


This repository


Search


Pull requests


Issues

Gist







 **DMTF / spm** PRIVATE

Unwatch

39

Star

1







Fork

9

branch: master

spm / Specification.md

 **jautor** 15 minutes ago Updated Change Log


9 contributors         


2377 lines (1636 sloc) 183.244 kB


Raw

Blame

History







DocTitle	DocNumber	DocType	DocVersion	DocStatus	DocConfidentiality	expiration	released	copyright
Redfish Scalable Platforms Management API Specification	0266	Specification	1.00.0	Work in Progress	– Not a DMTF Standard – DMTF Confidential	2015-08-24	false	2014-2015

Foreword

The Redfish Scalable Platform Management API ("Redfish") was prepared by the Scalable Platforms Management Forum of the DMTF.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. For information about the DMTF, see <http://www.dmtf.org>.

Acknowledgments

The DMTF acknowledges the following individuals for their contributions to this document:

- Jeff Autor - Hewlett-Packard Company
- David Brockhaus - Emerson Network Power
- Richard Brunner - VMware Inc.
- Lee Calcote - Seagate Technology
- P Chandrasekhar - Dell Inc
- Chris Davenport - Hewlett-Packard Company
- Gamma Dean - Emerson Network Power
- Wassim Fayed - Microsoft Corporation
- Mike Garrett - Hewlett-Packard Company
- Steve Geffin - Emerson Network Power
- Jon Hass - Dell Inc
- Jeff Hilland - Hewlett-Packard Company
- Chris Hoffman - Emerson Network Power
- John Leung - Intel Corporation
- Milena Natanov - Microsoft Corporation
- Michael Pizzo - Microsoft Corporation
- Irina Salvan - Microsoft Corporation
- Hemal Shah - Broadcom Corporation
- Jim Shelton - Emerson Network Power
- Tom Slaight - Intel Corporation

- Donnie Sturgeon - Emerson Network Power
- Pawel Szymanski - Intel Corporation
- Paul Vancil - Dell Inc
- Linda Wu - Super Micro Computer, Inc.

Abstract

The Redfish Scalable Platforms Management API ("Redfish") is a new specification that uses RESTful interface semantics to access data defined in model format to perform out-of-band systems management. It is suitable for a wide range of servers, from stand-alone servers to rack mount and bladed environments but scales equally well for large scale cloud environments.

There are several out of band systems management standards (defacto and de jour) available in the industry. They all either vary widely in implementation, were developed for single server embedded environments or have their roots in antiquated software modeling constructs. There is no single industry standard that is simple to use, based on emerging programming standards, embedded friendly and capable of meeting large scale data center & cloud needs.

Normative References

The following referenced documents are indispensable for the application of this document. For dated or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies. For references without a date or version, the latest published edition of the referenced document (including any corrigenda or DMTF update versions) applies.

- IETF RFC 2616, R. Fielding et al., HTTP/1.1, <http://www.ietf.org/rfc/rfc2616.txt>
- IETF RFC 2617 J. Franks et al., HTTP Authentication: Basic and Digest Access Authentication, <http://www.ietf.org/rfc/rfc2617.txt>
- IETF RFC 3986 T. Berners-Lee et al, Uniform Resource Identifier (URI): Generic Syntax, <http://www.ietf.org/rfc/rfc3986.txt>
- IETF RFC 4627, D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), <http://www.ietf.org/rfc/rfc4627.txt>
- IETF RFC 4627, L. Dusseault et al, PATCH method for HTTP, <http://www.ietf.org/rfc/rfc5789.txt>
- IETF RFC 5988, M. Nottingham, Web linking, <http://www.ietf.org/rfc/rfc5988.txt>
- IETF RFC 6901, P. Bryan, Ed. et al, JavaScript Object Notation (JSON) Pointer, <http://www.ietf.org/rfc/rfc6901.txt>
- IETF RFC 6906, E. Wilde, The 'profile' Link Relation Type, <http://www.ietf.org/rfc/rfc6906.txt>
- ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards, <http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtypeH>
- JSON Schema, Core Definitions and Terminology, Draft 4 <http://tools.ietf.org/html/draft-zyp-json-schema-04.txt>
- JSON Schema, Interactive and Non-Interactive Validation, Draft 4 <http://tools.ietf.org/html/draft-fge-json-schema-validation-00.txt>
- OData Version 4.0 Part 1: Protocol. 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/odata-v4.0-part1-protocol.html>
- OData Version 4.0 Part 2: URL Conventions. 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/odata-v4.0-part2-url-conventions.html>
- OData Version 4.0 Part 3: Common Schema Definition Language (CSDL). 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/odata-v4.0-part3-csdl.html>
- OData Version 4.0: Core Vocabulary. 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Core.V1.xml>
- OData Version 4.0 JSON Format. 24 February 2014. <http://docs.oasis-open.org/odata/odata-json-format/v4.0/os/odata-json-format-v4.0-os.html>
- OData Version 4.0: Units of Measure Vocabulary. 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml>

Terms and Definitions

In this document, some terms have a specific meaning beyond the normal English meaning. Those terms are defined in this clause.

The terms "shall" ("required"), "shall not", "should" ("recommended"), "should not" ("not recommended"), "may", "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Annex H. The terms in parenthesis are alternatives for the preceding term, for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that ISO/IEC Directives, Part 2, Annex H specifies additional alternatives. Occurrences of such additional alternatives shall be interpreted in their normal English meaning.

The terms "clause", "subclause", "paragraph", and "annex" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 5.

The terms "normative" and "informative" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 3. In this document, clauses, subclauses, or annexes labeled "(informative)" do not contain normative content. Notes and examples are always informative elements.

The following additional terms are used in this document.

Term	Definition
Baseboard Management Controller	An embedded device or service, typically an independent microprocessor or System-on-Chip with associated firmware, within a Computer System used to perform systems monitoring and management-related tasks, which are commonly performed out-of-band.
Collection	A Collection is a resource that acts as a container of other Resources. The members of a collection usually have similar characteristics. The container processes messages sent to the container. The members of the container process messages sent only to that member without affecting other members of the container.
CRUD	Basic intrinsic operations used by any interface: Create, Read, Update and Delete.
Event	A record that corresponds to an individual alert.
Managed System	In the context of this specification, a managed system is a system that provides information or status, or is controllable, via a Redfish-defined interface.
Message	A complete request or response, formatted in HTTP/HTTPS. The protocol, based on REST, is a request/response protocol where every Request should result in a Response.
Operation	The HTTP request methods which map generic CRUD operations. These are POST, GET, PUT/PATCH, HEAD and DELETE.
OData	The Open Data Protocol, as defined in OData-Protocol .
OData Service Document	The name for a resource that provides information about the Service Root. The Service Document provides a standard format for enumerating the resources exposed by the service that enables generic hypermedia-driven OData clients to navigate to the resources of the Redfish Service.
Redfish Alert Receiver	The name for the functionality that receives alerts from a Redfish Service. This functionality is typically software running on a remote system that is separate from the managed system.
Redfish Client	Name for the functionality that communicates with a Redfish Service and accesses one or more resources or functions of the Service.
Redfish Protocol	The set of protocols that are used to discover, connect to, and inter-communicate with a Redfish Service.
Redfish	The Schema definitions for Redfish resources. It is defined according to OData Schema

Schema	representation that can be directly translated to a JSON Schema representation.
Redfish Service	Also referred to as the "Service". The collection of functionality that implements the protocols, resources, and functions that deliver the interface defined by this specification and its associated behaviors for one or more managed systems.
Redfish Service Entry Point	Also referred to as "Service Entry Point". The interface through which a particular instance of a Redfish Service is accessed. A Redfish Service may have more than one Service Entry Point.
Request	A message from a Client to a Server. It consists of a request line (which includes the Operation), request headers, an empty line and an optional message body.
Resource	A Resource is addressable by a URI and is able to receive and process messages. A Resource can be either an individual entity, or a collection that acts as a container for several other entities.
Resource Tree	A Resource Tree is a tree structure of JSON encoded resources accessible via a well-known starting URI. A client may discover the resources available on a Redfish Service by following the resource links from the base of the tree. NOTE for Redfish client implementation: Although the resources are a tree, the references between resources may result in graph instead of a tree. Clients traversing the resource tree must contain logic to avoid infinite loops.
Response	A message from a Server to a Client in response to a request message. It consists of a status line, response headers, an empty line and an optional message body.
Service Root	The term Service Root is used to refer to a particular resource that is directly accessed via the service entry point. This resource serves as the starting point for locating and accessing the other resources and associated metadata that together make up an instance of a Redfish Service.
Subscription	The act of registering a destination for the reception of events.

Symbols and Abbreviated Terms

The following additional abbreviations are used in this document.

Term	Definition
BMC	Baseboard Management Controller
CRUD	Create, Replace, Update and Delete
CSRF	Cross-Site Request Forgery
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over TLS
IP	Internet Protocol
IPMI	Intelligent Platform Management Interface
JSON	JavaScript Object Notation
KVM-IP	Keyboard, Video, Mouse redirection over IP
NIC	Network Interface Card
PCI	Peripheral Component Interconnect
PCIe	PCI Express

TCP	Transmission Control Protocol
XSS	Cross-Site Scripting

Overview

The Redfish Scalable Platform Management API ("Redfish") is a management standard using a data model representation inside of a hypermedia RESTful interface. Because it is based on REST, Redfish is easier to use and implement than many other solutions. Since it is model oriented, it is capable of expressing the relationships between components in modern systems as well as the semantics of the services and components within them. It is also easily extensible. By using a hypermedia approach to REST, Redfish can express a large variety of systems from multiple vendors. By requiring JSON representation, a wide variety of resources can be created in a denormalized fashion not only to improve scalability, but the payload can be easily interpreted by most programming environments as well as being relatively intuitive for a human examining the data. The model is exposed in terms of an interoperable Redfish Schema, expressed in an OData Schema representation with translations to a JSON Schema representation, with the payload of the messages being expressed in a JSON following OData JSON conventions. The ability to externally host the Redfish Schema definition of the resources in a machine-readable format allows the meta data to be associated with the data without encumbering Redfish services with the meta data, thus enabling more advanced client scenarios as found in many data center and cloud environments.

Scope

The scope of this specification is to define the protocols, data model, and behaviors, as well as other architectural components needed for an inter-operable, cross-vendor, remote and out-of-band capable interface that meets the expectations of Cloud and Web-based IT professionals for scalable platform management. While large scale systems are the primary focus, the specifications are also capable of being used for more traditional system platform management implementations.

The specifications define elements that are mandatory for all Redfish implementations as well as optional elements that can be chosen by system vendor or manufacturer. The specifications also define points at which OEM (system vendor) - specific extensions can be provided by a given implementation.

The specifications set normative requirements for Redfish services and associated materials, such as Redfish Schema files. In general, the specifications do not set requirements for Redfish clients, but will indicate what a Redfish client should do in order to access and utilize a Redfish Service successfully and effectively.

The specifications do not set requirements that particular hardware or firmware must be used to implement the Redfish interfaces and functions.

Principle Goals

There are many objectives and goals of Redfish as an architecture, as a data representation, and of the definition of the protocols that are used to access and interact with a Redfish service. Redfish seeks to provide specifications that meet the following goals:

- Scalable – To support stand-alone machines to racks of equipment found in cloud service environments.
- Flexible - To support a wide variety of systems found in service today.
- Extensible – To support new and vendor-specific capabilities cleanly within the framework of the data model.
- Backward Compatible– To enable new capabilities to be added while preserving investments in earlier versions of the specifications.
- Interoperable – To provide a useful, required baseline that ensures common level of functionality and implementation consistency across multiple vendors.
- System-Focused – To efficiently support the most commonly required platform hardware management capabilities that are used in scalable environments, while also being capable of managing current server environments.

- Standards based - To leverage protocols and standards that are widely accepted and used in environments today - in particular, programming environments that are being widely adopted for developing web-based clients today.
- Simple – To be directly usable by software developers without requiring highly specialized programming skills or systems knowledge.
- Lightweight - To reduce the complexity and cost of implementing and validating Redfish Services on managed systems.

Design Tenets

The following design tenets and technologies are used to help deliver the previously stated goals and characteristics:

- Provide a RESTful interface using a JSON payload and an Entity Data Model
- Separate protocol from data model, allowing them to be revised independently
- Specify versioning rules for protocols and schema
- Leverage strength of internet protocol standards where it meets architectural requirements, such as JSON, HTTP, OData, and the RFCs referenced by this document.
- Focus on out-of-band access -- implementable on existing BMC and firmware products
- Organize the schema to present value-add features alongside standardized items
- Make data definitions as obvious in context as possible
- Maintain implementation flexibility. Do not tie the interface to any particular underlying implementation architecture. "Standardize the interface, not the implementation."
- Focus on most widely used 'common denominator' capabilities. Avoid adding complexity to address functions that are only valued by a small percentage of users.
- Avoid placing complexity on the management controller to support operations that can be better done at the client.

Limitations

Redfish does not guarantee that client software will never need to be updated. Examples that may require updates include accommodation of new types of systems or their components, data model updates, and so on. System optimization for an application will always require architectural oversight. However, Redfish does attempt to minimize instances of forced upgrades to clients using Schemas, strict versioning and forward compatibility rules and through separation of the protocols from the data model.

Inter-operable does not mean identical. A Redfish client may need to adapt to the optional elements that are provided by different vendors. Implementation and configurations of a particular product from a given vendor can also vary.

For example, Redfish does not enable a client to read a Resource Tree and write it to another Redfish Service. This is not possible as it is a hypermedia API. Only the root object has a well known URI. The resource topology reflects the topology of the system and devices it represents. Consequently, different server or device types will result in differently shaped resource trees, potentially even for identical systems from the same manufacturer.

Additionally, not all Redfish resources are simple read/write resources. Implementations may follow other interaction patterns discussed later. As an example, user credentials or certificates cannot simply be read from one service and transplanted to another. Another example is the use of Setting Data instead of writing to the same resource that was read from.

Lastly, the value of links between resources and other elements can vary across implementations. Clients should not assume that links can be reused across different instantiations of a Redfish service.

Additional Design Background and Rationale

REST based

This document defines a RESTful interface. Many service applications are exposed as RESTful interfaces.

There are several reasons to define a RESTful interface:

- It enables a lightweight implementation, where economy is a necessity (smaller data transmitted than SOAP, fewer layers to the protocol than WS-Man).
- It is a prevalent access method in the industry.
- It is easy to learn and easy to document.
- There are a number of toolkits & development environments that can be used for REST.
- It supports data model semantics and maps easily to the common CRUD operations.
- It fits with our design principle of simplicity.
- It is equally applicable to software application space as it is for embedded environments thus enabling convergence and sharing of code of components within the management ecosystem.
- It is schema agnostic so adapts well to any modeling language.
- By using it, Redfish can leverage existing security & discovery mechanisms in the industry.

