



Document Identifier: DSP0266

Date: 2018-12-13

Version: 1.6.1

Redfish Scalable Platforms Management API Specification

Supersedes: 1.6.0

Document Class: Normative

Document Status: Published

Document Language: en-US

Copyright Notice

Copyright © 2015-2018 DMTF. All rights reserved.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

For information about patents held by third-parties which have notified the DMTF that, in their opinion, such patent may relate to or impact implementations of DMTF standards, visit <http://www.dmtf.org/about/policies/disclosures.php>.

This document's normative language is English. Translation into other languages is permitted.

CONTENTS

1. Abstract	8
2. Normative references	8
3. Terms and definitions	9
4. Symbols and abbreviated terms	12
5. Overview	13
5.1. Scope	13
5.2. Goals	14
5.3. Design tenets	14
5.4. Limitations	15
5.5. Additional design background and rationale	15
5.5.1. REST-based	15
5.5.2. Follow OData conventions	16
5.5.3. Model-oriented	16
5.5.4. Separation of protocol from data model	16
5.5.5. Hypermedia API service endpoint	17
5.6. Service elements	17
5.6.1. Synchronous and asynchronous operation support	17
5.6.2. Eventing mechanism	17
5.6.3. Actions	17
5.6.4. Service entry point discovery	18
5.6.5. Remote access support	18
5.7. Security	18
6. Protocol details	19
6.1. Use of HTTP	19
6.1.1. URIs	20
6.1.2. HTTP methods	21
6.1.3. HTTP redirect	22
6.1.4. Media types	22
6.1.5. ETags	22
6.2. Protocol version	23
6.3. Redfish-defined URIs and relative URI rules	24
6.4. Requests	25
6.4.1. Request headers	25
6.4.2. Read requests (GET)	29
6.4.3. HEAD	36
6.4.4. Data modification requests	36
6.4.5. Operation apply time	43
6.5. Responses	46
6.5.1. Response headers	46
6.5.2. Status codes	49
6.5.3. Metadata responses	52

6.5.4. Resource responses	54
6.5.5. Resource Collection responses	64
6.5.6. Error responses	66
7. Data model and Schema	68
7.1. Schema, registry and profile repository	69
7.1.1. Schema, registry, and profile file naming conventions	69
7.1.2. Programmatic access to schema, registry, or profile files	71
7.2. Type identifiers	71
7.2.1. Type identifiers in JSON Schema	71
7.2.2. Type identifiers in JSON	72
7.3. Common naming conventions	72
7.4. Localization considerations	73
7.5. Schema definition	73
7.5.1. Common annotations	73
7.5.2. Schema documents	73
7.5.3. Resource type definitions	75
7.5.4. Resource capabilities	76
7.5.5. Resource URI pattern definitions	77
7.5.6. Resource properties	78
7.5.7. Reference properties	82
7.5.8. Resource actions	84
7.5.9. Resource extensibility	85
7.5.10. Oem property examples	88
7.6. Common Redfish resource properties	90
7.6.1. Id	90
7.6.2. Name	90
7.6.3. Description	90
7.6.4. MemberId	90
7.6.5. Status	90
7.6.6. Links	90
7.6.7. Members	91
7.6.8. RelatedItem	91
7.6.9. Actions	17
7.6.10. OEM	91
7.7. Redfish resources	91
7.7.1. Current configuration	92
7.7.2. Settings	92
7.7.3. Services	93
7.7.4. Registry	93
7.8. Special resource situations	94
7.8.1. Absent resources	94
7.8.2. Schema variations	94
8. Service details	95

