Document Identifier: DSP2044

Date: 2016-06-15

Version: 1.0.2

Redfish White Paper

Document Class: Informative

Document Status: Published

Document Language: en-US

CONTENTS

1. Why a new interface? 5
2. Why REST, JSON and OData? 6
3. Why a hypermedia API? 7
4. Accessing an Implementation 8
5. Root 8
6. Operations 10
7. Versioning 11
8. References 11
9. Main Objects 11
10. Collections 12
11. Common Properties 12
12. Common Annotations 13
13. Actions 14
14. Schema 14
15. Sessions 15
16. Redundancy 16
17. RelatedItem 16
18. Services 17
19. Registries 17
20. Headers 17
21. ETags 17
    21.1. Handling Client Race Conditions 18
22. Updating Resources 18
    22.1. PUT vs PATCH 18
    22.2. Current Configurations vs Settings 19
    22.3. Determining Properties that can be updated. 19
23. Extended Error Response 20
24. Eventing 20
25. Creating User Accounts and other resources 20
26. Message Alignment 21
27. Oem 21
28. Idempotency 21
    28.1. Idempotent Modify: GET/PUT/PATCH 22
    28.2. Create, Use, Delete: POST/GET/DELETE (non-idempotent) 22
    28.3. Do Action: POST with "Action" property (non-idempotent) 22
29. Inheritance & Polymorphism 22
30. Inheritance by Copy 22
31. Polymorphism by Union 23
32. Finding Temperature Sensors for a System 25
33. Chassis within Chassis 25

What is the Redfish API?

Redfish is a management standard using a data model representation inside of a hypermedia RESTful interface. The model is expressed in terms of standard, machine-readable Schema, with the payload of the messages being expressed in JSON. The protocol itself leverages OData v4. Since it is a hypermedia API, it is capable of representing a variety of implementations via a consistent interface. It has mechanisms for managing data center resources, handling events, long lived tasks and discovery as well.

1. Why a new interface?

A variety of influences have resulted in the need for a new standard management interface.

First, the market is shifting from traditional data center environments to scale-out solutions. Scale-out solutions & hyper-scale have adopted the use of massive quantities of simple servers brought together to perform a set of tasks in a distributed fashion (as opposed to centrally located). Reliability in these environments is achieved through software that is either proprietary or open source. As a result, the usage model different than traditional Enterprise environments. One analogy is that servers are treated as “cattle”, not “pets”. This set of customers demand a standards-based interface that is consistent in heterogeneous, multi-vendor environments.

Functionality and homogeneous interfaces are lacking in scale-out management. For instance, the IPMI feature use is limited to a “least common denominator” set of commands (e.g. Power On/Off/Reboot, temperature value, text console). As a result, these customers, while desiring out of band functionality, have been forced to use a reduced set of functionality because vendor extensions are not common across all platforms. This set of new customers are increasingly developing their own tools for tight integration, sometimes relying on in-band software for management since they are able to develop a common set of manageability in that environment. Increasing fragmentation of platform management specifications as OEM extensions proliferate result in features that do not satisfy scale-out customer needs since they are fragmented. And by referencing specific security and encryption requirements, existing management solutions no longer meets customer security requirements.

Other standards, such as SMASH, have not met the ubiquity that was hoped for. This is due to it's complexity. The CLP ended up being implemented in most hardware but not with a consistent output format thus parsing resulting data was implementation dependent. WS Management was only implemented in a limited number of out of band environments. It is a complex, layered protocol that works best in homogeneous environments and thus never fulfilled the heterogeneity requirements. Additionally, the complexity of the combination of understanding the protocol, generic operations, schema and the profiles themselves ended in a solution that took years to develop, years to change and add new functionality. It takes months for customers to understand as well as significant resource commitments and expertise in order to become proficient in its use. It also requires a significant number of operations to perform even the simplest tasks. Thus, while it can represent scalable systems the interface itself has an IO pattern that is not scalable.

The hardware landscape is rapidly changing. Systems that either put multiple systems on a traditional "blade" or aggregate a single system out of multiple blades are becoming more prevalent. These need to be managed by the same software that is managing traditional enterprise environments. But these new systems cannot be adequately expressed in bit-wise protocols. They cannot represent complex architectural relationships between components in modern systems. Additionally, system aggregation points, such as chassis or enclosure managers are incapable of using these protocols to perform their function of expressing these complexities in the same method.