Follow OData Conventions

With the popularity of RESTful APIs, there are nearly as many RESTful interfaces as there are applications. While following REST patterns helps promote good practices, due to design differences between the many RESTful APIs there is no interoperability between them.

OData defines a set of common RESTful conventions and markup which, if adopted, provides for interoperability between APIs.

Adopting OData conventions for describing Redfish 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 client libraries, applications, and tools.

Model Oriented

The Redfish model is built for managing systems. All resources are defined in OData Schema representation and translated to JSON Schema representation. OData is an industry standard that encapsulates best practices for RESTful services and provides interoperability across services of different types. JSON is being widely adopted in multiple disciplines and has a large number of tools and programming languages that accelerate development when adopting these approaches.

Separation of Protocol from Data Model

The protocol operations are specified independently of the data model. The protocols are also versioned independently of the data model. The expectation is that the protocol version changes extremely infrequently, while the data model version is allowed to change as needed. This implies that innovation should happen primarily in the data model, not the protocols. It allows the data model to be extended and changed as needed without requiring the protocols or API version to change. Conversely, separating the protocols from the data model allows for changes to occur in the protocols without causing significant changes to the data model.

Hypermedia API Service Endpoint

Like other hypermedia APIs, Redfish has a single service endpoint URI and all other resources are accessible via opaque URIs referenced from the root. Any resource discovered through links found by accessing the root service or any service or resource referenced using references from the root service will conform to the same versions of the protocols supported by the root service.

Note that the ServiceRoot Redfish Schema places requirements on the last segment of the path for the URIs discoverable through the service root.

Service Elements

Synchronous and Asynchronous Operation Support

While the majority of operations in this architecture are synchronous in nature, some operations can take a long time to execute, more time than a client typically wants to wait. For this reason, some operations can be asynchronous at the discretion of the service. The request portion of an asynchronous operation is no different from the request portion of a synchronous operation.

The use of HTTP Response codes enable a client to determine if the operation was completed synchronously or asynchronously. For more information see the section on [Tasks](#).

Eventing Mechanism

In some situations it is useful for a service to provide messages to clients that fall outside the normal request/response paradigm. These messages, called events, are used by the service to asynchronously notify the client of some significant state change or error condition, usually of a time critical nature.

Only one style of eventing is currently defined by this specification - push style eventing. In push style eventing, when the server detects the need to send an event, it uses an HTTP POST to push the event message to the client. Clients can subscribe to the eventing service to enable reception of events by creating a EventDestination subscription entry in the Event Service, or an administrator can create subscriptions as part of the Redfish service configuration. All subscriptions are persistent configuration settings.

Events originate from a specific resource. Not all resources are able to generate events. Those resources capable of generating events might not generate any events unless a subscription has been created to listen for the event. An administrator or client creates a subscription by sending a "subscribe" message to the Event Service. A subscribe message is sent using HTTP POST to the Event Subscriptions collection.

The Section on [Eventing](#) further in this specification discusses the details of the eventing mechanism.

Actions

Operations can be divided into two sets: intrinsic and extrinsic. Intrinsic operations, often referred to as CRUD, are mapped to [HTTP methods](#). The protocol also has the ability to support extrinsic operations -- those operations that do not map easily to CRUD. Examples of extrinsic would be items that collectively would be better performed if done as a set (for scalability, ease of interface, server side semantic preservation or similar reasons) or operations that have no natural mapping to CRUD operations. One examples is system reset. It is possible to combine multiple operations into a single action. A system reset could be modeled as an update to state, but semantically the client is actually requesting a state change and not simply changing the value in the state.

In Redfish, these extrinsic operations are called **actions** and are discussed in detail in different parts of this specification.

The Redfish Schema defines certain standard actions associated with [common Redfish resources](#). For these standard actions, the Redfish Schema contains the normative language on the behavior of the action. OEM extensions are also allowed to the Redfish [schema](#), including defining [actions](#) for existing resources.

Service Entry Point Discovery

While the service itself is at a well-known URI, the service host must be discovered. Redfish, like UPnP, uses SSDP for discovery. SSDP is supported in a wide variety of devices, such as printers. It is simple, lightweight, IPv6 capable and suitable for implementation in embedded environments. Redfish is investigating additional service entry point discovery (e.g. DHCP-based) approaches.

For more information, see the section on [Discovery](#)

Remote Access Support

A wide variety of remote access and redirection services are supported in this architecture. Critical to out-of-band

environments are mechanisms to support Serial Console access, Keyboard Video and Mouse re-direction (KVM-IP), Command Shell (i.e. Command Line interface) and remote Virtual Media. Support for Serial Console, Command Shell, KVM-IP and Virtual Media are all encompassed in this standard and are expressed in the Redfish Schema. This standard does not define the protocols or access mechanisms for accessing those devices and services. The Redfish Schema provides for the representation and configuration of those services, establishment of connections to enable those services and the operational status of those services. However, the specification of the protocols themselves are outside the scope of this specification.

Security

The challenge with security in a remote interface that is programmatic is to ensure both the interfaces used to interact with Redfish and the data being exchanged are secured. This means designing the proper security control mechanisms around the interfaces and securing the channels used to exchange the data. As part of this, specific behaviors are to be put in place including defining and using minimum levels of encryption for communication channels etc.

Protocol Details

The Redfish Scalable Platform Management API is based on REST and follows OData conventions for interoperability, as defined in [OData-Protocol](#), JSON payloads, as defined in [OData-JSON](#), and a machine-readable representation of schema, as defined in [OData-Schema](#). The OData Schema representations include annotations to enable direct translation to JSON Schema representations for validation and consumption by tools supporting JSON Schema. Following these common standards and conventions increases interoperability and enables leveraging of existing tool chains.

Redfish follows the OData minimal conformance level for clients consuming minimal metadata.

Throughout this document, we refer to Redfish as having a protocol mapped to a data model. More accurately, HTTP is the application protocol that will be used to transport the messages and TCP/IP is the transport protocol. The RESTful interface is a mapping to the message protocol. For simplicity though, we will refer to the RESTful mapping to HTTP, TCP/IP and other protocol, transport and messaging layer aspects as the Redfish protocol.

The Redfish protocol is designed around a web service based interface model, and designed for network and interaction efficiency for both user interface (UI) and automation usage. The interface is specifically designed around the REST pattern semantics.

[HTTP methods](#) are used by the Redfish protocol for common CRUD operations and to retrieve header information.

[Actions](#) are used for expanding operations beyond CRUD type operations, but should be limited in use.

[Media types](#) are used to negotiate the type of data that is being sent in the body of a message.

[HTTP status codes](#) are used to indicate the server's attempt at processing the request. [Extended error handling](#) is used to return more information than the HTTP error code provides.

The ability to send secure messages is important; the [Security](#) section of this document describes specific TLS requirements.

Some operations may take longer than required for synchronous return semantics. Consequently, [deterministic asynchronous semantic](#) are included in the architecture.

Use of HTTP

HTTP is ideally suited to a RESTful interface. This section describes how HTTP is used in the Redfish interface and what constraints are added on top of HTTP to assure interoperability of Redfish compliant implementations.

- A Redfish interface shall be exposed through a web service endpoint implemented using Hypertext Transfer

Protocols, version 1.1 ([RFC2616](#)).

URIs

A URI is used to identify a resource, including the base service and all Redfish resources.

- A URI shall be a unique identifier to a resource.
- A URI shall be treated by the client as opaque, and thus should not be attempted to be understood or deconstructed by the client

To begin operations, a client must know the URI for a resource.

- Performing a GET operation yields a representation of the resource containing properties and links to associated resources.

The base resource URI is well known and is based on the protocol version. Discovering the URIs to additional resources is done through observing the associated resource links returned in previous responses. This type of API that is consumed by navigating URIs returned by the service is known as a Hypermedia API.

The URI is the primary unique identifier of resources. Redfish considers three parts of the URI as described in [RFC3986](#).

The first part includes the scheme and authority portions of the URI. The second part includes the root service and version. The third part is a unique resource identifier.

For example, in the following URL:

Example: `https://mgmt.vendor.com/redfish/v1/Systems/1`

- The first part is the scheme and authority portion (<https://mgmt.vendor.com>).
- The second part is the root service and version (`/redfish/v1/`).
- The third part is the unique resource path (`Systems/1`).

The scheme and authority part of the URI shall not be considered part of the unique *identifier* of the resource. This is due to redirection capabilities and local operations which may result in the variability of the connection portion. The remainder of the URI (the service and resource paths) is what *uniquely identifies* the resource, and this is what is returned in all Redfish payloads.

- The unique identifier part of a URI shall be unique within the implementation.

For example, a POST may return the following URI in the Location header of the response (indicating the new resource created by the POST):

Example: `/redfish/v1/Systems/2`

Assuming the client is connecting through an appliance named "mgmt.vendor.com", the full URI needed to access this new resource is <https://mgmt.vendor.com/redfish/v1/Systems/2>.

URIs, as described in [RFC3986](#), may also contain a query (`?query`) and a frag (`#frag`) components. Queries are addressed in the section [Query Parameters](#). Fragments (frag) shall be ignored by the server when used as the URI for submitting an operation.

HTTP Methods

An attractive feature of the RESTful interface is the very limited number of operations which are supported. The following table describes the general mapping of operations to HTTP methods. If the value in the column entitled "required" has the value "yes" then the HTTP method shall be supported by a Redfish interface.

HTTP Method	Interface Semantic	Required
POST	Object create, Object action, Eventing	Yes
GET	Object or Collection retrieval	Yes
PUT	Object replace	No
PATCH	Object update	Yes
DELETE	Object delete	Yes
HEAD	Object or Collection header retrieval	No

Other HTTP methods are not allowed and shall receive a [405](#) response.

HTTP Redirect

HTTP redirect allows a service to redirect a request to another URL. Among other things, this enables Redfish resources to alias areas of the data model.

- All Redfish Clients shall correctly handle HTTP redirect.

NOTE: Refer to the [Security](#) section for security implications of HTTP Redirect

Media Types

Some resources may be available in more than one type of representation. The type of representation is indicated by the media type.

In HTTP messages the media type is specified in the Content-Type header. A client can tell a service that it wants the response to be sent using certain media types by setting the HTTP Accept header to a list of the acceptable media types.

- All resources shall be made available using the JSON media type "application/json".
- Redfish services shall make every resource available in a representation based on JSON, as specified in [RFC4627](#). Receivers shall not reject a message because it is encoded in JSON, and shall offer at least one response representation based on JSON. An implementation may offer additional representations using non-JSON media types.

Clients may request compression by specifying an [Accept-Encoding header](#) in the request.

- Responses to GET requests shall only be compressed if requested by the client.
- Services should support gzip compression when requested by the client.

ETags

In order to reduce the cases of unnecessary RESTful accesses to resources, the Redfish Service should support associating a separate ETag with each resource.

- Implementations should support returning [ETag properties](#) for each resource.
- Implementations should support returning ETag headers for each response that represents a single resource. Implementations shall support returning ETag headers for certain requests and responses as listed in the [Security](#) section.

The ETag is generated and provided as part of the resource payload because the service is in the best position to know if the new version of the object is different enough to be considered substantial. There are two types of ETags: weak and strong.

- Weak model -- only "important" portions of the object are included in formulation of the ETag. For instance, meta-data such as a last modified time should not be included in the ETag generation. The "important" properties that

determine ETag change include writable settings and changeable attributes such as UUID, FRU data, serial numbers, etc.

- Strong model -- all portions of the object are included in the formulation of the ETag.

This specification does not mandate a particular algorithm for creating the ETag, but ETags should be highly collision-free. An ETag could be a hash, a generation ID, a time stamp or some other value that changes when the underlying object changes.

If a client [PUTs](#) or [PATCHes](#) a resource, it should include an ETag in the HTTP If-Match/If-None-Match header from a previous GET.

In addition to returning the ETag property on each resource,

- A Redfish Service should return the ETag header on client PUT/POST/PATCH
- A Redfish Service should return the ETag header on a GET of an individual resource

The format of the ETag header is:

```
ETag W/"<string>"
```

Protocol Version

The protocol version is separate from the version of the resources or the version of the Redfish Schema supported by them.

Each version of the Redfish protocol is strongly typed. This is accomplished using the URI of the Redfish service in combination with the resource obtained at that URI, called the ServiceRoot.

The root URI for this version of the Redfish protocol shall be `"/redfish/v1/"`.

While the major version of the protocol is represented in the URI, the major version, minor version and errata version of the protocol are represented in the Version property of the ServiceRoot resource, as defined in the Redfish Schema for that resource. The protocol version is a string of the form:

MajorVersion.MinorVersion.Errata

where:

- *MajorVersion* = integer: something in the class changed in a backward incompatible way.
- *MinorVersion* = integer: a minor update. New functionality may have been added but nothing removed. Compatibility will be preserved with previous minorversions.
- *Errata* = integer: something in the prior version was broken and needed to be fixed.

Any resource discovered through links found by accessing the root service or any service or resource referenced using references from the root service shall conform to the same version of the protocol supported by the root service.

A GET on the resource `"/redfish"` shall return the following body:

```
{
  "v1": "/redfish/v1/"
}
```

Redfish-Defined URIs and Relative URI Rules

Redfish is a hypermedia API with a small set of defined URIs. All other resources are accessible via opaque URIs referenced from the root service. The following Redfish-defined URIs shall be supported by a Redfish service:

URI	Description
/redfish	The URI that is used to return the version .
/redfish/v1/	The URI for the Redfish Service Root
/redfish/v1/odata	The URI for the Redfish OData Service Document
/redfish/v1/\$metadata	The URI for the Redfish Metadata Document

In addition, the following URI without a trailing slash shall be either Redirected to the Associated Redfish-defined URI shown in the table below or else shall be treated by the service as the equivalent URI to the associated Redfish-defined URI:

URI	Associated Redfish-Defined URI .
/redfish/v1	/redfish/v1/

If a service implementation chooses not to redirect these two APIs and instead treat them as equivalent APIs, then all relative URIs used by the service shall include the full path starting with /redfish/v1/...

Requests

This section describes the requests that can be sent to Redfish services.

Request Headers

HTTP defines headers that can be used in request messages. The following table defines those headers and their requirements for Redfish services.