8.1. Eventing	95
8.1.1. Event subscription types	95
8.1.2. EventType based eventing	96
8.1.3. Ways to register for events	97
8.1.4. Event formats	98
8.1.5. OEM Extensions	99
8.2. Asynchronous operations	99
8.3. Resource Tree stability	101
8.4. Discovery	101
8.4.1. UPnP compatibility	101
8.4.2. USN format	101
8.4.3. M-SEARCH response	101
8.4.4. Notify, alive, and shutdown messages	102
8.5. Server-Sent Events	102
8.5.1. EventService	103
9. Security	18
9.1. Protocols	106
9.1.1. TLS	106
9.1.2. Cipher suites	107
9.1.3. Certificates	107
9.2. Authentication	107
9.2.1. HTTP header security	108
9.2.2. Extended error handling	108
9.2.3. HTTP header authentication	108
9.2.4. Session management	109
9.2.5. AccountService	111
9.2.6. Async tasks	111
9.2.7. Event subscriptions	112
9.2.8. Privilege model/Authorization	112
9.2.9. Redfish Service Operation-to-Privilege mapping	113
10. Redfish Host Interface	120
11. Redfish Composability	120
11.1. Composition requests	121
11.1.1. Specific Composition	121
12. ANNEX A (informative)	122
12.1. Change log	122

Foreword

The Redfish Scalable Platforms Management API ("Redfish") was prepared by the Redfish 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 Enterprise
- Jeff Bobzin - Insyde Software Corp.
- Patrick Boyd - Dell Inc.
- David Brockhaus - Vertiv
- Richard Brunner - VMware Inc.
- Lee Calcote - Seagate Technology
- P Chandrasekhar - Dell Inc.
- Chris Davenport - Hewlett Packard Enterprise
- Gamma Dean - Vertiv
- Daniel Dufresne - Dell Inc.
- Samer El-Haj-Mahmoud - Lenovo, Hewlett Packard Enterprise
- George Ericson - Dell Inc.
- Wassim Fayed - Microsoft Corporation
- Mike Garrett - Hewlett Packard Enterprise
- Steve Geffin - Vertiv
- Joe Handzik - Hewlett Packard Enterprise
- Jon Hass - Dell Inc.
- Jeff Hilland - Hewlett Packard Enterprise
- Chris Hoffman - Vertiv
- Steven Krig - Intel Corporation
- John Leung - Intel Corporation
- Jagan Molleti - Dell Inc.
- Milena Natanov - Microsoft Corporation
- Michael Pizzo - Microsoft Corporation
- Chris Poblete - Dell Inc.
- Michael Raineri - Dell Inc.
- Irina Salvan - Microsoft Corporation
- Hemal Shah - Broadcom Limited
- Jim Shelton - Vertiv
- Tom Slaight - Intel Corporation
- Donnie Sturgeon - Vertiv
- Pawel Szymanski - Intel Corporation
- Paul Vancil - Dell Inc.
- Linda Wu - Super Micro Computer, Inc.

1. Abstract

The Redfish Scalable Platforms Management API ("Redfish") is a standard that uses RESTful interface semantics to access data defined in model format to perform 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.

While the initial Redfish scope was targeted at servers, expansion of scope has grown both in the DMTF and through DMTF alliance partners to cover most data center IT equipment and other solutions as well. It also covers both in-band and out-of-band access methods.

Educational material is also increasing, both from the DMTF and other organizations that utilize Redfish as part of their industry standard or solution.

2. 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 1738](http://www.ietf.org/rfc/rfc1738.txt) T. Berners-Lee et al, Uniform Resource Identifier (URI), <http://www.ietf.org/rfc/rfc1738.txt>
- [IETF RFC 3986](http://www.ietf.org/rfc/rfc3986.txt) T. Berners-Lee et al, Uniform Resource Identifier (URI): Generic Syntax, <http://www.ietf.org/rfc/rfc3986.txt>
- [IETF RFC 4627](http://www.ietf.org/rfc/rfc4627.txt), D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), <http://www.ietf.org/rfc/rfc4627.txt>
- [IETF RFC 5789](http://www.ietf.org/rfc/rfc5789.txt), L. Dusseault et al, PATCH method for HTTP, <http://www.ietf.org/rfc/rfc5789.txt>
- [IETF RFC 5280](http://www.ietf.org/rfc/rfc5280.txt), D. Cooper et al, Web linking, <http://www.ietf.org/rfc/rfc5280.txt>
- [IETF RFC 5988](http://www.ietf.org/rfc/rfc5988.txt), M. Nottingham, Web linking, <http://www.ietf.org/rfc/rfc5988.txt>
- [IETF RFC 6585](http://www.ietf.org/rfc/rfc6585.txt), M. Nottingham, et al, Additional HTTP Status Codes, <http://www.ietf.org/rfc/rfc6585.txt>
- [IETF RFC 6901](http://www.ietf.org/rfc/rfc6901.txt), P. Bryan, Ed. et al, JavaScript Object Notation (JSON) Pointer, <http://www.ietf.org/rfc/rfc6901.txt>
- [IETF RFC 6906](http://www.ietf.org/rfc/rfc6906.txt), E. Wilde, The 'profile' Link Relation Type, <http://www.ietf.org/rfc/rfc6906.txt>
- [IETF RFC 7230](http://www.ietf.org/rfc/rfc7230.txt), R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, <http://www.ietf.org/rfc/rfc7230.txt>
- [IETF RFC 7231](http://www.ietf.org/rfc/rfc7231.txt), R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, <http://www.ietf.org/rfc/rfc7231.txt>
- [IETF RFC 7232](http://www.ietf.org/rfc/rfc7232.txt), R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, <http://www.ietf.org/rfc/rfc7232.txt>