Finally, customers are asking for a modern interface. They expect APIs to use the protocols, structures and security models that are common in emerging cloud interfaces. These interfaces have the advantage that customers have invested in the tools to accelerate development. Specifically, customers (without any prompting from vendors) are asking for RESTful protocols that express data in JSON formats.

2. Why REST, JSON and OData?

REST is rapidly replacing SOAP as the protocol du jour. The entire cloud ecosystem is adopting it and thus the web API community is as well. It is much quicker to learn that SOAP. It has the simplicity of being a data pattern (as it is not strictly a protocol) mapped to HTTP operations directly.

- Most Popular Protocols used by Web APIs

REST semantics are easily mapped to HTTP/HTTPS. Thus, it can be easily understood by most system administrators, has a well known security model and network configuration settings for its support are understood. It is being taught in High Schools and Jr Colleges. As a result, the education requirements for understanding its use a greatly reduced. Open source implementations of web servers abound and programming language support is ubiquitous.

JSON is rapidly becoming the modern data format. It is inherently human readable, is more concise than XML, has a plethora of modern language support and is the most rapidly growing data format for web service APIs. It has one additional advantage for embedded manageability environments. Most BMCs already support a web server and managing a server through a browser is common. This is usually done via a Java script driven interface. By choosing JSON, the web browser can use the data from an Redfish service directly in the web browser, ensuring that the data through the browser and the programmatic interface is uniform in semantics and value.

- Most Popular Coding Languages 2014
- Data Formats used in New APIs

But following RESTful practices and formatting results as JSON alone are not enough. There are nearly as many RESTful interfaces as there are applications, and they all differ in the resources they expose, the headers and query options available, and how results are represented. Similarly, while JSON provides an easy-to read representation, semantics of common properties such as id, type, links, etc. are imposed through naming conventions that vary from service to service.

OData defines a set of common RESTful conventions which provides for interoperability between APIs.

Adopting 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 which can be used in traditional and scalable environments but further enables Redfish services to be consumed by a growing ecosystem of generic client libraries, applications, and tools.

3. Why a 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 stand alone servers to hyper-scale as well as partionable and even virtual systems is needed by customers. A fixed URI API 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 hyper-scale. The use of URLs to represent associations and collections for similar resources has been proven in hypermedia APIs.

Getting Started

One easy way to understand the Redfish API is to see an implementation in use, so these next sections will walk through an implementation and explain the various architectural structures and concepts by walking through an implementation.

You will need two things to start - a browser and an implementation to access with the browser. Alternatively, you can walk through the files using a file browser or manager.

Because it is based on REST and JSON, all you need is a browser to be able to start viewing an Redfish implementation. To make the data easier to read, it is suggested that you have a browser capable of viewing JSON format. This can be done with a plug-in for most browsers. For a real "Redfish" experience, you can download a RESTful plug in for your browser such as the Advanced REST Client for Chrome. This allows you to set headers, see HTTP codes and other items that browsers normally hide. You will also get an advantage when accessing a "real" Redfish implementation such as the emulator or a vendor implementation.

4. Accessing an Implementation

You can view the markup either directly or through a web server. All you need is access to an implementation. There are a few ways to do that:

- Access the public mock-up. There is a web server running JSON at redfish.dmtf.com. You can point your browser to it and access the data. This will let you explore the mockup and the schema.
- Copy the data to your web server. You can access the Redfish Mockup and copy all of the files under the Mock-up directory to your local hard drive and place them under the /redfish/v1 directory that your server uses for serving up HTML pages. And example of this would be downloading nginx, setting it up to return JSON format and then loading the mock-up files in the html directory.

5. Root

Redfish is a hypermedia API. That means that you get to all the resources through URLs returned from other resources. But there is one well-known URL so that every implementation has a common starting point. That URI is "/redfish/v1" for version 1 of the Redfish interface.

URLs have a schema (the HTTP:// part), a node (such as www.dmtf.org or an IP address like 127.0.0.1) and a resource part. You put these together in the URL of your browser. So, if you are using the nginx server on your own machine, you should be able to put "HTTP://127.0.0.1/redfish/v1/" in your browser to access the Redfish root.

Do a GET on the root of the service from the URL you have constructed above.

