood and accepted by users, and avoids complications caused by introducing special privilege states or additional roles to handle privilege propagation corner cases.

20. Logging of events

Redfish requires logging of some events, but does not require the implementation of a local log, though one is implied due to the fact that it is represented in both metadata/schema and the mockup. This allows implementations of logging that are record-only logs, or logs which are securely stored outside the service using services like syslog.