- [IETF RFC 7234](http://www.ietf.org/rfc/rfc7234.txt), R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Caching, <http://www.ietf.org/rfc/rfc7234.txt>
- [IETF RFC 7235](http://www.ietf.org/rfc/rfc7235.txt), R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Authentication, <http://www.ietf.org/rfc/rfc7235.txt>
- [ISO/IEC Directives](http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtypeH), 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: A Media Type for Describing JSON Documents, Draft 7](https://tools.ietf.org/html/draft-handrews-json-schema-01) <https://tools.ietf.org/html/draft-handrews-json-schema-01>
- [JSON Schema Validation: A Vocabulary for Structural Validation of JSON, Draft 7](https://tools.ietf.org/html/draft-handrews-json-schema-validation-01) <https://tools.ietf.org/html/draft-handrews-json-schema-validation-01>
- [OData Version 4.0 Part 1: Protocol](http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html). 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html>
- [OData Version 4.0 Part 2: URL Conventions](http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html). 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html>
- [OData Version 4.0 Part 3: Common Schema Definition Language \(CSDL\)](http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html). 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html>
- [OData Version 4.0: Core Vocabulary](http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Core.V1.xml). 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](http://docs.oasis-open.org/odata/odata-json-format/v4.0/os/odata-json-format-v4.0-os.html). 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](http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml). 24 February 2014. <http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml>
- [Simple Service Discovery Protocol/1.0](http://tools.ietf.org/html/draft-cai-ssdp-v1-03). 28 October 1999. <http://tools.ietf.org/html/draft-cai-ssdp-v1-03>
- [The Unified Code for Units of Measure](http://www.unitsofmeasure.org/ucum.html). <http://www.unitsofmeasure.org/ucum.html>
- [W3C Recommendation of Cross-Origin Resource Sharing](http://www.w3.org/TR/cors/). 16 January 2014. <http://www.w3.org/TR/cors/>
- [SNIA TLS Specification for Storage Systems](http://www.snia.org/tls/). 20 November 2014. <http://www.snia.org/tls/>
- [DMTF DSP0270 Redfish Host Interface Specification](http://www.dmtf.org/sites/default/files/standards/documents/DSP0270_1.0.pdf), http://www.dmtf.org/sites/default/files/standards/documents/DSP0270_1.0.pdf
- [HTML5 Specification: Server-Sent Events](https://html.spec.whatwg.org/multipage/server-sent-events.html) <https://html.spec.whatwg.org/multipage/server-sent-events.html>
- [OpenAPI Specification](https://github.com/OAI/OpenAPI-Specification) <https://github.com/OAI/OpenAPI-Specification>

3. 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	See Resource Collection.
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.
Member	A Member is a single resource instance contained in a Resource Collection
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 that 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.
Property	A name/value pair included in a Redfish-defined resource as part of a request or

Term	Definition
	response. A property may be defined as any valid JSON data type.
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	The 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 Provider	A Redfish provider interacts with the Redfish Service to contribute resources to the Redfish resource tree and reacts to changes in its owned resources. There are two types of Redfish providers: internal providers and external providers. A internal provider is the Redfish Service itself that has a data model and can react to RESTful operations from a client. An external provider is an architected means for agents external to the Redfish Service to augment the Redfish resource tree. The interaction between a Redfish provider and a Redfish Service is not covered by this specification.
Redfish Schema	The Schema definitions for Redfish resources. It is defined according to OData Schema representation that can be directly translated to a JSON Schema representation.
Redfish Service	Also referred to as the "Service". The set 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 Collection	A Resource Collection is a Resource that acts as a container of other Resources. The Members of a Resource 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

Term	Definition
	Members of the container.
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 hyperlinks 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.