Basic Concepts

Every URL represents a resource. This could be a service, a collection, an entity or some other construct. But in RESTful terms, these URIs point to resources and clients interact with resources. So when you see the term resource, you can think of it as what you get back when you access a URI.

The resource format is defined by a Schema. Each resource has a specific format that is specified in the Redfish Schema that the client can use to determine the semantics about the resource (though we try to make things as intuitive as possible). The Redfish Schema is defined in two formats: an OData-Schema format and a JSON Schema format. It is defined in the OData Schema format (CSDL) so that generic OData tools and applications can interpret it. It is defined in the JSON Schema format for other environments - like Python scripts, JavaScript code and visualization.

Structural properties of the resource are intended to be used as JavaScript variables. This should accelerate adoption and allow JavaScript web pages and enabled apps to use the data directly. URIs are persistent across reboots but clients are expected to start at /redfish/v1/ and do discovery of the URIs from there.

One common mistake is to fixate on the URI. Redfish is a hypermedia API so URIs can be different between implementations - even from the same vendor. Current state objects can be separate from desired state objects.

6. Operations

The operations are GET, PUT, PATCH, POST, DELETE and HEAD. GET is what your browser will do if you aren't using an advanced REST plug-in. Only the emulator and real implementations will support the other operations.

GET retrieves data. POST is used for creating resources or to use actions (more on this later). DELETE will delete a resource, but there are currently only a few resources that can be deleted. PATCH is used to change one or more properties on a resource while PUT is used to replace a resource entirely (though only a few resources can be completely replaced - more on this later too). HEAD is like a GET without the body data returned and can be used for figuring out the URI structure by programs accessing an Redfish implementation.

7. Versioning

Redfish has two kinds of versioning - the version of the protocol and the version of the resource schema. The version of the protocol is in the URI - that's why you should start at /redfish/v1/. It means you are accessing version one of the protocol. Version 1 is the only one available now, but we needed to accommodate potential future versions. This starting URI indicates the implementation complies with the Version 1 Redfish Specification. Note that since it is based on OData v4, implementations also require the OData protocol header (OData-Version) have a value of 4.

Each resource has a resource type definition. Resource types are defined in versioned namespaces. Each resource instance has the type represented using the OData type annotation "@odata.type". The value of the type annotation is the URI of the resource type, including the versioned namespace. So when you see "@odata.type" : "#ServiceRoot.v1_0_0.ServiceRoot", you are dealing with a resource that adheres to the ServiceRoot type definition, defined in 1.0.0 version of the ServiceRoot schema. The corresponding schema file would be located at /schema/v1/ServiceRoot in the Redfish schema repository. So the full URI for the type would be "/schema/v1/ServiceRoot#ServiceRoot.V1_0_0.ServiceRoot. The schema file may contain other types used in the resource type definition (for example, structured types and enums), which would have the same resource path but the fragment would describe a different type definition, typically within the same namespace.

8. References

When you see the links section, it will have references to other resources in it.

JSON has no native reference type to refer to another resource. Redfish requires a fully connected resource tree without requiring the client to consult schema (for simple operations). Therefore, this specification leverages OData conventions for representing a reference to another internal or external resource.

Properties representing references to other resources that follow OData conventions are identified with a property name suffixed with "@odata.id". Properties representing URLs to other types of references, for example an external help topic, are identified as string properties in metadata with an annotation specifying that they represent a URL.

URIs are either absolute or relative. Absolute ones won't have the IP address but will start with /redfish/v1/. If you have a plug in like the Chrome Advanced REST client, you can click on this to fill in the URI for your next GET.

9. Main Objects

The "main" objects are Systems, Managers and Chassis. These are all collections (see next heading).

We will dig into these resources in a minute, but it's good to know a bit about them.

Systems represent your typical server. Anything accessed in the data plane from the CPU is represented as a system and these are all in the systems collection. Thus they will have CPUs, Memory and other devices.

Managers represent your BMC, Enclosure Manager or other component that is managing the infrastructure. Managers handle various management services but also have their own devices (like NICs).

Chassis represents the physical aspects and containment of the infrastructure. Racks, enclosures, the blades within them - all of these and more are Chassis. Thus there is a way to represent a Chassis in a Chassis. It is the chassis that houses sensors, fans and the like.