- Redfish services shall understand and be able to process the headers in the following table as defined by the HTTP 1.1 specification if the value in the Required column is set to "Yes".
- Redfish services shall understand and be able to process the headers in the following table as defined by the HTTP 1.1 specification if the value in the Required column is set to "Conditional" under the conditions noted in the description.
- Redfish services should understand and be able to process the headers in the following tables as defined by the HTTP 1.1 specification if the value in the Required column is set to "No".

Header	Required	Supported Values	Description
Accept	Yes	RFC 2616, Section 14.1	Indicates to the server what media type(s) this client is prepared to accept. <code>application/json</code> shall be supported for requesting resources and <code>application/xml</code> shall be supported for requesting metadata.
Accept-Encoding	Yes	RFC 2616, Section 14.4	Indicates if gzip encoding can be handled by the client
Accept-Language	No	RFC 2616, Section 14.4	This header is used to indicate the language(s) requested in the response. If this header is not specified, the appliance default locale will be used.
Content-Type	Conditional	RFC 2616, Section 14.17	Describes the type of representation used in the message body. <code>charset=utf-8</code> shall be supported for requests that have a body. Shall be required if there is a request body.

Content-Length	No	RFC 2616, Section 14.3	Describes the size of the message body. An optional means of indicating size of the body uses Transfer-Encoding: chunked, which does not use the Content-Length header. If a service does not support Transfer-Encoding and needs Content-Length instead, the service will respond with status code 411 .
Max-OData-Version	No	4.0	Indicates the maximum version of OData that an odata-aware client understands
OData-Version	Yes	4.0	If provided, services shall reject requests which specify an unsupported OData version.
Authorization	Conditional	RFC 2617, Section 2	Required for Basic Authorization
User-Agent	Yes	RFC 2616, Section 14.43	Required for tracing product tokens and their version. Multiple product tokens may be listed.
Host	Yes	RFC 2616, Section 14.23	Required to allow support of multiple origin hosts at a single IP address.
Origin	Yes	W3C CORS, Section 5.7	Used to allow web applications to consume Redfish service while preventing CSRF attacks.
Via	No	RFC 2616, Section 14.45	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.
Max-Forwards	No	RFC 2616, Section 14.31	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
If-Match	Conditional	RFC 2616, Section 14.31	If-Match shall be supported for Atomic requests on AccountService objects. If-Match shall be supported on PUT and PATCH requests for resources for which the service returns ETags.
If-None-Match	No	RFC 2616, Section 14.31	If this HTTP header is present, the service will only return the requested resource if the current ETag of that resource does not match the ETag sent in this header. If the ETag specified in this header matches the resource's current ETag, the status code returned from the GET will be 304 .

- Redfish services shall understand and be able to process the headers in the following table as defined by this specification if the value in the Required column is set to "yes" .

Header	Required	Supported Values	Description
X-Auth-Token	Yes	Opaque encoded octet strings	Used for bearer authentication of user sessions. The token value shall be indistinguishable from random.

Read Requests (GET)

The GET method is used to retrieve a representation of a resource. That representation can either be a single resource

or a collection. The service will return the representation using one of the media types specified in the Accept header, subject to requirements in the Media Types section [Media Types](#). If the Accept header is not present, the service will return the resources representations as application/json.

- The HTTP GET method shall be used to retrieve a resource without causing any side effects.
- The service shall ignore the content of the body on a GET.
- The GET operation shall be idempotent in the absence of outside changes to the resource.

Service Root Request

The root URL for Redfish version 1 services shall be `"/redfish/v1/"`.

The root URL for the service returns a ServiceRoot resource as defined by this specification.

Metadata Document Request

Redfish services shall expose a [metadata document](#) describing the service at the `"/redfish/v1/$metadata"` resource. This metadata document describes the resources and collections available at the root, and references additional metadata documents describing the full set of resource types exposed by the service.

Services shall not require authentication in order to retrieve the metadata document.

OData Service Document Request

Redfish services shall expose an [OData Service Document](#), at the `"/redfish/v1/odata"` resource. This service document provides a standard format for enumerating the resources exposed by the service, enabling generic hypermedia-driven OData clients to navigate to the resources of the service.

Services shall not require authentication in order to retrieve the service document.

Resource Retrieval Requests

Clients request resources by issuing GET requests to the URI for the individual resource or resource collection. The URI for a resource or resource collection may be obtained from a [resource identifier property](#) returned in a previous request (for example, within the [links](#) section of a previously returned resource). Services may, but are not required to, support the convention of retrieving individual properties of a resource by appending a segment containing the property name to the URI of the resource.

Query Parameters

When the resource addressed is a collection, the client can use the following paging query options to specify that a subset of the members be returned. These paging query options apply to the Members property of a collection resource.

Attribute	Description	Example
\$skip	Integer indicating the number of resources in the collection to skip before retrieving the first resource.	<code>http://collection?\$skip=5</code>
\$top	Integer indicating the number of collection members to include in the response. The minimum value for this parameter is 1. The default behavior is to return all members.	<code>http://collection?\$top=30</code>

- Services should support the \$top and \$skip query parameters.
- Implementation shall return the 501, Not Implemented, status code for any query parameters starting with "\$" that are not supported, and should return an [extended error](#) indicating the requested query parameter(s) not supported for this resource.
- Implementations shall ignore unknown or unsupported query parameters that do not begin with "\$".

Retrieving Collections

Retrieving a collection is done by sending the HTTP GET method to the URI for the collection. The response will be a [resource collection representation](#) that includes the collection's attributes as well as the list of the members of the collection. A subset of the members can be returned using [client paging query parameters](#).

No requirements are placed on implementations to return a consistent set of members when a series of requests using paging query parameters are made over time to obtain the entire set of members. It is possible that this could result in missed or duplicate elements being retrieved if multiple GETs are used to retrieve a collection using paging.

- Clients shall not make assumptions about the URIs for the resource members of a collection.
- Retrieved collections shall always include the [count](#) property to specify the total number of members in the collection.
- If only a portion of the collection is returned due to client-specified paging query parameters or services returning [partial results](#), then the total number of resources across all pages shall be returned in the count property.

HEAD

The HEAD method differs from the GET method in that it MUST NOT return message body information. However, all of the same meta information and status codes in the HTTP headers will be returned as though a GET method were processed, including authorization checks.

- Services may support the HEAD method in order to return meta information in the the form of HTTP response headers.
- Services may support the HEAD method in order to verify link validity.
- Services may support the HEAD method in order to verify resource accessibility
- Services shall not support any other use of the HEAD method.
- The HEAD method shall be idempotent in the absence of outside changes to the resource.

Data Modification Requests

Clients create, modify, and delete resources by issuing the appropriate [Create](#), [Update](#), [Replace](#) or [Delete](#) operation, or by invoking an [Action](#) on the resource. Services return a status code [405](#) if the specified resource exists but does not support the requested operation. If a client (4xx) or service (5xx) [status code](#) is returned, the resource shall not be modified as a result of the operation.

Update (PATCH)

The PATCH method is the preferred method used to perform updates on pre-existing resources. Changes to the resource are sent in the request body. Properties not specified in the request body are not directly changed by the PATCH request. The response is either empty or a representation of the resource after the update was done. The implementation may reject the update operation on certain fields based on its own policies and, if so, shall not apply any of the update requested.

- Services shall support the PATCH method to update a resource. If the resource can never be updated, status code [405](#) shall be returned.
- Services may return a representation of the resource after any server-side transformations in the body of the response.
- If a property in the request can never be updated, such as when a property is read only, a status code of [200](#) shall be returned along with a representation of the resource containing an [annotation](#) specifying the non-updatable property. In this success case, other properties may be updated in the resource.
- Services should return status code [405](#) if the client specifies a PATCH request against a collection.
- The PATCH operation should be idempotent in the absence of outside changes to the resource, though the original ETag value may no longer match.

Within a PATCH request, unchanged members within a JSON array may be specified as empty JSON objects.

OData markup ([resource identifiers](#), [type](#), [etag](#) and [links](#)) are ignored on Update.

Replace (PUT)

The PUT method is used to completely replace a resource. Properties omitted from the request body are reset to their default value.

- Services may support the PUT method to replace a resource in whole. If a service does not implement this method, status code 405 shall be returned.
- Services may return a representation of the resource after any server-side transformations in the body of the response.
- Services should return status code 405 if the client specifies a PUT request against a collection.
- The PUT operation should be idempotent in the absence of outside changes to the resource, with the possible exception that ETag values may change as the result of this operation.

Create (POST)

The POST method is used to create a new resource. The POST request is submitted to the resource collection in which the new resource is to belong.

Submitting a POST request to a resource representing a collection is equivalent to submitting the same request to the Members property of that resource. Services that support adding members to a collection shall support both forms.

- Services shall support the POST method for creating resources. If the resource does not offer anything to be created, a status code 405 shall be returned.
- The POST operation shall not be idempotent.

The body of the create request contains a representation of the object to be created. The service can ignore any service controlled attributes (e.g. id), forcing those attributes to be overridden by the service. The service shall set the Location header to the URI of the newly created resource. The response to a successful create request should be 201 (Created) and may include a response body containing the representation of the newly created resource.

Delete (DELETE)

The DELETE method is used to remove a resource.

- Services shall support the DELETE method for resources that can be deleted. If the resource can never be deleted, status code 405 shall be returned.
- Services may return a representation of the just deleted resource in the response body.
- Services should return status code 405 if the client specifies a DELETE request against a collection.

Services may return status code 404 or a success code if the resource has already been deleted.

Actions (POST)

The POST method is used to initiate operations on the object (such as Actions).

- Services shall support the POST method for sending actions.
- The POST operation may not be idempotent.

Custom actions are requested on a resource by sending the HTTP POST method to the URI of the action. If the [actions property](#) within a resource does not specify a target property, then the URI of an action shall be of the form:

ResourceUri/Actions/QualifiedActionName

where

- *ResourceUri* is the URL of the resource which supports invoking the action.
- "Actions" is the name of the property containing the actions for a resource, as defined by this specification.
- *QualifiedActionName* is the namespace or alias qualified name of the action.

The first parameter of a bound function is the resource on which the action is being invoked. The remaining parameters

are represented as name/value pairs in the body of the request.

Clients can query a resource directly to determine the [actions](#) that are available as well as [valid parameter values](#) for those actions. Some parameter information may require the client to examine the Redfish Schema corresponding to the resource.

For instance, if a Redfish Schema document <http://redfish.dmtf.org/schemas/v1/ComputerSystem.xml> defines a Reset action in the `ComputerSystem` namespace, bound to the `ComputerSystem.1.0.0.Actions` type, such as this example:

```
<Schema Name="ComputerSystem">
...
  <Action Name="Reset" IsBound="true">
    <Parameter Name="Resource" Type="ComputerSystem.1.0.0.Actions"/>
    <Parameter Name="ResetType" Type="Resource.ResetType"/>
  </Action>
...
</Schema>
```

And a computer system resource contains an [Actions](#) property such as this:

```
"Actions": {
  "#ComputerSystem.Reset": {
    "target": "/redfish/v1/Systems/1/Actions/ComputerSystem.Reset",
    "ResetType@Redfish.AllowableValues": [
      "On",
      "ForceOff",
      "GracefulRestart",
      "GracefulShutdown",
      "ForceRestart",
      "Nmi",
      "ForceOn",
      "PushPowerButton"
    ]
  }
}
```

Then the following would represent a possible request for the Action:

```
POST /redfish/v1/Systems/1/Actions/ComputerSystem.Reset
{
  "ResetType": "On"
}
```

Responses

Redfish defines four types of responses:

- [Metadata Responses](#) - Describe the resources and types exposed by the service to generic clients.
- [Resource Responses](#) - JSON representation of an individual resource.
- [Resource Collection Responses](#) - JSON representation of a collections of resources.
- [Error Responses](#) - Top level JSON response providing additional information in the case of an HTTP error.

Response Headers

HTTP defines headers that can be used in response messages. The following table defines those headers and their requirements for Redfish services.

- Redfish services shall be able to return the headers in the following table as defined by the HTTP 1.1 specification if

the value in the Required column is set to "yes".

- Redfish services should be able to return the headers in the following tables as defined by the HTTP 1.1 specification if the value in the Required column is set to "no".
- Redfish clients shall be able to understand and be able to process all of the headers in the following table as defined by the HTTP 1.1. specification.

Header	Required	Supported Values	Description
OData-Version	Yes	4.0	Describes the OData version of the payload that the response conforms to.
Content-Type	Yes	RFC 2616, Section 14.17	Describes the type of representation used in the message body. <code>application/json</code> shall be supported. <code>charset=utf-8</code> shall be supported.
Content-Encoding	No	RFC 2616, Section 14.17	Describes the encoding that has been performed on the media type
Content-Length	No	RFC 2616, Section 14.3	Describes the size of the message body. An optional means of indicating size of the body uses Transfer-Encoding: chunked, which does not use the Content-Length header. If a service does not support Transfer-Encoding and needs Content-Length instead, the service will respond with status code 411 .
ETag	Conditional	RFC 2616, Section 14.19	An identifier for a specific version of a resource, often a message digest. Etags shall be included on Account objects.
Server	Yes	RFC 2616, Section 14.38	Required to describe a product token and its version. Multiple product tokens may be listed.
Link	Yes	See Link Header	Link headers shall be returned as described in the section on Link Headers .
Location	Conditional	RFC 2616, Section 14.30	Indicates a URI that can be used to request a representation of the resource. Shall be returned if a new resource was created. Location and X-Auth-Token shall be included on responses which create user sessions.
Cache-Control	Yes	RFC 2616, Section 14.9	This header shall be supported and is meant to indicate whether a response can be cached or not.
Via	No	RFC 2616, Section 14.45	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.
Max-Forwards	No	RFC 2616, Section 14.31	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
Link	No	RFC 5988, Section 5	Exposes additional metadata about response object. See Link Header .
Access-		W3C	

Control-Allow-Origin	Yes	CORS, Section 5.1	Prevents or allows requests based on originating domain. Used to prevent CSRF attacks.
Allow	Yes	POST, PUT, PATCH, DELETE	Returned on GET or HEAD operation to indicate the other allowable operations for this resource. Shall be returned with a 405 (Method Not Allowed) response to indicate the valid methods for the specified Request URI.
WWW-Authenticate	Yes	RFC 2617	Required for Basic and other optional authentication mechanisms. See the [Security] [#Security] section for details.

- Redfish services shall understand and be able to process the headers in the following table as defined by this specification if the value in the Required column is set to "yes".

Header	Required	Supported Values	Description
X-Auth-Token	Yes	Opaque encoded octet strings	Used for bearer authentication of user sessions. The token value shall be indistinguishable from random.

Link Header

The [Link header](#) provides metadata information on the accessed resource in response to a HEAD or GET operation. In addition to links from the resource, the URL of the JSON schema of the resource shall be returned with a `rel=describedby`.