4. 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

Term	Definition
NIC	Network Interface Card
PCI	Peripheral Component Interconnect
PCIe	PCI Express
TCP	Transmission Control Protocol
XSS	Cross-Site Scripting

5. Overview

The Redfish Scalable Platforms 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 and OpenAPI representations, 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 metadata, thus enabling more advanced client scenarios as found in many data center and cloud environments.

5.1. Scope

The scope of this specification is to define the protocols, data model, and behaviors, as well as other architectural components needed for an interoperable, 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 extensions specific to the OEM (system vendor) 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.

5.2. 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.

5.3. 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.

5.4. 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.

Interoperable 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 hyperlinks between resources and other elements can vary across implementations. Clients should not assume that hyperlinks can be reused across different instantiations of a Redfish Service.

5.5. Additional design background and rationale

5.5.1. 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 and 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 and discovery mechanisms in the industry.

5.5.2. 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 annotations that, 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.

5.5.3. Model-oriented

The Redfish model is built for managing systems. All resources are defined in OData Schema representation and translated to JSON Schema and OpenAPI representations. 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.

5.5.4. 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.

5.5.5. Hypermedia API service endpoint

Like other hypermedia APIs, Redfish has a single service endpoint URI and all other resources are accessible via URIs referenced from the root. Any resource discovered through hyperlinks 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.

5.6. Service elements

5.6.1. 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 status codes](#) enable a client to determine whether the operation was completed synchronously or asynchronously. For more information, see the clause on [Tasks](#).

5.6.2. 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 enable reception of events by creating a 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.

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

5.6.3. 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.

5.6.4. 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 clause on [Discovery](#)

5.6.5. 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 redirection (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.

5.7. 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.

6. 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 and OpenAPI representations for validation and consumption by tools supporting JSON Schema or OpenAPI. Following these common standards and conventions increases interoperability and enables leveraging of existing tool chains.

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](#) clause 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.

6.1. Use of HTTP

HTTP is ideally suited to a RESTful interface. This clause 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 ([RFC7230](#), [RFC7231](#), [RFC7232](#)).

6.1.1. URIs

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

- A URI shall identify each unique instance of a resource.
- URIs shall not include any unsafe characters as specified in [RFC1738](#):
 - Includes characters such as {, }, |, \, ^, ~, [,], ` , and ".
 - Also includes # for anything other than an indicator for the start of a fragment.
- URIs shall not include any percent encoding of characters:
 - Does not include the [query parameter](#) portion of the URI. To begin operations, a client must know a URI for a resource.
- Performing a GET operation yields a representation of the resource containing properties and hyperlinks 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 hyperlinks returned in previous responses. This type of API that is consumed by navigating URIs returned by the service is known as a Hypermedia API.

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 that 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 within a given Redfish service.

- The unique identifier part of a URI shall be unique within the implementation.
- An implementation may use a [relative URI](#) in the payload (body and/or HTTP headers) to identify a resource within the implementation.
- An implementation may use an absolute URI in the payload (body and/or HTTP headers) to identify a resource within a different implementation. See [RFC3986](#) for the absolute URI definition.

For example, a POST may return the following URI in the Location header of the response (indicating the new resource created by the POST): [/redfish/v1/Systems/2](#)

Assuming the client is connecting through an appliance named "mgmt.vendor.com", the absolute 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`) component. Queries are addressed in the [Query Parameters](#) clause. The server shall ignore fragments (`#frag`) that are used as the URI for submitting an operation.

If a property in a response is a reference to another property within a resource, the "URI Fragment Identifier Representation" format as specified in [RFC6901](#) shall be used. If the property is defined as a [reference property](#) within the schema, the fragment shall reference a valid [resource identifier](#).