Systems can have one or more managers (since some managers are redundant), and are in one chassis. Managers are in a Chassis and can manage more than one system. And Chassis can house more than one System and/or Managers.

This is just an overview, but looking at the ServiceRoot object, you can see these right away. You will also notice there are services such as Sessions (more on these later too).

10. Collections

Groups of similar resources are represented as collections. Examples of these in the Mockup include Systems, Managers, Chassis, LogEntries, Sessions, EventSubscriptions and more.

So pick either Systems, Managers or Chassis and go down into it. You will see this is a Collection.

Collection responses have a count property called "Members@odata.count" and a "Members" object that contains a list of members, or links to members. Links to members are represented as JSON objects with a single "@odata.id" property containing the URL of the member.

Collections may be paginated; collections with a property named "@odata.nextLink" are incomplete, and the client can use the URL-value of the next-link property to retrieve the next portion of the collection from the service.

11. Common Properties

As you go through the model, you keep seeing some of the same properties. You'll find "Name" & "Id" and in every resource. "Name" and "Id" are required. You'll also see "Status" as an embedded object that has the same definition across all usages. All of these are actually in a common part of the Schema and used by other Schema by reference. Common properties are referenced by each resource's schema via an odata "Reference" element within the schema so that the same definition is used everywhere. Examples

of these properties are:

- "Actions", which informs clients which actions can be invoked. (more on this in the Actions section)
- "Oem", which has vendor specific extensions to the standard definition of the resource. More on this in the Oem section.

12. Common Annotations

There are also properties that are annotations. These start with "@" or have "@" in them. There are two kinds of annotation properties allowed: Odata and DMTF annotations. OData annotations have "@odata." or have "@odata." in them. Examples of these properties are

- "@odata.type", which is used to find the schema that defines this resource.
- "@odata.id", which has the URL to this resource. This is an href, but since Redish is based on OData this property is called "@odata.id" and not "href".
- "Members@odata.count", which defines the number of resources in a collection.

DMTF annotations have "@Redfish." in them. Examples of these properies are:

- "@Redfish.Settings", which is used to indicate settings for the resource (more on this later).

One other common annotation is "@odata.context". This is really meant for generic OData v4 clients. This is properly defined in the OData v4 specification, but basically the @odata.context is used for a few different things:

1. It provides the location of the metadata that describes the payload.
2. It provides a root URL for resolving relative references

The structure of the @odata.context is the url to a metadata document with a fragment describing the data (typically rooted at the top-level singleton or collection).

Technically the metadata document only has to define, or reference, any of the types it directly uses, and different payloads could reference different metadata documents. However, since the @odata.context provides a root URL for resolving relative references (such as @odata.id's) we have to return the "canonical" metadata document. Further, because our "@odata.type" annotations are written as fragments, rather than full URLs, those fragments must be defined in, or referenced by, that metadata document. Also, because we qualify actions with the versionless namespace aliases, those aliases must also be defined through references in the referenced metadata document.

For example, in the resource /redfish/v1/Systems/1, you will see the property "@odata.context" with the value of "/redfish/v1/$metadata#Systems/Links/Members/$entity". This tells the generic OData v4 client to find the Systems definition in the $metadata and look in the Links property definition and within it is a Members property definition which has a reference to the definition for this entity.

You will also see an annotation called "@Redfish.Copyright". Implementations will not return this property. It is only here as a copyright statement for the static example responses used in the mockups.

13. Actions

Not everything can be done easily using REST. So we made Actions. Things like "push button" on a System (which would reset the system or turn it off, depending on its setting) can't easily be represented in the System because the service has no idea what the state of the button is and thus does not expose it as a property. Another use is for long lived operations that are more easily expressed as an atomic action as a convenience for the client - things like firmware update or graceful shutdown.

So instead of having hidden properties that you could PUT/PATCH to, or complex state machines, we created Actions. Actions are done with POSTs to the Resource (see the spec for specifics). You can tell what actions are supported in any resource by looking for the "Actions" property.

14. Schema

The schema contains a great deal of information that can be used by the client. It is recommended that Clients, especially programmatic clients, examine the schema for determining semantics around the properties. The schema contains data types, lists of enumerations, informative descriptions of properties, normative information about property behavior and other information.

More Concepts

That's the simplest form of the Redfish API. You'll notice the data is pretty readable, but you're probably wondering about security, what a schema is, what about headers, etc.