Link header(s) shall be returned on HEAD and a Link header satisfying `rel=describedby` shall be returned on GET and HEAD.

Status Codes

HTTP defines status codes that can be returned in response messages.

Where the HTTP status code indicates a failure, the response body contains an [extended error resource](#) to provide the client more meaningful and deterministic error semantics.

- Services shall return the extended error resource as described in this specification in the response body when a status code of 400 or 500 is returned.
- Services should return the extended error resource as described in this specification in the response body when a status code 400 or greater is returned.
- Extended error messages MUST NOT provide privileged info when authentication failures occur

NOTE: Refer to the [Security](#) section for security implications of extended errors

The following table lists some of the common HTTP status codes. Other codes may be returned by the service as appropriate. See the Description column for a description of the status code and additional requirements imposed by this specification.

- Clients shall understand and be able to process the status codes in the following table as defined by the HTTP 1.1 specification and constrained by additional requirements defined by this specification.
- Services shall respond with these status codes as appropriate.
- Exceptions from operations shall be mapped to HTTP status codes.
- Redfish services should not return the status code 100. Using the HTTP protocol for a multi-pass data transfer should be avoided, except upload of extremely large data.

HTTP Status	
--------------------	--

Code	Description
200 OK	The request was successfully completed and includes a representation in its body.
201 Created	A request that created a new resource completed successfully. The Location header shall be set to the canonical URI for the newly created resource. A representation of the newly created resource may be included in the response body.
202 Accepted	The request has been accepted for processing, but the processing has not been completed. The Location header shall be set to the URI of a Task resource that can later be queried to determine the status of the operation. A representation of the Task resource may be included in the response body.
204 No Content	The request succeeded, but no content is being returned in the body of the response.
301 Moved Permanently	The requested resource resides under a different URI
302 Found	The requested resource resides temporarily under a different URI.
304 Not Modified	The service has performed a conditional GET request where access is allowed, but the resource content has not changed. Conditional requests are initiated using the headers If-Modified-Since and/or If-None-Match (see HTTP 1.1, sections 14.25 and 14.26) to save network bandwidth if there is no change.
400 Bad Request	The request could not be processed because it contains missing or invalid information (such as validation error on an input field, a missing required value, and so on). An extended error shall be returned in the response body, as defined in section Extended Error Handling .
401 Unauthorized	The authentication credentials included with this request are missing or invalid.
403 Forbidden	The server recognized the credentials in the request, but those credentials do not possess authorization to perform this request.
404 Not Found	The request specified a URI of a resource that does not exist.
405 Method Not Allowed	The HTTP verb specified in the request (e.g. DELETE, GET, HEAD, POST, PUT, PATCH) is not supported for this request URI. The response shall include an Allow header which provides a list of methods that are supported by the resource identified by the Request-URI.
406 Not Acceptable	The Accept header was specified in the request and the resource identified by this request is not capable of generating a representation corresponding to one of the media types in the Accept header.
409 Conflict	A creation or update request could not be completed, because it would cause a conflict in the current state of the resources supported by the platform (for example, an attempt to set multiple attributes that work in a linked manner using incompatible values).
410 Gone	The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. Clients with link editing capabilities SHOULD delete references to the Request-URI after user approval. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) SHOULD be used instead. This response is cacheable unless indicated otherwise.
411 Length Required	The request did not specify the length of its content using the Content-Length header (perhaps Transfer-Encoding: chunked was used instead). The addressed resource requires the Content-Length header.

412 Precondition Failed	Precondition (If Match or If Not Modified) check failed.
415 Unsupported Media Type	The request specifies a Content-Type for the body that is not supported.
500 Internal Server Error	The server encountered an unexpected condition that prevented it from fulfilling the request. An extended error shall be returned in the response body, as defined in section Extended Error Handling .
501 Not Implemented	The server does not (currently) support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting the method for any resource.
503 Service Unavailable	The server is currently unable to handle the request due to temporary overloading or maintenance of the server.

Metadata Responses

Metadata describes resources, collections, capabilities and service-dependent behavior to generic consumers, including OData client tools and applications with no specific understanding of this specification. Clients are not required to request metadata if they already have sufficient understanding of the target service; for example, to request and interpret a JSON representation of a resource defined in this specification.

Service Metadata

The service metadata describes top-level resources and resource types of the service according to [OData-Schema](#). The Redfish Service Metadata is represented as an XML document with a root element named "Edmx", defined in the <http://docs.oasis-open.org/odata/ns/edmx> namespace, and with an OData Version attribute equal to "4.0".

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Version="4.0">
  <!-- edmx:Reference and edmx:Schema elements go here -->
</edmx:Edmx>
```

Referencing Other Schemas

The service metadata shall include the namespaces for each of the Redfish resource types, along with the "RedfishExtensions.1.0.0" namespace. These references may use the standard Uri for the hosted Redfish Schema definitions (i.e., on <http://redfish.dmtf.org/schema>) or a Url to a local version of the Redfish Schema that shall be identical to the hosted version.

```
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/AccountService.xml">
  <edmx:Include Namespace="AccountService"/>
  <edmx:Include Namespace="AccountService.1.0.0"/>
</edmx:Reference>
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/ServiceRoot.xml">
  <edmx:Include Namespace="ServiceRoot"/>
  <edmx:Include Namespace="ServiceRoot.1.0.0"/>
</edmx:Reference>

...
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/VirtualMedia.xml">
  <edmx:Include Namespace="VirtualMedia"/>
  <edmx:Include Namespace="VirtualMedia.1.0.0"/>
</edmx:Reference>
```

```
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/RedfishExtensions.xml">
  <edmx:Include Namespace="RedfishExtensions.1.0.0" Alias="Redfish"/>
</edmx:Reference>
```

The service metadata shall include an entity container that defines the top level resource and collections. This entity container shall extend the ServiceContainer defined in the ServiceRoot.<%= DocVersion %> schema and may include additional resources or collections.

```
<edmx:DataServices>
  <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Service">
    <EntityContainer Name="Service" Extends="ServiceRoot.1.0.0.ServiceContainer"/>
  </Schema>
</edmx:DataServices>
```

Referencing OEM Extensions

The metadata document may reference additional schema documents describing OEM-specific extensions used by the service, for example custom types for additional collections.

```
<edmx:Reference Uri="http://contoso.org/Schema/CustomTypes">
  <edmx:Include Namespace="CustomTypes"/>
</edmx:Reference>
```

Annotations

The service can annotate sets, types, actions and parameters with Redfish-defined or custom annotation terms. These annotations are typically in a separate Annotations file referenced from the service metadata document using the IncludeAnnotations directive.

```
<edmx:Reference Uri="http://service/metadata/Service.Annotations">
  <edmx:IncludeAnnotations TermNamespace="Annotations.1.0.0"/>
</edmx:Reference>
```

The annotation file itself specifies the Target Redfish Schema element being annotated, the Term being applied, and the value of the term:

```
<Annotations Target="ComputerSystem.Reset/ResetType">
  <Annotation Term="Annotation.AdditionalValues">
    <Collection>
      <String>Update and Restart</String>
      <String>Update and PowerOff</String>
    </Collection>
  </Annotation>
</Annotations>
```

OData Service Document

The OData Service Document serves as a top-level entry point for generic OData clients.

```
{
  "@odata.context": "/redfish/v1/$metadata",
  "value": [
    {
      "name": "Service",
      "kind": "Singleton",
      "url": "/redfish/v1/"
    },
  ],
}
```

```

{
  "name": "Systems",
  "kind": "Singleton",
  "url": "/redfish/v1/Systems"
},
{
  "name": "Chassis",
  "kind": "Singleton",
  "url": "/redfish/v1/Chassis"
},
{
  "name": "Managers",
  "kind": "Singleton",
  "url": "/redfish/v1/Managers"
},
...
]
}

```

The OData Service Document shall be returned as a JSON object, using the MIME type `application/json`.

The JSON object shall contain a context property named "@odata.context" with a value of `/redfish/v1/$metadata`. This context tells a generic OData client how to find the [service metadata](#) describing the types exposed by the service.

The JSON object shall include a property named "value" whose value is a JSON array containing an entry for the [service root](#) and each resource that is a direct child of the service root.

Each entry shall be represented as a JSON object and shall include a "name" property whose value is a user-friendly name of the resource, a "kind" property whose value is "Singleton" for individual resources (including collection resources) or "EntitySet" for top-level resource collections, and a "url" property whose value is the relative URL for the top-level resource.

Resource Responses

Resources are returned as JSON payloads, using the MIME type `application/json`.

Context Property

Responses that represent a single resource shall contain a context property named "@odata.context" describing the source of the payload. The value of the context property shall be the context URL that describes the resource according to [OData-Protocol](#).

The context URL for a resource that exists within a collection is of the form:

MetadataUrl#Collection[(Selectlist)]/\$entity

Where:

- *MetadataUrl* = the metadata url of the service (`/redfish/v1/$metadata`)
- *Collection* = the collection resource. For contained resources this includes the path from the root collection or singleton resource to the containment property.
- *Selectlist* = comma-separated [list of properties](#) included in the response if the response includes a subset of properties defined for the represented resources.

The context URL for a resource that is a top-level singleton resource is of the form:

MetadataUrl#SingletonName[(Selectlist)]

Where:

- *MetadataUrl* = the metadata url of the service (`/redfish/v1/$metadata`)

- *SingletonName* = the name of the top-level singleton resource
- *Selectlist* = comma-separated [list of properties](#) included in the response if the response includes a subset of properties defined for the represented resources.

Select List

If a response contains a subset of the properties defined in the Redfish Schema for a type, then the context URL shall specify the subset of properties included. An asterisk (*) can be used to specify "all structural properties" for a given resource.

Expanded [reference properties](#) shall be included in the select list if the result includes a subset of the properties defined for the expanded resource.

For example, the following context URL specifies that the result contains a single resource from the Members collection nested under the Links property of the Systems resource:

```
"@odata.context":"/redfish/v1/$metadata#Systems/Members/$entity",
```

Resource Identifier Property

Resources in a response shall include a unique identifier property named "@odata.id". The value of the identifier property shall be the [unique identifier](#) for the resource.

Resource Identifiers shall be represented in JSON payloads as uri paths relative to the Redfish Schema portion of the uri. That is, they shall always start with "/redfish/".

The resource identifier is the canonical URL for the resource and can be used to retrieve or edit the resource, as appropriate.

Type Property

All resources in a response shall include a type property named "@odata.type". The value of the type property shall be an absolute URL that specifies the type of the resource and shall be of the form:

#Namespace.TypeName

Where:

- *Namespace* = The full namespace name of the Redfish Schema in which the type is defined. For Redfish resources this will be the versioned namespace name.
- *TypeName* = The name of the type of the resource.

The client may issue a GET request to this URL using a content type of `application/xml` in order to retrieve a document containing the [definition of the resource](#).

ETag Property

ETags provide the ability to conditionally retrieve or update a resource. Resources should include an ETag property named "@odata.etag". The value of the ETag property is the [Etag](#) for a resource.

Primitive Properties

Primitive properties shall be returned as JSON values according to the following table.

Type	JSON Representation
Edm.Boolean	Boolean
Edm.DateTimeOffset	String, formatted as specified in DateTime Values
Edm.Decimal	Number, optionally containing a decimal point

Edm.Type	Number, optionally containing a decimal point
Edm.Double	Number, optionally containing a decimal point and optionally containing an exponent
Edm.Guid	String, matching the pattern ([0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})
Edm.Int64	Number with no decimal point
Edm.String	String

When receiving values from the client, services should support other valid representations of the data within the specified JSON type. In particular, services should support valid integer and decimal values written in exponential notation and integer values containing a decimal point with no non-zero trailing digits.

DateTime Values

DateTime values shall be returned as JSON strings according to the ISO 8601 "extended" format, with time offset or UTC suffix included, of the form:

YYYY-MM-DD T hh:mm:ss.SSS

where:

- SSS = one or more digits representing a decimal fraction of a second, with the number of digits implying precision.
- The 'T' separator and 'Z' suffix shall be capitals.

Structured Properties

Structured properties, defined as [complex types](#) or [expanded resource types](#), are returned as JSON objects. The type of the JSON object is specified in the Redfish Schema definition of the property containing the structured value.

Collection Properties

Collection-valued properties are returned as JSON arrays, where each element of the array is a JSON object whose type is specified in the Redfish Schema document describing the containing type.

Collection-valued properties may contain a subset of the members of the full collection. In this case, the collection-valued property shall be annotated with a next link property. The property representing the next link shall be a peer of the collection-valued property, with the name of the collection-valued property suffixed with "@odata.nextLink". The value of the next link property shall be an opaque URL that the client can use to retrieve the next set of collection members. The next link property shall only be present if the number of resources requested is greater than the number of resources returned.

Collection-valued properties shall be annotated with a count. The property representing the count is a peer of the collection-valued property, with the name of the collection-valued property suffixed with "@odata.count". The value of the count is the total number of members available in the collection.

Collection-valued properties shall not be null. Empty collections shall be returned in JSON as an empty array.

Actions Property

Available actions for a resource are represented as individual properties nested under a single structured property on the resource named "Actions".

Action Representation

Actions are represented by a property nested under "Actions" whose name is the unique URI that identifies the action. This URI shall be of the form:

#Namespace.ActionName

Where:

- *Namespace* = The namespace used in the reference to the Redfish Schema in which the action is defined. For Redfish resources this shall be the version-independent namespace.
- *ActionName* = The name of the action

The client may use this fragment to identify the [action definition](#) within the [referenced](#) Redfish Schema document associated with the specified namespace.

The value of the property is a JSON object containing a property named "target" whose value is a relative or absolute URL used to invoke the action.

The property representing the available action may be annotated with the [AllowableValues](#) annotation in order to specify the list of allowable values for a particular parameter.

For example, the following property represents the Reset action, defined in the ComputerSystem namespace:

```
"#ComputerSystem.Reset": {
  "target": "/redfish/v1/Systems/1/Actions/ComputerSystem.Reset",
  "ResetType@Redfish.AllowableValues": [
    "On",
    "ForceOff",
    "GracefulRestart",
    "GracefulShutdown",
    "ForceRestart",
    "Nmi",
    "ForceOn",
    "PushPowerButton"
  ]
}
```

Given this, the client could invoke a POST request to `/redfish/v1/Systems/1/Actions/ComputerSystem.Reset` with the following body:

```
{
  "ResetType": "On"
}
```

Allowable Values

The property representing the action may be annotated with the "AllowableValues" annotation in order to specify the list of allowable values for a particular parameter.

The set of allowable values is specified by including a property whose name is the name of the parameter followed by "@Redfish.AllowableValues", and whose value is a JSON array of strings representing the allowable values for the parameter.

Links Property

[References](#) to other resources are represented by the links property on the resource.

The links property shall be named "Links" and shall contain a property for each [non-contained reference property](#) defined in the Redfish Schema for that type. For single-valued reference properties, the value of the property shall be the [single related resource id](#). For collection-valued reference properties, the value of the property shall be the [array of related resource ids](#).

The links property shall also include an [Oem property](#) for navigating vendor-specific links.

Reference to a Single Related Resource

A reference to a single resource is returned as a JSON object containing a single [resource-identifier-property](#) whose

name is the name of the relationship and whose value is the uri of the referenced resource.

```
{
  "Links" : {
    "ManagedBy": {
      "@odata.id": "/redfish/v1/Chassis/Enc11"
    }
  }
}
```

Array of References to Related Resources

A reference to a collection of zero or more related resources is returned as an array of JSON objects whose name is the name of the relationship. Each member of the array is a JSON object containing a single [resource-identifier-property](#) whose value is the uri of the referenced resource.

```
{
  "Links" : {
    "Contains" : [
      {
        "@odata.id": "/redfish/v1/Chassis/1"
      },
      {
        "@odata.id": "/redfish/v1/Chassis/Enc11"
      }
    ]
  }
}
```

OEM Property

OEM-specific properties are nested under an OEM property.

Extended Information

Response objects may include extended information, for example information about properties that are not able to be updated. This information is represented as an annotation applied to a [specific property](#) of the JSON response or an entire JSON object.

Extended Object Information

A JSON object can be annotated with "@Message.ExtendedInfo" in order to specify object-level status information.

```
{
  "@odata.context": "/redfish/v1/$metadata#Managers/Members/1/SerialInterfaces/Members/$entity",
  "@odata.id": "/redfish/v1/Managers/1/SerialInterfaces/1",
  "@odata.type": "#SerialInterface.1.0.0.SerialInterface",
  "Name": "Managed Serial Interface 1",
  "Description": "Management for Serial Interface",
  "Status": {
    "State": "Enabled",
    "Health": "OK"
  },
  "InterfaceEnabled": true,
  "SignalType": "Rs232",
  "BitRate": 115200,
  "Parity": "None",
  "DataBits": 8,
  "StopBits": 1,
  "FlowControl": "None",
  "ConnectorType": "RJ45",
  "PinOut": "Cyclades"
  "@Message.ExtendedInfo" : {
```

```

    "MessageId": "Base.1.0.PropertyDuplicate",
    "Message": "The property InterfaceEnabled was duplicated in the request.",
    "RelatedProperties": [
        "#/InterfaceEnabled"
    ],
    "Severity": "Warning",
    "Resolution": "Remove the duplicate property from the request body and resubmit the request if the opera
}
}

```

The value of the property is an array of message objects.

Extended Property Information

An individual property within a JSON object can be annotated with extended information using "@Message.ExtendedInfo", prepended with the name of the property.

```

{
    "@odata.context": "/redfish/v1/$metadata#Managers/Members/1/SerialInterfaces/Members/$entity",
    "@odata.id": "/redfish/v1/Managers/1/SerialInterfaces/1",
    "@odata.type": "#SerialInterface.1.0.0.SerialInterface",
    "Name": "Managed Serial Interface 1",
    "Description": "Management for Serial Interface",
    "Status": {
        "State": "Enabled",
        "Health": "OK"
    },
    "InterfaceEnabled": true,
    "SignalType": "Rs232",
    "BitRate": 115200,
    "Parity": "None",
    "DataBits": 8,
    "StopBits": 1,
    "FlowControl": "None",
    "ConnectorType": "RJ45",
    "PinOut": "Cyclades"
    "PinOut@Message.ExtendedInfo" : [
        {
            "MessageId": "Base.1.0.PropertyValueNotInList",
            "Message": "The value Cyclades for the property PinOut is not in the list of acceptable values.",
            "Severity": "Warning",
            "Resolution": "Choose a value from the enumeration list that the implementation can support and resubm
        }
    ],
    "Oem": {}
}

```

The value of the property is a message object.

Additional Annotations

A resource representation in JSON may include additional annotations represented as properties whose name is of the form:

[PropertyName]@Namespace.TermName

where

- *PropertyName* = the name of the property being annotated. If omitted, the annotation applies to the entire resource.
- *Namespace* = the name of the namespace where the annotation term is defined. This namespace must be referenced by the [metadata document](#) specified in the [context url](#) of the request.

- *TermName* = the name of the annotation term being applied to the resource or property of the resource.

The client can get the definition of the annotation from the [service metadata](#), or may ignore the annotation entirely, but should not fail reading the resource due to unrecognized annotations, including new annotations defined within the Redfish namespace.

Resource Collections

Resource collections are returned as a JSON object. The JSON object shall include a [context](#), [resource count](#), and array of [values](#), and may include a [next link](#) for partial results.

Context Property

Responses that represent a collection of resources shall contain a context property named "@odata.context" describing the source of the payload. The value of the context property shall be the context URL that describes the resources according to [OData-Protocol](#).

The context URL for a resource collection is of the form:

MetadataUrl.#Collection[(SelectList)]

Where:

- *MetadataUrl* = the metadata url of the service (/redfish/v1/\$metadata)
- *Collection* = the collection resource. For contained resources this includes the path from the root collection or singleton resource to the containment property.
- *SelectList* = comma-separated [list of properties](#) included in the response if the response includes a subset of properties defined for the represented resources.

Resource Count Property

The total number of resources available in the collection is represented through the count property. The count property shall be named "@odata.count" and its value shall be an integer representing the total number of records in the result. This count is not affected by the \$top or \$skip [query parameters](#).

Resource Members Property

The members of the collection of resources are returned as a JSON array. The name of the property representing the members of the collection shall be "value".

Partial Results

Responses representing a single resource shall not be broken into multiple results.

Collections of resources, or resource ids, may be returned in multiple partial responses. For partial collections the service includes a next link property named "@odata.nextLink". The value of the next link property shall be an opaque URL that the client can use to retrieve the next set of resources. The next link shall only be returned if the number of resources requested is greater than the number of resources returned.

The value of the [count property](#) represents the total number of resources available if the client enumerates all pages of the collection.

Additional Annotations

A JSON object representing a collection of resources may include additional annotations represented as properties whose name is of the form:

@Namespace.TermName

where

- *Namespace* = the name of the namespace where the annotation term is defined. This namespace shall be referenced by the *metadata document* specified in the *context url* of the request.
- *TermName* = the name of the annotation term being applied to the resource collection.

The client can get the definition of the annotation from the *service metadata*, or may ignore the annotation entirely, but should not fail reading the response due to unrecognized annotations, including new annotations defined within the Redfish namespace.

Error Responses

HTTP response status codes alone often do not provide enough information to enable deterministic error semantics. For example, if a client does a PATCH and some of the properties do not match while others are not supported, simply returning an HTTP status code of 400 does not tell the client which values were in error. Error responses provide the client more meaningful and deterministic error semantics.

Error responses are defined by an extended error resource, represented as a single JSON object with a property named "error" with the following properties.

Property	Description
code	A string indicating a specific MessageId from the message registry. "Base.1.0.GeneralError" should be used only if there is no better message.
message	A human readable error message corresponding to the message in the message registry.
@Message.ExtendedInfo	An array of message objects describing one or more error message(s).

```
{
  "error": {
    "code": "Base.1.0.GeneralError",
    "message": "A general error has occurred. See ExtendedInfo for more information.",
    "@Message.ExtendedInfo": [
      {
        "@odata.type": "/redfish/v1/$metadata#Message.1.0.0.Message",
        "MessageId": "Base.1.0.PropertyValueNotInList",
        "RelatedProperties": [
          "#/IndicatorLED"
        ],
        "Message": "The value Red for the property IndicatorLED is not in the list of acceptable values",
        "MessageArgs": [
          "RED",
          "IndicatorLED"
        ],
        "Severity": "Warning",
        "Resolution": "Remove the property from the request body and resubmit the request if the operatic
      },
      {
        "@odata.type": "/redfish/v1/$metadata#Message.1.0.0.Message",
        "MessageId": "Base.1.0.PropertyNotWriteable",
        "RelatedProperties": [
          "#/SKU"
        ],
        "Message": "The property SKU is a read only property and cannot be assigned a value",
        "MessageArgs": [
          "SKU"
        ],
        "Severity": "Warning",
        "Resolution": "Remove the property from the request body and resubmit the request if the operatic
      }
    ]
  }
}
```

```
}  
}
```

Message Object

Message Objects provide additional information about an [object](#), [property](#), or [error response](#).

Messages are represented as a JSON object with the following properties:

Property	Description
MessageId	String indicating a specific error or message (not to be confused with the HTTP status code). This code can be used to access a detailed message from a message registry.
Message	A human readable error message indicating the semantics associated with the error. This shall be the complete message, and not rely on substitution variables.
RelatedProperties	An optional array of JSON Pointers defining the specific properties within a JSON payload described by the message.
MessageArgs	An optional array of strings representing the substitution parameter values for the message. This shall be included in the response if a MessageId is specified for a parameterized message.
Severity	An optional string representing the severity of the error.
Resolution	An optional string describing recommended action(s) to take to resolve the error.

Each instance of a Message object shall contain at least a MessageId, together with any applicable MessageArgs, or a Message property specifying the complete human-readable error message.

MessageIds identify specific messages defined in a message registry.

The value of the MessageId property shall be of the form

RegistryName.MajorVersion.MinorVersion.MessageKey

where

- *RegistryName* is the name of the registry. The registry name shall be Pascal-cased.
- *MajorVersion* is a positive integer representing the major version of the registry
- *MinorVersion* is a positive integer representing the minor version of the registry
- *MessageKey* is a human-readable key into the registry. The message key shall be Pascal-cased and shall not include spaces, periods or special chars.

The client can use the MessageId to search the message registry for the corresponding message.

The message registry approach has advantages for internationalization (since the registry can be translated easily) and light weight implementation (since large strings need not be included with the implementation).

Data Model & Schema

One of the key tenants of the Redfish interface is the separation of protocol and data model. This section describes common data model, resource, and Redfish Schema requirements.

- Each resource shall be strongly typed according to a [resource type definition](#). The type shall be defined in a Redfish [schema document](#) and identified by a unique [type identifier](#).

Type Identifiers

Types are identified by a *Type URI*. The URI for a type is of the form:

#Namespace.TypeName

where:

- *Namespace* = the name of the namespace in which the type is defined
- *TypeName* = the name of the type

The namespace for types defined by this specification is of the form:

ResourceTypeName.MajorVersion.MinorVersion.Errata

where

- *ResourceTypeName* = the name of the resource type. For [structured \(complex\) types](#), [enumerations](#), and [actions](#), this is generally the name of the containing resource type.
- *MajorVersion* = integer: something in the class changed in a backward incompatible way.
- *MinorVersion* = integer: a minor update. New properties may have been added but nothing removed. Compatibility will be preserved with previous minorversions.
- *Errata* = integer: something in the prior version was broken and needed to be fixed.

An example of a valid type namespace might be "ComputerSystem.1.0.0".

Type Identifiers in JSON

Types used within a JSON payload shall be defined in, or referenced, by the [service metadata](#).

Resource types defined by this specification shall be referenced in JSON documents using the full (versioned) namespace name.

NOTE: Refer to the [Security](#) section for security implications of Data Model & Schema

Common Naming Conventions

The Redfish interface is intended to be easily readable and intuitive. Thus, consistency helps the consumer who is unfamiliar with a newly discovered property understand its use. While this is no substitute for the normative information in the Redfish Specification and Redfish Schema, the following rules help with readability and client usage.

Resource Name, Property Names, and constants such as Enumerations shall be Pascal-cased

- The first letter of each word shall be upper case with spaces between words shall be removed (eg PowerState, SerialNumber.)
- No underscores are used.
- Both characters are capitalized for two-character acronyms (eg IPAddress, RemoteIP)
- Only the first character of acronyms with three or more characters is capitalized, except the first word of a Pascal-cased identifier (eg Wwn, VirtualWwn)

Exceptions are allowed for the following cases:

- Well-known technology names like "iSCSI"
- Product names like "iLO"
- Well-known abbreviations or acronyms

For attributes that have units, or other special meaning, the unit identifier should be appended to the name. The current list includes:

- Bandwidth (Mbps), (eg PortSpeedMbps)
- CPU speed (Mhz), (eg ProcessorSpeedMhz)
- Memory size (MegaBytes, MB), (eg MemoryMB)
- Counts of items (Count), (eg ProcessorCount, FanCount)
- The State of a resource (State) (eg PowerState.)
- State values where "work" is being done end in (ing) (eg Applying, Clearing)

Localization Considerations

Localization and translation of data or meta data is outside of the scope of version 1.0 of the Redfish Specification. Property names are never localized.

Schema Definition

Individual resources and their dependent types and actions are defined within a Redfish [schema document](#).

Common Annotations

All Redfish types and properties shall include [description](#) and [long description](#) annotations.

Description

The Description annotation can be applied to any type, property, action or parameter in order to provide a human-readable description of the Redfish Schema element.

The `Description` annotation is defined in <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Core.V1.xml>.

Long Description

The LongDescription annotation term can be applied to any type, property, action or parameter in order to provide a formal, normative specification of the schema element.

The `LongDescription` annotation term is defined in <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Core.V1.xml>.

Schema Documents

Individual resources are defined as entity types within an OData Schema representation of the Redfish Schema according to [OData-Schema](#). The representation may include annotations to facilitate automatic generation of JSON Schema representation of the Redfish Schema capable of validating JSON payloads.

The outer element of the OData Schema representation document shall be the `Edmx` element, and shall have a `Version` attribute with a value of "4.0".

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Version="4.0">
  <!-- edmx:Reference and edmx:DataService elements go here -->
</edmx:Edmx>
```

Referencing other Schemas

Redfish Schemas may reference types defined in other schema documents. In the OData Schema representation, this is done by including a `Reference` element. In the JSON Schema representation, this is done with a `$ref` property.

The reference element specifies the `uri` of the OData schema representation document describing the referenced type and has one or more child `Include` elements that specify the `Namespace` attribute containing the types to be referenced, along with an optional `Alias` attribute for that namespace.