15. Sessions

Normally, only the root can be accessed without establishing a session. But if you are using the public mock-up or your local web server this won't be necessary as there is no security. If you have an emulator running in secure mode, or are working with a real implementation, you will need to establish a session.

First, you will need to know a valid user name and password for your implementation.

The URI for establishing as session is also allowed to be used for POST unsecured so that you may establish a session. This URI is /redfish/v1/Sessions in most cases and can be determined by looking at the `Sessions` property under `Links` and finding the `@odata.id` property's value in the service root (/redfish/v1/)

A session is created by an HTTP POST to the URI indicated by /redfish/v1#/Links/Sessions/@odata.id including the following POST body:

```
POST /redfish/v1/SessionService/Sessions HTTP/1.1
Host: <host-path>
Content-Type: application/json; charset=utf-8
Content-Length: <computed-length>
Accept: application/json
OData-Version: 4.0

{
    "UserName": "<username>",
    "Password": "<password>"
}
```

The return includes an X-Auth-Token header with a session token and Location header.

The return JSON body includes a representation of the newly created session object:

```
Location: /redfish/v1/SessionService/Sessions/1
X-Auth-Token: <session-auth-token>

{
    "@odata.context": "/redfish/v1/$metadata#SessionService/Sessions/$entity",
    "@odata.id": "/redfish/v1/SessionService/Sessions/1",
    "@odata.type": "#Session.v1_0_0.Session",
    "Id": "1",
    "Name": "User Session",
    "Description": "User Session",
    "UserName": "<username>"
}
```

You will use the token string in the response X-Auth-Token header in the same header for all subsequent requests to your service. When it's time to delete the session, you can do a DELETE operation on the URL that was returned in the @odata.id in the response (/redfish/v1/Sessions/Administrator1 in the example above).

16. Redundancy

Go back to one of the Chassis and take a look at the fans by following the link to "Thermal" and you will see how Redfish shows redundancy.

You will notice an array called "Redundancy". It shows the two fans in its set using the same values in the RelatedItem properties. Redundancy has a common schema definition in Redfish and has other properties in it besides the members to show other important attributes about redundancy. This is how the client can figure out which items belongs to which redundancy set since the @odata.id values are pointers to the redundancy set members.

The value of the "@odata.id" property, though, doesn't have to be to a whole resource. The value of this property will be of two formats: a JSON Pointer or an OData reference.

- In the case of a JSON Pointer, there will be a # in it that indicates where the resource stops and where the property pattern begins. The schema will also have a reference to the property. An example of a JSON Pointer value might be "/redfish/v1/Chassis/1/Thermal#/Fans/0".
- In the case of an OData reference, there will not be a # in it. The schema will have a definition of the property. An example of a JSON Pointer value might be "/redfish/v1/Chassis/1/Thermal/Fans/0".

17. RelatedItem

If you're still in the "Thermal" resource, you can see that the Temperature array elements have a

RelatedItem property with an "@odata.id" in it. This is a reference to the sub-resource that this temperature sensor is measuring - perhaps a processor. Like Redudancy links, the value may be to a sub-resource. Thus a client can determine what is being measured by this temperature sensor.

RelatedItem, while having a common schema definition, is situational dependent in it's usage. But it always is used to show a relationship between two different resources or sub-resources.

And like redundancy, the value of the "@odata.id" property doesn't have to be to a whole resource.

18. Services

Let's go back to root and take a look at some common services. Tasks, Sessions, EventService and AccountService are all common services. You can read more about them in the spec, but the service they provide should be fairly obvious by their name. Tasks contains a list of jobs that may have been started, usually as the result of Actions. Sessions was discussed above. AccountService is how you create users. EventService will be discussed more below.

19. Registries

Message registries allow the Management Processor to retain a short identifier for use in events and errors which clients can then look up in a registry to determine the actual message. Mechanisms are in place for parameter substitution on a per-message basis.

By having the message registries as separate entities, it is 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.

20. Headers

A subset of the common HTTP headers are supported by a Redfish service. A complete list of these are in the Specification, but the two that are most relevant are the Accept header (which must be set to 'application/json') and the X-Auth-Token discussed previously. But there are many more that you will get from an actual implementation that you will not see from a static mock-up.

21. ETags

ETags are used by browsers to optimize IO (caching). They do an If-Match and if it matches, they don't bother dragging the data. We use them to determine if an object has changed. Thus every ETag will change if a PUT is done or if a tool (like BIOS configuration at the console) takes place. Thus any race condition on PUTs between Redfish and non-Redfish clients (as well as other Redfish clients) are always

noticed.

The problem is that this is a mock-up and not a real web service so you can't see the ETags work.

21.1. Handling Client Race Conditions

It is possible for competing clients to attempt to set Setting Data overwriting each other. This condition is handled using ETags and If- Match.

A client read/modify/write cycle consists of:

1. Get the resource, including the current ETag
2. Modify the resource properties
3. Generate a new ETag
4. PATCH the modified resource including the If-Match header containing the last known ETag.
5. If the supplied ETag no longer matches the object (something has changed since the read), the service will return 304 Not Modified. The client should recover by repeating these steps.
6. Clients *should* include an ETag when updating Setting Data to enable Providers to more efficiently detect changes and to prevent multi-client race conditions.

22. Updating Resources

The simple GET, PUT/PATCH method is a model where the client may GET the current configuration and PUT or PATCH a new desired configuration to the same resource URI. The success of a PUT or PATCH is determined by the HTTP status code and response body.

This is intended for interaction with intrinsic providers such as the Redfish Services' own data where the firmware can process and respond appropriately to actions and configuration changes.

Upon PUT or PATCH, the HTTP response includes an HTTP status code and, in the case of failure, an Extended Error Information structure as the response body.

22.1. PUT vs PATCH

So why both PUT and PATCH? When designing this specification, we ran into the semantic issue of when to do a complete replacement of a resource instead of a partial replacement. Some resources allow either replacement of some properties or complete replacement of the resource. We needed a way to determine when to clear existing properties in resources like SettingData. Things started to get very convoluted and special (=unnatural) rules were put in place to try to use a single operation.

PATCH solves this. PUT always does a complete replacement. PATCH always does a partial

replacement. Deterministic behavior for the service is achieved without complex logic for either client or server.

PATCH is gaining wide adoption in the industry. It is already supported by Open Stack and many other APIs and in available in many off the shell web servers.

22.2. Current Configurations vs Settings

There are basically two kinds of objects in Redfish - Current Configurations and Settings. Most objects represent the current state of any given resource. Occasionally, you'll see a property called "@Redfish.Settings" in a resource. This annotation has a link tells you where to do PUTs and PATCHes for configuration. It represents the future state of the resource.

Some resources can handle changes to them right away, others may require a restart/reboot of the system or service. "@Redfish.Settings" is used to let the client know what type of resource this is and where to make the changes.

If you see the "@Redfish.Settings" property, it has a link to a resource to make the changes that will be picked up at the next opportunity, like a reset or reboot. If you don't see a Settings link, any PATCHes to the object should take place right away (in the absence of a spawned task).

There is also information about when the last time the settings were applied to the resource - time, an ETag and any messages that would have been returned if the settings had been able to be applied "live".

Examples of resources that may need Settings are devices like NICs and Storage as well as BIOS.

22.3. Determining Properties that can be updated.

Both the Schema and the metadata define which properties can be updated. Properties that are marked "OData.Permissions/Read" or have read-only as true indicate these properties can never be updated. Instead an "OData.Permissions/ReadWrite" or read-only = false indicates that it might be writable. The LongDescription (normative text) may indicate which properties are writable and under what conditions. Note that by default, metadata without a "Permissions" annotation or schema without readonly = true can assumed to be writable.

Note that even if they are marked writable, that doesn't necessarily indicate that a property can be written as implementations can allow the property to be read-only.

The permission on a complex type property is the default permission for all properties within the complex type that do not have a "Permissions" annotation. For example the SubnetMask property in the IPv4Address complex type has no "Permissions" annotation. The IPv4Addresses property in EthernetInterface is of type IPv4Address. If IPv4Addresses has an "OData.Permission/Read" annotation, then SubnetMask becomes read-only. If IPv4Addresses has an "OData.Permission/ReadWrite" annotation, then SubnetMask becomes read-write. For clarity, the best practice is to define a

"Permissions" annotation for all properties within the complex type.