Type definitions generally reference the OData and Redfish namespaces for common type annotation terms, and resource type definitions reference the Redfish Resource. <%= DocVersion %> namespace for base types. Redfish OData Schema representations that include measures such as temperature, speed, or dimensions generally include the [OData Measures namespace](#).

```
<edmx:Reference Uri="http://docs.oasis-open.org/odata/odata/v4.0/cs01/vocabularies/Org.OData.Core.V1.xml">
  <edmx:Include Namespace="Org.OData.Core.V1" Alias="OData"/>
</edmx:Reference>
<edmx:Reference
  Uri="http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml">
  <edmx:Include Namespace="Org.OData.Measures.V1" Alias="OData.Measures"/>
</edmx:Reference>
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/RedfishExtensions.xml">
  <edmx:Include Namespace="RedfishExtensions.1.0.0" Alias="Redfish"/>
</edmx:Reference>
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/Resource.xml">
  <edmx:Include Namespace="Resource"/>
  <edmx:Include Namespace="Resource.1.0.0"/>
</edmx:Reference>
```

Namespace Definitions

Resource types are defined within a namespace in the OData Schema representations. The namespace is defined through a `Schema` element that contains attributes for declaring the `Namespace` and local `Alias` for the schema.

The OData Schema element is a child of the `DataService` element, which is a child of the `Edmx` element.

```
<edmx:DataService>
  <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="MyTypes.1.0.0">

    <!-- Type definitions go here -->

  </Schema>
</edmx:DataService>
```

Resource Type Definitions

Resource types are defined within a namespace using `EntityType` elements. The `Name` attribute specifies the name of the resource and the `BaseType` specifies the base type, if any.

Redfish resources derive from a common Resource base type named "Resource" in the Resource.1.0.0 namespace.

The `EntityType` contains the [property](#) and [reference property](#) elements that define the resource, as well as annotations describing the resource.

```
<EntityType Name="TypeA" BaseType="Resource.Resource">
  <Annotation Term="Core.Description" String="This is the description of TypeA."/>
  <Annotation Term="Core.LongDescription" String="This is the specification of TypeA."/>

  <!-- Property and Reference Property definitions go here -->

</EntityType>
```

All resources shall include [Description](#) and [LongDescription](#) annotations.

Resource Properties

Structural properties of the resource are defined using the `Property` element. The `Name` attribute specifies the name of

the property, and the `Type` its type.

Properties that must have a non-nullable value include the `nullable attribute` with a value of "false".

```
<Property Name="Property1" Type="Edm.String" Nullable="false">
  <Annotation Term="Core.Description" String="This is a property of TypeA."/>
  <Annotation Term="Core.LongDescription" String="This is the specification of Property1."/>
  <Annotation Term="OData.Permissions" EnumMember="OData.Permissions/Read"/>
  <Annotation Term="Redfish.Required"/>
  <Annotation Term="OData.Measures.Unit" String="Watts"/>
</Property>
```

All properties shall include `Description` and `LongDescription` annotations.

Properties that are read-only are annotated with the `Permissions` annotation with a value of `ODataPermissions/Read`.

Properties that are required to be implemented by all services are annotated with the `required` annotation.

Properties that have units associated with them can be annotated with the `units` annotation

Property Types

Type type of a property is specified by the `Type` attribute. The value of the type attribute may be a `primitive type`, a `structured type`, an `enumeration type` or a `collection` of primitive, structured or enumeration types.

Primitive Types

Primitive types are prefixed with the "Edm" namespace prefix.

Redfish services may use any of the following primitive types:

Type	Meaning
Edm.Boolean	True or False
Edm.DateTimeOffset	Date and time with a time-zone
Edm.Decimal	Numeric values with fixed precision and scale
Edm.Double	IEEE 754 binary64 floating-point number (15-17 decimal digits)
Edm.Guid	A globally unique identifier
Edm.Int64	Signed 64-bit integer
Edm.String	Sequence of UTF-8 characters

Structured Types

Structured types are defined within a `namespace` using `ComplexType` elements. The `Name` attribute of the complex type specifies the name of the structured type. Complex types can include a `BaseType` attribute to specifies the base type, if any.

Structured types may be reused across different properties of different resource types.

```
<ComplexType Name="PropertyTypeA">
  <Annotation Term="Core.Description" String="This is type used to describe a structured property."/>
  <Annotation Term="Core.LongDescription" String="This is the specification of the type."/>

  <!-- Property and Reference Property definitions go here -->

</ComplexType>
```

Structured types can contain [properties](#), [reference properties](#) and annotations.

Structured types shall include [Description](#) and [LongDescription](#) annotations.

Enums

Enumeration types are defined within a [namespace](#) using `EnumType` elements. The `Name` attribute of the enumeration type specifies the name of the enumeration type.

Enumeration types may be reused across different properties of different resource types.

`EnumType` elements contain `Member` elements that define the members of the enumeration. The `Member` elements contain a `Name` attribute that specifies the string value of the member name.

```
<EnumType Name="EnumTypeA">
  <Annotation Term="Core.Description" String="This is the EnumTypeA enumeration."/>
  <Annotation Term="Core.LongDescription" String="This is used to describe the EnumTypeA enumeration."/>
  <Member Name="MemberA">
    <Annotation Term="Core.Description" String="Description of MemberA"/>
  </Member>
  <Member Name="MemberB">
    <Annotation Term="Core.Description" String="Description of MemberB"/>
  </Member>
</EnumType>
```

Enumeration Types shall include [Description](#) and [LongDescription](#) annotations.

Enumeration Members shall include [Description](#) annotations.

Collections

The `type` attribute may specify a collection of [primitive](#), [structured](#) or [enumeration](#) types.

The value of the `type` attribute for a collection-valued property is of the form:

`Collection(NamespaceQualifiedTypeName)`

where *NamespaceQualifiedTypeName* is the namespace qualified name of the primitive, structured, or enumeration type.

Additional Properties

The `AdditionalProperties` annotation term is used to specify whether a type can contain additional properties outside of those defined. Types annotated with the `AdditionalProperties` annotation with a `Boolean` attribute with a value of `"False"`, must not contain additional properties.

```
<Annotation Term="OData.AdditionalProperties"/>
```

The `AdditionalProperties` annotation term is defined in <https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Core.V1.xml>.

Non-Nullable properties

Properties may include the `Nullable` attribute with a value of `false` to specify that the property cannot contain null values. A property with a nullable attribute with a value of `"true"`, or no nullable attribute, can accept null values.

```
<Property Name="Property1" Type="Edm.String" Nullable="false">
```

Read-only properties

The `Permissions` annotation term can be applied to a property with the value of `OData.Permissions/Read` in order to

specify that it is read-only.

```
<Annotation Term="OData.Permissions" EnumMember="OData.Permissions/Read"/>
```

The `Permissions` annotation term is defined in <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Core.V1.xml>.

Required Properties

The `Required` annotation term is used to specify that a property is required to be supported by services. Properties not annotated with the `Required` annotation, or annotated with a `Boolean` attribute with a value of `"false"`, are optional.

If an implementation supports a property, it shall always provide a value for that property. If a value is unknown, then null is an acceptable values in most cases. Properties not returned from a GET operation shall indicate that the property is not currently supported by the implementation.

```
<Annotation Term="Redfish.Required"/>
```

The `Required` annotation term is defined in <http://redfish.dmtf.org/schemas/v1/RedfishExtensions.1.0.0>.

Required Properties On Create

The `RequiredOnCreate` annotation term is used to specify that a property is required to be specified on creation of the resource. Properties not annotated with the `RequiredOnCreate` annotation, or annotated with a `Boolean` attribute with a value of `"false"`, are not required on create.

```
<Annotation Term="Redfish.RequiredOnCreate"/>
```

The `RequiredOnCreate` annotation term is defined in <http://redfish.dmtf.org/schemas/v1/RedfishExtensions.1.0.0>.

Units of Measure

In addition to following [naming conventions](#), properties representing units of measure shall be annotated with the `Units` annotation term in order to specify the units of measurement for the property.

```
<Annotation Term="OData.Measures.Unit" String="Watts"/>
```

The `Unit` annotation term is defined in <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml>.

Reference Properties

Properties that reference other resources are represented as reference properties using the `NavigationProperty` element. The `NavigationProperty` element specifies the `Name` and namespace qualified `Type` of the related resource(s).

If the property references a single type, the value of the type attribute is the namespace qualified name of the related resource type.

```
<NavigationProperty Name="RelatedType" Type="MyTypes.TypeB">
  <Annotation Term="Core.Description" String="This property references a related resource."/>
  <Annotation Term="Core.LongDescription" String="This is the specification of the related property."/>
  <Annotation Term="OData.AutoExpandReferences"/>
</NavigationProperty>
```

If the property references a collection of resources, the value of the type attribute is of the form:

Collection(NamespaceQualifiedTypeName)

where NamespaceQualifiedTypeName is the namespace qualified name of the type of related resources.

```
<NavigationProperty Name="RelatedTypes" Type="Collection(MyTypes.TypeB)">
  <Annotation Term="Core.Description" String="This property represents a collection of related resources."/>
  <Annotation Term="Core.LongDescription" String="This is the specification of the related property."/>
  <Annotation Term="OData.AutoExpandReferences"/>
</NavigationProperty>
```

All reference properties shall include [Description](#) and [LongDescription](#) annotations.

Contained Resources

Reference properties whose members are contained by the referencing resource are specified with the `ContainsTarget` attribute with a value of `true`.

For example, to specify that a Chassis resource contains a Power resource, you would specify `ContainsTarget=true` on the resource property representing the Power Resource within the Chassis type definition.

```
<NavigationProperty Name="Power" Type="Power.Power" ContainsTarget="true">
  <Annotation Term="OData.Description" String="A reference to the power properties (power supplies, power
  <Annotation Term="OData.LongDescription" String="The value of this property shall be a reference to the
  <Annotation Term="OData.AutoExpandReferences"/>
</NavigationProperty>
```

Expanded References

Reference properties in a Redfish JSON payload are expanded to include the [related resource id](#) or [collection of related resource ids](#). This behavior is expressed using the `AutoExpandReferences` annotation.

```
<Annotation Term="OData.AutoExpandReferences"/>
```

The `AutoExpandReferences` annotation term is defined in <https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Core.V1.xml>.

Expanded Resources

This term can be applied to a [reference property](#) in order to specify that the default behavior for the service is to expand the related [resource](#) or [collection of resources](#) in responses.

```
<Annotation Term="OData.AutoExpand"/>
```

The `AutoExpand` annotation term is defined in <https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Core.V1.xml>.

Resource Actions

Actions are grouped under a property named "Actions".

```
<Property Name="Actions" Type="MyType.Actions">
```

The type of the Actions property is a [structured type](#) with a single OEM property whose type is a structured type with no defined properties.

```

<ComplexType Name="Actions">
  <Property Name="OEM" Type="MyType.OEMActions"/>
</ComplexType>

<ComplexType Name="OEMActions"/>

```

Individual actions are defined within a `namespace` using `Action` elements. The `Name` attribute of the action specifies the name of the action. The `IsBound` attribute specifies that the action is bound to (appears as a member of) a resource or structured type.

The `Action` element contains one or more `Parameter` elements that specify the `Name` and `Type` of each parameter.

The first parameter is called the "binding parameter" and specifies the resource or `structrual type` that the action appears as a member of (the type of the `Actions` property on the resource). The remaining `Parameter` elements describe additional parameters to be passed to the action.

```

<Action Name="MyAction" IsBound="true">
  <Parameter Name="Thing" Type="MyType.Actions"/>
  <Parameter Name="Parameter1" Type="Edm.Boolean"/>
</Action>

```

Resource Extensibility

Companies, OEMs, and other organizations can define additional `properties`, `links`, and `actions` for common Redfish resources using the `Oem` property on resources, links, and actions.

While the information and semantics of these extensions are outside of the standard, the schema representing the data, the resource itself, and the semantics around the protocol shall conform to the requirements in this specification.

Oem Property

In the context of this section, the term "OEM" refers to any company, manufacturer, or organization that is providing or defining an extension to the DMTF-published schema and functionality for Redfish. The base schema for Redfish-specified resources include an empty complex type property called "Oem" whose value can be used to encapsulate one or more OEM-specified complex properties. The `Oem` property in the standard Redfish schema is thus a pre-defined placeholder that is available for OEM-specific property definitions.

Correct use of the `Oem` property requires defining the metadata for an OEM-specified complex type that can be referenced within the `Oem` property. The following fragment is an example of an XML schema that defines a pair of OEM-specific properties under the complex type "AnvilType1". (Other schema elements that would typically be present, such as XML and OData schema description identifiers, are not shown in order to simplify the example).

```

<Schema Name="Contoso.v.v.v">
  . . .
  <ComplexType Name="AnvilType1">
    <Property Name="slogan" Type="Edm.String"/>
    <Property Name="disclaimer" Type="Edm.String"/>
  </ComplexType>
  . . .
</Schema>

```

The next fragment shows an example of how the previous schema and the "AnvilType1" property type might appear in the instantiation of an `Oem` property as the result of a GET on a resource. The example shows two required elements in the use of the `Oem` property: A name for the object and a type property for the object. Detailed requirements for these elements are provided in the following sections.


```

. . .
  "Oem": {
    "Contoso": {
      "@odata.type": "http://Contoso.com/schemas/extensions.v.v.v#contoso.AnvilType1",
      "slogan": "Contoso anvils never fail",
      "disclaimer": "* Most of the time"
    }
  }
. . .

```

Oem Property Format and Content

OEM-specified objects that are contained within the [Oem property](#) must be valid JSON objects that follow the format of a [Redfish complex type](#). The name of the object (property) shall uniquely identify the OEM or organization that manages the top of the namespace under which the property is defined. This is described in more detail in the following section. The OEM-specified property shall also include a [type property](#) that provides the location of the schema and the type definition for the property within that schema. The Oem property can simultaneously hold multiple OEM-specified objects, including objects for more than one company or organization

The definition of any other properties that are contained within the OEM-specific complex type, along with the functional specifications, validation, or other requirements for that content is OEM-specific and outside the scope of this specification. While there are no Redfish-specified limits on the size or complexity of the OEM-specified elements within an OEM-specified JSON object, it is intended that OEM properties will typically only be used for a small number of simple properties that augment the Redfish resource. If a large number of objects or a large quantity of data (compared to the size of the Redfish resource) is to be supported, the OEM should consider having the OEM-specified object point to a separate resource for their extensions.

Oem Property Naming

The OEM-specified objects within the Oem property are named using a unique OEM identifier for the top of the namespace under which the property is defined. There are two specified forms for the identifier. The identifier shall be either an ICANN-recognized domain name (including the top-level domain suffix), or an IANA-assigned Enterprise Number prefaced with "EID:".

Organizations using '.com' domain names may omit the '.com' suffix (e.g. Contoso.com may use 'Contoso', but Contoso.org must use 'Contoso.org' as their OEM property name). The domain name portion of an OEM identifier shall be considered to be case independent. That is, the text "Contoso.biz", "contoso.BIZ", "conTOso.biZ", and so on, all identify the same OEM and top level namespace.