Semantics on Redfish PUT/PATCH are such that attempting to update a non-writable property isn't treated as an error. An implementation of a Redfish Service will return a 200 with Extended Error Information indicating which properties were unable to be updated.

23. Extended Error Response

HTTP error semantics alone are not informative enough for users or applications to understand the cause and take corrective action, leading to the concept of an Extended Error Response construct. It also supports registries, and allows for more meaningful error semantics for clients.

For example, let's suppose a client wants to change a couple dozen properties all at once. But one of them is invalid. If the implementation returned a "400", it would not help the client know which property was in error and why. Or if it returns a "200" but one of the properties didn't exist, that isn't a meaningful response.

If the response is accompanied by a JSON body with an Extended Error Response, the structure can contain not only the reason for the code being returned but the property name and even more information. And since ExtendedError has an array of messages, more than one message can be returned if more than one problem was encountered.

24. Eventing

We need some kind of eventing mechanism to meet comp with SNMP, IPMI and other protocols. The mechanism is already there in BMCs. The methodology chosen was a PUSH mechanism based on a subscription. PUSH eventing is preferred for BMCs since it means that the event does not to be persistent in memory once the event has been sent.

A client first needs to determine the URI it wants Events to which events will be sent. By submitting a POST request to the EventSubscription collection, a session may be established.

Redfish supports two different categories of events: life-cycle and alert events. You can look at the EventService for the types of eventing supported by the implementation.

More information on eventing is in the specification.

25. Creating User Accounts and other resources

Like Eventing and Sessions, resources are created via a POST operation. This is usually through a POST to a collection. Event subscription, sessions and account services all have collections. The URI of these

collection are discovered like any other resource in this hypermedia API.

The schema defines an annotation called "RequiredOnCreate" that a client can use to determine which properties must be supplied in the POST.

26. Message Alignment

One thing the architects attempted in Redfish was to align all of the messaging formats. Event Messages, Extended Error Response Messages, Logs Entries and Message Registries are all aligned. While vendor specific & other formats can be preserved in an implementation, there is a path forward for message format alignment enabling not only internationalization of messages but optimizing storage and hopefully unifying both client and service tasks that access and present this data.

27. Oem

In the mockups, and probably implementations, you will see a construct called "Oem". This construct can occur in 3 places: as a base property of a resource, in a "Links" section or in the "Actions" object. Redfish defines a mechanism so that vendors can include their own extensions by adding their own object within the "Oem" object. It must have an "@odata.type" property so that clients can find schema files for it. Beyond that, the standard makes no claims about requirements but it is a mechanism where a multi-vendor environment could potentially place extensions in with the same construct as the standard resource. This prevents the need for more IOs and allows for scalability. It is hoped that vendors will bring in features to become part of the standard over time.

28. Idempotency

RESTful operations for any RESTful service are GET, PUT/PATCH, POST, and DELETE.

Idempotency is an important concept for Redfish simplicity. Idempotent operations can be invoked repeatedly without additional effect after the 1st time. For example: The STOP button on a DVD remote is idempotent, push it once and the movie stops, push it more than once and the movie remains stopped. The Pause button, on the other hand, is not idempotent. If you push it twice and it starts to play again. Non-idempotent operations depend upon previous state.

Idempotent operators are HTTP PUT and PATCH. If a client PUTs the same data a second time, the resource does not change. PATCH replaces properties so a replayed PATCH does not change a resource again because it is already in the desired state from the 1st PATCH.

PUTting to a resource is the simplest possible expression of altering configuration. It means "replace resource X with Y". Other interaction patterns are more complex than this. A PATCH is slightly more complex since it requires differential updating of a resource. A service than can be completely

configurable via idempotent PATCH operations is ideal.

Redfish defines three basic interaction patterns using the basic REST operations.

28.1. Idempotent Modify: GET/PUT/PATCH

PATCHing to a resource replaces the content of the resource with new values for the supplied properties. This is used in cases where a service can simply consume desired configuration and produce resulting state. Presumably, a client would likely GET a resource, modify property values, and then PATCH to commit changes.

PUTing to a resource replaces the content of the resource with new values. This is used in cases with a client can simply set a resource to desired state.

28.2. Create, Use, Delete: POST/GET/DELETE (non-idempotent)

POSTing to a resource creates a new instance of a child resource. This is typically used for data models where items must be created and deleted. Examples include creating new user accounts, deleting user accounts, new log entries, adding or removing licenses, etc. These items may support delete as well. Creation is performed by POSTing content to a URI. The newly created resource is a child of that URI and may be deleted by DELETE to the new URI.