The OEM identifier portion of the property name may be followed by a colon and any additional string to allow further namespacing of OEM-specified objects as desired by the OEM. E.g. "Contoso.com:xxxx" or "EID:412:xxxx". The form and meaning of any text that follows the colon is completely OEM-specific. OEM-specified extension suffixes may be case sensitive, depending on the OEM. Generic client software should treat such extensions, if present, as opaque and not attempt to parse nor interpret the content.

There are many ways this suffix could be used, depending on OEM need. For example, the Contoso company may have a sub-organization "Research", in which case the OEM-specified property name might be extended to be "Contoso:Research". Alternatively, it could be used to identify a namespace for a functional area, geography, subsidiary, and so on.

The OEM identifier portion of the name will typically identify the company or organization that created and maintains the schema for the property. However, this is not a requirement. The identifier is only required to uniquely identify the party that is the top-level manager of a namespace to prevent collisions between OEM property definitions from different vendors or organizations. Consequently, the organization for the top of the namespace may be different than the organization that provides the definition of the OEM-specified property. For example, Contoso may allow one of their customers, e.g. "CustomerA", to extend a Contoso product with certain CustomerA proprietary properties. In this case,

although Contoso allocated the name "contosos:customers.CustomerA" it could be CustomerA that defines the content and functionality under that namespace. In all cases, OEM identifiers should not be used except with permission or as specified by the identified company or organization.

Oem Property Examples

The following fragment presents some examples of naming and use of the Oem property as it might appear when accessing a resource. The example shows that the OEM identifiers can be of different forms, that OEM-specified content can be simple or complex, and that the format and usage of extensions of the OEM identifier is OEM-specific.

```
. . .

"Oem": {
  "Contoso": {
    "@odata.type": "http://contoso.com/schemas/extensions.v.v.v#contoso.AnvilTypes1",
    "slogan": "Contoso anvils never fail",
    "disclaimer": "* Most of the time"
  }
  "Contoso.biz": {
    "@odata.type": "http://contoso.biz/schemas/extension1.1#RelatedSpeed",
    "speed" : "ludicrous"
  }
  "EID:412:ASB_123": {
    "@odata.type": "http://AnotherStandardsBody/schemas.1.0.1#powerInfoExt",
    "readingInfo": {
      "readingAccuracy": "5",
      "readingInterval": "20"
    }
  }
  "Contoso:customers.customerA": {
    "@odata.type" : "http://slingShots.customerA.com/catExt.2015#slingPower",
    "AvailableTargets" : [ "rabbit", "duck", "runner" ],
    "launchPowerOptions" : [ "low", "medium", "eliminate" ],
    "powerSetting" : "eliminate",
    "targetSetting" : "rabbit"
  }
}
. . .
```

Custom Actions

OEM-specific actions can be defined by defining actions bound to the OEM property of the [resource's Actions](#) property type.

```
<Action Name="Ping" IsBound="true">
  <Parameter Name="ContosoType" Type="MyType.OEMActions"/>
</Action>

</Schema>
```

Such bound actions appear in the JSON payload as properties of the Oem type, nested under an [Actions](#) property.

```
"Actions": {
  "OEM": {
    "Contoso.v.v.v#Contoso.Ping": {
      "target": "/redfish/v1/Systems/1/Actions/OEM/Contoso.Ping"
    }
  }
}
```

Custom Annotations

This specification defines a set of common annotations for extending the definition of resource types used by Redfish. In addition, services may define custom annotations.

Services may apply annotations to resources in order to provide service-specific information about the type, such as whether the service supports modifications of particular properties.

Services can apply annotations to existing resources where those resources don't already define a value for the annotation. Services cannot change the value of an annotation applied as part of the resource definition.

Because [service annotations](#) may be applied to existing resource definitions, they are generally specified in a service-specific metadata document referenced by the [service metadata](#).

Common Redfish Resource Properties

This section contains a set of common properties across all Redfish resources. The property names in this section shall not be used for any other purpose, even if they are not implemented in a particular resource.

Common properties are defined in the base Resource Redfish Schema. For OData Schema Representations, this is in Resource.xml and for JSON Schema Representations, this is in Resource.<%= DocVersion %>.json.

Id

The Id property of a resource identifies the resource within a collection.

Name

The Name property is used to convey a human readable moniker for the resource. The type of the Name property shall be string.

Description

The Description property is used to convey a human readable description of the resource. The type of the Description property shall be string.

Status

The Status property represents the status of a resource.

The value of the status property is a common status object type as defined by this specification. By having a common representation of status, clients can depend on consistent semantics. The Status object is capable of indicating the current intended state, the state the resource has been requested to change to, the current actual state and any problem affecting the current state of the resource.

Links

The [Links property](#) represents the links associated with the resource, as defined by that resources schema definition. All associated reference properties defined for a resource shall be nested under the links property. All directly (subordinate) referenced properties defined for a resource shall be in the root of the resource.

RelatedItem

The [RelatedItem property](#) represents links to a resource (or part of a resource) as defined by that resources schema definition. This is not intended to be a strong linking methodology like other references. Instead it is used to show a relationship between elements or sub-elements in disparate parts of the service. For example, since Fans may be in one area of the implementation and processors in another, RelatedItem can be used to inform the client that one is related to

the other (in this case, the Fan is cooling the processor).

Actions

The [Actions](#) property contains the actions supported by a resource.

OEM

The [OEM](#) property is used for OEM extensions as defined in [Schema Extensibility](#).

Redfish Resources

Collectively known as the Redfish Schema, the set of resource descriptions contains normative requirements on implementations conforming to this specification.

Redfish Resources are one of several general kinds:

- Root Service Resource
 - Contains the mapping of a particular service instance to applicable subtending resources.
 - Contains the UUID of a service instance. This UUID would be the same UUID returned via SSDP discovery.
- Current Configuration Resources, contain a mixture of:
 - Inventory (static and read-only)
 - Health Telemetry (dynamic and read-only)
 - Current Configuration Settings (dynamic and read/write)
 - Current Metric values
- Setting Resources
 - Dynamic, Read/Write Pending Configuration Settings
- Services
 - Common services like Eventing, Tasks, Sessions
- Registry Resources
 - Static, Read-Only JSON encoded information for Event and Message Registries

Current Configuration

Current Configuration resources represent the service's knowledge of the current state and configuration of the resource. This may be directly updatable with a PATCH or it may be read-only by the client and the client must PATCH to a separate Setting resource.

Settings

Setting resources represent the future state and configuration of the resource. This property is always associated with a resource through the Redfish.Settings annotation. Where the resource represents the current state, the settings resource represents the future intended state. The state of the resource is changed either directly, such as with a POST of an action or PUT request or indirectly, such as when a user reboots a machine outside of the Redfish service.

Services

Service resources represent components of the Redfish Service itself as well as dependent resources. While the complete list is discoverable only by traversing the Redfish Service tree, the list includes services like the Eventing service, Task management and Session management.

Registry

Registry resources are those resources that assist the client in interpreting Redfish resources beyond the Redfish Schema definitions. Examples of registries include Message Registries, Event Registries and enumeration registries, such as those used for BIOS. In registries, a identifier is used to retrieve more information about a given resource, event,

message or other item. This can include other properties, property restrictions and the like. Registries are themselves resources.

Special Resource Situations

There are some situations that arise with certain kinds of resources that need to exhibit common semantic behavior.

Absent Resources

Resources may be either absent or their state unknown at the time a client requests information about that resource. For removed resources where the URI is expected to remain constant (such as when a fan is removed), the Resource should represent the State property of the Status object as "Absent". In this circumstance, any required or supported properties for which there is no known value shall be represented as null.

Schema Variations

There are cases when deviations from the published Redfish Schema are necessary. An example is BIOS where different servers may have minor variations in available configuration settings. A provider may build a single schema that is a superset of the individual implementations. In order to support these variations, Redfish supports omitting parameters defined in the class schema in the current configuration object. The following rules apply:

- All Redfish services must support attempts to set unsupported configuration elements in the Setting Data by marking them as exceptions in the Setting Data Apply status structure, but not failing the entire configuration operation.
- The support of a specific property in a resource is signaled by the presence of that property in the Current Configuration object. If the element is missing from Current Configuration, the client may assume the element is not supported on that resource.
- For ENUM configuration items that may have variation in allowable values, a special read-only capabilities element will be added to Current Configuration which specifies limits to the element. This is an override for the schema only to be used when necessary.

Providers may split the schema resources into separate files such as Schema + String Registry, each with a separate URI and different Content-Encoding.

- Resources may communicate omissions from the published schema via the Current Configuration object if applicable.

Service Details

Eventing

This section covers the REST-based mechanism for subscribing to and receiving event messages.

The Redfish service requires a client or administrator to create subscriptions to receive events. A subscription is created when an administrator sends an HTTP POST message to the URI of the subscription resource. This request includes the URI where an event-receiver client expects events to be sent, as well as the type of events to be sent. The Redfish service will then, when an event is triggered within the service, send an event to that URI.

- Services shall support "push" style eventing for all resources capable of sending events.
- Services shall not "push" events (using HTTP POST) unless an event subscription has been created. Either the client or the service can terminate the event stream at any time by deleting the subscription. The service may delete a subscription if the number of delivery errors exceeds pre-configured thresholds.
- Services shall respond to a successful subscription with HTTP status 201 and set the HTTP Location header to the address of a new subscription resource. Subscriptions are persistent and will remain across event service restarts.
- Clients shall terminate a subscription by sending an HTTP DELETE message to the URI of the subscription

resource.

- Services may terminate a subscription by sending a special "subscription terminated" event as the last message. Future requests to the associated subscription resource will respond with HTTP status 404.

There are two types of events generated in a Redfish service - life cycle and alert.

Life cycle events happen when resources are created, modified or destroyed. Not every modification of a resource will result in an event - this is similar to when ETags are changed and implementations may not send an event for every resource change. For instance, if an event was sent for every Ethernet packet received or every time a sensor changed 1 degree, this could result in more events than fits a scalable interface. This event usually indicates the resource that changed as well as, optionally, any attributes that changed.

Alert events happen when a resource needs to indicate an event of some significance. This may be either directly or indirectly pertaining to the resource. This style of event usually adopts a message registry approach similar to extended error handling in that a MessageId will be included. Examples of this kind of event are when a chassis is opened, button is pushed, cable is unplugged or threshold exceeded. These events usually do not correspond well to life cycle type events hence they have their own category.

NOTE: Refer to the [Security](#) section for security implications of Eventing

Event Message Subscription

The client locates the eventing service through traversing the Redfish service interface. When the eventing service has been discovered, clients subscribe to messages by sending a HTTP POST to the URL of the collection for subscriptions in the Eventing Service for which they are requesting events. This should be found off of the root service as described in the Redfish Schema for that service.

The specific syntax of the subscription body is found in the Redfish Schema.

On success, the "subscribe" action shall return with HTTP status 201 (CREATED) and the Location header in the response shall contain a URI giving the location of the newly created "subscription" resource. The body of the response, if any, shall contain a representation of the subscription resource. Sending an HTTP GET to the subscription resource shall return the configuration of the subscription.

Clients begin receiving events once a subscription has been registered with the service and do not receive events retroactively. Historical events are not retained by the service.

Event Message Objects

Event message objects POSTed to the specified client endpoint shall contain the properties as described in the Redfish Event Schema.

This event message structure supports a message registry. In a message registry approach there is a message registry that has a list or array of MessageIds in a well known format. These MessageIds are terse in nature and thus they are much smaller than actual messages, making them suitable for embedded environments. In the registry, there is also a message. The message itself can have arguments as well as default values for Severity and RecommendedActions.

The MessageId property contents shall be of the form

RegistryName.MajorVersion.MinorVersion.MessageKey

where

- *RegistryName* is the name of the registry. The registry name shall be Pascal-cased.
- *MajorVersion* is a positive integer representing the major version of the registry
- *MinorVersion* is a positive integer representing the minor version of the registry
- *MessageKey* is a human-readable key into the registry. The message key shall be Pascal-cased and shall not include spaces, periods or special chars.

Subscription Cleanup

To unsubscribe from the messages associated with this subscription, the client or administrator simply sends an HTTP DELETE request to the subscription resource URI.

These are some configurable properties that are global settings that define the behavior for all event subscriptions. See the properties defined in the EventService Redfish Schema for details of the parameters available to configure the service's behavior.

Asynchronous Operations

Services that support asynchronous operations will implement the Task service & Task resource.

The Task service is used to describe the service that handles tasks. It contains a collection of zero or more task resources. The Task resource is used to describe a long running operation that is spawned when a request will take longer than a few seconds, such as when a service is instantiated. Clients will poll the URI of the task resource to determine when the operation has completed and if it was successful.

The Task structure in the Redfish Schema contains the exact structure of a Task. The type of information it contains are start time, end time, task state, task status, and zero or more messages associated with the task.

Each task has a number of possible states. The exact states and their semantics are defined in the Task resource of the Redfish Schema.

When a client issues a request for a long-running operation, the service returns a status of 202 (Accepted).

Any response with a status code of 202 (Accepted) shall include a location header containing the URL of a monitor for the task and may include a wait header to specify the amount of time the client should wait before querying status of the operation.

The client should not include the mime type application/http in the Accept Header when performing a GET request to the status monitor.

The response body of a 202 (Accepted) should contain an instance of the Task resource describing the state of the task.

As long as the operation is in process, the service shall continue to return a status code of 202 (Accepted) when querying the status monitor returned in the location header.

Once the operation has completed, the status monitor shall return a status code of OK (200) and include the headers and response body of the initial operation, as if it had completed synchronously. If the initial operation resulted in an error, the body of the response shall contain an [Error Response](#).

The service may return a status code of 410 (Gone) if the operation has completed and the service has already deleted the task.

The client can continue to get information about the status by directly querying the Task resource using the [resource identifier](#) returned in the body of the 202 (Accepted) response.

- Services that support asynchronous operations shall implement the Task resource
- The response to an asynchronous operation shall return a status code of 202 (Accepted) and set the HTTP response header "Location" to the URI of a status monitor associated with the activity. The response may also include a wait header specifying the amount of time the client should wait before polling for status. The response body should contain a representation of the Task resource in JSON.
- GET requests to either the Task monitor or the Task Resource shall return the current status of the operation without blocking.
- Operations using HTTP GET, PUT, PATCH should always be synchronous.
- Clients shall be prepared to handle both synchronous and asynchronous responses for requests using HTTP

DELETE and HTTP POST methods.