28.3. Do Action: POST with "Action" property (non-idempotent)

POSTing to an action returned in the Actions property of a resource results in the execution of the custom action. This pattern is not idempotent because re-posting the same content will result in additional work being performed. An example of a custom action might be sending a test event or clearing a log.

29. Inheritance & Polymorphism

One of the items you will not see much of us Inheritance or Polymorphism.
Object inherit from a single master object called Resource. Other than that, Redfish doesn't use inheritance. Likewise, Redfish doesn't take common properties and put them into a parent object. Thus, there are no sub-objects to specialize from any schema. So you will never see sub-types returned in the same collection along with a parent type.

30. Inheritance by Copy

Redfish takes common definitions and, if they aren't in Resource, the authors just cut and paste the objects that are needed from other schema and include them in the object. This does place a burden of

keeping things in sync on the authors but it also prevents the reader from digging through reference into another reference into another reference just to find out the semantics of any given object, property or action.

31. Polymorphism by Union

Redfish takes resources that have semantics in common and puts the definition all in the same resource. This way, similar resources will have similar semantics. Implementers then just don't use the properties that aren't supported. So, for example, if there are 3 different devices that are 85% similar, the 15% difference of each device is included in the common schema definiiion.
Implementers would then use the parts of the 85% they need for their resource as well as the 15% of the definition that applies to the device they are implementing. This does make for larger schema, but also makes for fewer schema. When you see this approach, you will often see an extra "resource type" style property in the schema and resource to help clients tell what flavor this device is.

Conclusion

The Redfish API represents a new style of programming for IT that is capable of managing systems from hyper-scale to blades to stand alone servers in a consistent manner. We believe it represents a natural evolution that fits customer demand as well as the tools being used by those customers.

Common Use Cases

Here are a few use cases that will help you understand the architecture and begin your client code.

32. Finding Temperature Sensors for a System

Application code should always start at the root: /redfish/v1/

1. In the root object is a property called "Systems".
    1. Find the "@odata.id" value. This is the URI of the Systems collection.
    2. In the mockup, this URI is /redfish/v1/Systems.
    3. Do a GET on that URI.
2. Look at the "Links" object at the "Members" array.
    1. Find the "@odata.id" of the System in question (this may require GETs on each system and looking at properties in the system to determine the right system).
    2. In the mockup, this URI is /redfish/v1/Systems/1.
    3. Do a GET on that URI.
3. Look in the "Links" object for an object called "Chassis".
    1. Find the "@odata.id" value. This is the chassis that this system is in.
    2. In the mockup, this URI is /redfish/v1/Chassis/1.
    3. Do a GET on that URI.
4. Look in the "Links" section for an object called "Thermal".
    1. Find the "@odata.id" value. This is has the temperature sensors in it.
    2. In the mockup, this URI is /redfish/v1/chassis/1/Thermal.
    3. Do a GET on that resource.
5. Look in the "Temperature" array - this is the temperature sensors for a system.
    1. Look at the "RelatedItem" to find out specifically which components each temperature sensor monitors.

33. Chassis within Chassis

Application code should always start at the root: /redfish/v1/

1. In the root object is a property called "Chassis".
    1. Find the "@odata.id" value. This is the URI of the Chassis collection.
    2. In the mockup, this URI is /redfish/v1/Chassis.

3. Do a GET on that URI.

2. Look at the "Links" object at the "Members" array.
    1. Find the "@odata.id" of the Chassis in question (this may require GETs on each system and looking at properties in the chassis to determine the right system).
    2. In the mockup, this URI is /redfish/v1/Chassis/Enc1.
    3. Do a GET on that URI.
3. Look in the "Links" object for an object called "Contains".
    1. Find the "@odata.id" value. This is the chassis that are in this chasis.
4. Look in the "Links" section for an object called "ContainedBy".
    1. Find the "@odata.id" value. This is the chassis that this chassis is in.

Revision History

Version	Date	Description
1.0.0	2015-8	Initial release
1.0.1	2015-8	Corrected @DMTF to @Redfish
1.0.2	2017-4	Added Statement about Copyright Annotation in Mockups