Resource Tree Stability

The Resource Tree, which is defined as the set of URIs and array elements within the implementation, must be consistent on a single service across device reboot and A/C power cycle, and must withstand a reasonable amount of configuration change (e.g. adding an adapter to a server). The resource Tree on one service may not be consistent across instances of devices. The client must walk the data model and discover resources to interact with them. It is possible that some resources will remain very stable from system to system (e.g. BMC network settings) -- but it is not an architectural guarantee.

- A Resource Tree should remain stable across Service restarts and minor device configuration changes, thus the set of URIs and array element indexes should remain constant.
- A Resource Tree shall not be expected by the client to be consistent between instances of services.

Discovery

Automatic discovery of managed devices supporting the Redfish Scalable Platform Management API is accomplished using the Simple Service Discovery Protocol (SSDP). This protocol allows for network-efficient discovery without resorting to ping-sweeps, router table searches, or restrictive DNS naming schemes. Use of SSDP is optional, and if implemented, shall allow the user to disable the protocol through the 'Manager Network Service' resource.

As the objective of discovery is for client software to locate Redfish-compliant managed devices, the primary SSDP functionality incorporated is the M-SEARCH query. Redfish also follows the SSDP extensions and naming used by UPnP where applicable, such that Redfish-compliant systems can also implement UPnP without conflict.

UPnP Compatibility

For compatibility with general purpose SSDP client software, primarily UPnP, TCP port 1900 should be used for all SSDP traffic. In addition, the Time-to-Live (TTL) hop count setting for SSDP multicast messages should default to 2. It is recommended that devices also respond to M-SEARCH queries for UPnP Root Devices (with NT:upnp:rootdevice), with appropriate descriptors and XML documents.

USN Format

The UUID supplied in the USN field shall equal the UUID returned for the Manager implementing the Redfish service. If there are multiple / redundant managers, the UUID shall remain static regardless of redundancy failover. The Unique ID shall be in the canonical UUID format, followed by '::dtmf-org'

M-SEARCH Response

The managed device must respond to M-SEARCH queries searching for Search Target (ST) of the Redfish Service from clients with the AL pointing to the Redfish service root URI. Redfish device shall also respond to M-SEARCH queries for Search Target type of "ssdp:all".

Redfish Service root Search Target (ST): URN:dtmf-org:service:redfish-rest:1

The URN in the reply shall use a service name of 'redfish-rest:' followed by the major version of the Redfish specification. If the minor version of the Redfish Specification to which the service conforms is a non-zero value, and that version is backwards-compatible with previous minor revisions, then that minor version shall be appended, preceeded with a colon. For example, a service conforming to a Redfish specification version "1.4" would reply with a service of "redfish-rest:1:4".

An example response to an M-SEARCH multicast or unicast query shall follow the format shown below. Fields in brackets are placeholders for device-specific values.

HTTP/1.1 200 OK


```

CACHE-CONTROL:<seconds, at least 1800>
ST:urn:dmtf-org:service:redfish-rest:1
USN:uuid:<UUID of Manager>::urn:dmtf-org:service:redfish-rest:1
AL:<URL of Redfish service root>
EXT:

```

Notify, Alive, and Shutdown messages

Redfish devices may implement the additional SSDP messages defined by UPnP to announce their availability to software. This capability, if implemented, must allow the end user to disable the traffic separately from the M-SEARCH response functionality. This allows users to utilize the discovery functionality with minimal amounts of network traffic generated.

Security

Goals

- Privilege Model to Monitor and Manage:
 - System Settings
 - BIOS Configuration
 - System Power States
 - Sensor Information (power/thermal/health)
 - Network Settings
 - Storage Settings
 - Logs
 - Redfish Service Configuration
 - Account Management
 - Network Settings
 - Logs
 - Firmware versions
 - OEM vendor-specific features and functionality
- Permission/ authorization model shall be consistent between instances of Redfish compliant devices
 - Define a minimum baseline for the permission/ authorization model
- Infrastructure Authentication
- CURL compatibility
- Automated clients
- Embedded Service Processors

Protocols

TLS

Implementations shall support TLS v1.1 or later

Cipher suites

Implementations should support AES-256 based ciphers from the TLS suites.

Redfish implementations should consider supporting ciphers similar to below which enable authentication and identification without use of trusted certificates.

```
TLS_PSK_WITH_AES_256_GCM_SHA384
```

```
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
```

Additional advantage with using above recommended ciphers is -

"AES-GCM is not only efficient and secure, but hardware implementations can achieve high speeds with low cost and low latency, because the mode can be pipelined."

References to RFCs -

```
http://tools.ietf.org/html/rfc5487
http://tools.ietf.org/html/rfc5288
```

Certificates

Implementations shall support replacement of the default certificate if one is provided, with a certificate having at least a 4096 bit RSA key and sha512-rsa signature.

Authentication

- Authentication Methods

Service shall support both "Basic Authentication" and "Redfish Session Login Authentication" (as described below under Session Management). Services shall not require a client to create a session when Basic Auth is used.

Services may implement other authentication mechanisms.

HTTP Header Security

- All write activities shall be authenticated, i.e. POST, PUT/PATCH, and DELETE, except for
 - The POST operation to the Sessions service/object needed for authentication
 - Extended error messages shall NOT provide privileged info when authentication failures occur
- REST objects shall not be available unauthenticated, except for
 - The root object which is needed to identify the device and service locations
 - The \$metadata object which is needed to retrieve resource types
 - The OData Service Document which is needed for compatibility with OData clients
 - The version object located at /redfish
- External services linked via external references are not part of this spec, and may have other security requirements.

HTTP Redirect

- When there is a HTTP Redirect the privilege requirements for the target resource shall be enforced

Extended Error Handling

- Extended error messages shall NOT provide privileged info when authentication failures occur

HTTP Header Authentication

- HTTP Headers for authentication shall be processed before other headers that may affect the response, i.e.: etag, If-Modified, etc.
- HTTP Cookies shall NOT be used to authenticate any activity i.e.: GET, POST, PUT/PATCH, and DELETE.

BASIC authentication

HTTP BASIC authentication as defined by [RFC2617](#) shall be supported, and shall only use compliant TLS connections to transport the data between any third party authentication service and clients.

Request / Message Level Authentication

Every request that establishes a secure channel shall be accompanied by an authentication header.

Session Management

Session Lifecycle Management

Session management is left to the implementation of the Redfish Service. This includes orphaned session timeout and number of simultaneous open sessions.

- **A Redfish Service shall provide login sessions compliant with this specification.**

Redfish Login Sessions

For functionality requiring multiple Redfish operations, or for security reasons, a client may create a Redfish Login Session via the session management interface. The URI used for session management is specified in the Session Service. The URI for establishing a session can be found in the SessionService's Session property or in the Service Root's Links Section under the Sessions property. Both URIs shall be the same.

```
{
  ...
  "SessionService": {
    "@odata.id": "/redfish/v1/SessionService"
  },
  "Links": {
    "Sessions": {
      "@odata.id": "/redfish/v1/SessionService/Sessions"
    }
  },
  ...
}
```

Session Login

A Redfish session is created by an HTTP POST to the SessionService' Sessions collection resource, including the following POST body:

```
POST /redfish/v1/SessionService/Sessions HTTP/1.1
Host: <hostpath>
Accept: application/json
Content-Type: charset=utf-8
OData-Version: 4.0

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

The Origin header should be saved in reference to this session creation and compared to subsequent requests using this session to verify the request has been initiated from an authorized client domain.

The response to the POST request to create a session includes:

- an X-Auth-Token header that contains a "session auth token" that the client can use in subsequent requests, and
- a "Location" header that contains a link to the newly created session resource.
- The JSON response body that contains a full representation of the newly created session object:

```
POST /redfish/v1/SessionService/Sessions HTTP/1.1
Content-Type: application/json
```

```

Content-Length: <computed length>
OData-Version: 4.0
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.1.0.0.Session",
  "Id": "1",
  "Name": "User Session",
  "Description": "User Session",
  "UserName": "<username>"
}

```

The client sending the session login request should save the "Session Auth Token" and the link returned in the Location header. The "Session Auth Token" is used to authentication subsequent requests by setting the Request Header "X-Auth-Token" with the "Session Auth Token" received from the login POST. The client will later use the link that was returned in the Location header of the POST to logout or terminate the session.

Note that the "Session ID" and "Session Auth Token" are different. The Session ID uniquely identifies the session resource and is returned with the response data as well as the last segment of the Location header link.

An administrator with sufficient privilege can view active sessions and also terminate any session using the associated sessionId. Only the client that executes the login will have the Session Auth Token.

X-Auth-Token HTTP Header

Implementations shall only use compliant TLS connections to transport the data between any third party authentication service and clients. Therefore, the POST to create a new session shall only be supported with HTTPS, and all requests that use Basic Auth shall require HTTPS.

Session Lifetime

Note that Redfish sessions "time-out" as apposed to having a token expiration time like some token-based methods use. For Redfish sessions, as long a client continues to send requests for the session more often than the session timeout period, the session will remain open and the session auth token remains valid. If the sessions times-out then the session is automatically terminated. Note that the Redfish.

Session Termination or Logout

A Redfish session is terminated when the client Logs-out. This is accomplished by performing a DELETE to the Session resource identified by the link returned in the Location header when the session was created, or the SessionId returned in the response data.

The ability to DELETE a Session by specifying the Session resource ID allows an administrator with sufficient privilege to terminate other users sessions from a different session.

AccountService

- User passwords should be stored with one-way encryption techniques.
- Implementations may support exporting user accounts with passwords, but shall do so using encryption methods to protect them.
- User accounts shall support ETags and shall support atomic operations
 - Implementations may reject requests which do not include an ETag
- User Management activity is atomic
- Extended error messages shall NOT provide privileged info when authentication failures occur

Async Tasks

- Irrespective of which users/ privileged context was used to start an async task the information in the status object shall be used to enforce the privilege(s) required to access that object.

Event Subscriptions

- The Redfish device may verify the destination for identity purposes before pushing event data object to the Destination

Privilege Model / Authorization

The Authorization subsystem uses Roles and Privileges to control which users have what access to resources.

- Roles:
 - A Role is a defined set of Privileges. Therefore, two roles with the same privileges shall behave equivalently.
 - All users are assigned exactly one role.
 - This specification defines a set of pre-defined roles, one of which shall be assigned to a user when a user is created.
 - The pre-defined roles shall be created as follows:
 - Role Name = "Administrator"
 - AssignedPrivileges = Login, ConfigureManager, ConfigureUser, ConfigureComponent, ConfigureSelf
 - Role Name = "Operator"
 - AssignedPrivileges = Login, ConfigureComponent, ConfigureSelf
 - Role Name = "ReadOnly"
 - AssignedPrivileges = Login, ConfigureSelf
 - Implementations shall support all of the pre-defined roles.
 - The pre-defined Roles may include OEM privileges.
 - The privilege array defined for the predefined roles shall not be modifiable.
 - A service may optionally support additional "Custom" roles, and may allow users to create such custom roles by:
 - 1) posting to the Roles collection; or 2) an implementation may implement a predefined custom role; or 3) other mechanism outside the specification.
- Privileges:
 - A privilege is a permission to perform an operation (e.g. Read, Write) within a defined management domain (e.g. Configuring Users).
 - The Redfish specification defines a set of "assigned privileges" in the AssignedPrivileges array in the Role resource.
 - An implementation may also include "OemPrivileges" which are then specified in an OemPrivileges array in the Role resource.
 - Privileges are mapped to resources using the privilege mapping annotations defined in the Privileges Redfish Schema file.
 - Multiple privileges in the mapping constitute an OR of the privileges.
- User Management:
 - Users are assigned a Role when the user account is created.
 - The privileges that the user has are defined by its role.
- ETag Handling:
 - Implementations shall enforce the same privilege model for ETag related activity as is enforced for the data being represented by the ETag.
 - For example, when activity requiring privileged access to read data item represented by ETag requires the same privileged access to read the ETag.

Data Model Validation

Schema

Server and Client implementations should check supplied data against Redfish Schema and perform data validation checks to prevent vulnerabilities caused by later processing errors.

When there is a disagreement between a Server and Client on Redfish Schema validation, the server may enforce its version and reject the request.

Clients shall NOT perform data interpolation unless the Redfish Schema permits that.

Privileges should NOT be modified without a strong security related requirement. Redfish Schema validation shall include privilege checks when privilege requirements have been modified.

NOTE: Privilege changes as part of Redfish Schema updates/changes shall be captured in the Redfish Schema change log.

Idempotent actions shall be rejected when there is a security reason to do so.

Resource definitions shall include required privileges to perform read/ RW actions on that resource.

Resource tree stability - Permissions on resources should be stable as well.

Custom Actions - Privilege model shall be applied consistently to both the body and the response. Where applicable the privilege model defined for the URI should be inherited for custom actions.

Logging

Required data for security log entries

Implementations shall log authentication requests including failures. Authentication login/logout log entries shall contain a user identifier that can be used to uniquely identify the client and a time stamp.

Completeness of Logging

- Every entity from the originator of the RESTful service call, through every intermediary, to the very last entity in the call chain, log an entry in their audit log for the call activity triggered/ taken/ ... This means same as any RESTful service call, the audit log entry will 'be complete' for the activity performed within said entity.
 - shall - All write activities i.e. POST, PUT/ PATCH and DELETE
 - NOTE: When a new log entry is created logging the occurrence of that event is not required.
 - shall - Have the ability to log the privileged reads i.e. GETs
 - This ability may be turned on by default.
- Rejection of idempotent actions due to security reasons shall be logged

Content of Audit Logs

Details : Need to generate events for the following

1. logon, log-off, modification of user accounts
2. successful and rejected login attempts,
3. successful and rejected connections to nodes and other resource access attempts
4. details about the modification of user accounts
5. all changes to the system configuration,
6. information about the use of built-in utilities running in Redfish compliant-devices(e.g. low-level diagnostic tools),
7. information about accessing the system interfaces of the Redfish compliant-devices
8. network addresses and protocols (e.g. workstation IP address and protocol used for access)

9. activation and de-activation of protection measures

The file where the events are written, one or more messages per event should at least have the following information :

- User ID
- Date, time
- Event type
- Event description

ANNEX A (informative)

Change Log

Version	Date	Description
0.94.0	2015-13-1	Initial merge of 0.91 and 0.92 versions
0.96.0	2015-3-3	Near-final Chassis and ComputerSystem schemas. Introduction of referenceable (array) members and use for power metrics, thermal metrics. Introduction of SessionService. Added JSONSchemaFile to OData metadata, mockups. Miscellaneous clean-up.
0.99.0	2015-6-18	Final work in progress release prior to v1.0.