For example, the following fragment identifies a property at index 0 of the `Fans` array within the `/redfish/v1/Chassis/MultiBladeEncl/Thermal` resource:

```
{
  "@odata.id": "/redfish/v1/Chassis/MultiBladeEncl/Thermal#/Fans/0"
}
```

6.1.2. HTTP methods

An attractive feature of the RESTful interface is the very limited number of operations that 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 retrieval	Yes
PUT	Object replace	No
PATCH	Object update	Yes
DELETE	Object delete	Yes
HEAD	Object header retrieval	No
OPTIONS	Header retrieval, CORs preflight	No

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

6.1.3. 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](#) clause for security implications of HTTP Redirect

6.1.4. 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.

- Services should support gzip compression when requested by the client.

6.1.5. 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 GET requests of ManagerAccount resources.

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. If an ETag is supported for a given resource, it shall use the strong ETag format as specified in [RFC7232](#).

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. If a service supports returning the ETag header on a resource, the service may respond with status code [428](#) if the If-Match/If-None-Match header is missing from the PUT/PATCH request for the same resource, as specified in [RFC6585](#).

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: "<string>"
```

6.2. 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 minor versions.
- *Errata* = integer: something in the prior version was broken and needed to be fixed.

Any resource discovered through hyperlinks 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/"
}
```

6.3. Redfish-defined URIs and relative URI rules

The following Redfish-defined URIs shall be supported by a Redfish Service:

URI	Description
<code>/redfish</code>	The URI that is used to return the version
<code>/redfish/v1/</code>	The URI for the Redfish Service Root
<code>/redfish/v1/odata</code>	The URI for the Redfish OData Service Document
<code>/redfish/v1/\$metadata</code>	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 treated by the service as the equivalent URI to the associated Redfish-defined URI:

URI	Associated Redfish-Defined URI
<code>/redfish/v1</code>	<code>/redfish/v1/</code>

All other Redfish URIs supported by the service shall match the URI patterns described by the [Resource URI pattern definitions section](#), with the exception to supplemental resources referenced by the `"@Redfish.Settings"`, `"@Redfish.ActionInfo"`, and `"@Redfish.CollectionCapabilities"` payload annotations. The URIs for these supplemental resources shall be treated as opaque by the client.

All URIs supported by a Redfish Service shall be considered reserved for future standardization by DMTF and DMTF Alliance Partners, with the exception to OEM extension URIs, which shall conform to the requirements in clause [URIs for OEM resources](#).

All relative URIs used by the service shall start with a double forward slash (`"/"`) and include the authority (e.g., `//mgmt.vendor.com/redfish/v1/Systems`) or a single forward slash (`"/"`) and include the absolute-path (e.g., `/redfish/v1/Systems`).

6.4. Requests

This clause describes the requests that can be sent to Redfish Services.

6.4.1. Request headers

HTTP defines headers that can be used in request messages. The following table defines those headers and their requirements for Redfish Services. Note that these are requirements for the Redfish Services, and not the clients sending the HTTP requests.

- 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 Service Requirement 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 Service Requirement 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 Service Requirement column is set to "No".
- Redfish Clients shall include the headers in the following table as defined by the HTTP 1.1 specification if the value in the Client Requirement column is set to "Yes".
- Redfish Clients shall include the headers in the following table as defined by the HTTP 1.1 specification if the value in the Client Requirement column is set to "Conditional" under the conditions noted in the description.
- Redfish Clients should transmit the headers in the following tables as defined by the HTTP 1.1 specification if the value in the Client Requirement column is set to "No".

Header	Service Requirement	Client Requirement	Supported Values	Description
Accept	Yes	No	RFC 7231	Indicates to the server what media type(s) this client is prepared to accept. Services shall support requests for resources with an Accept header including application/json or application/json; charset=utf-8. Services shall support requests for metadata with an Accept header including

Header	Service Requirement	Client Requirement	Supported Values	Description
				application/xml or application/xml; charset=utf-8. Services shall support requests for all resources with an Accept header including application/*, application/*; charset=utf-8, */*, or */*; charset=utf-8.
Accept-Encoding	No	No	RFC 7231	Indicates if gzip encoding can be handled by the client. If an Accept-Encoding header is present in a request and the service cannot send a response that is acceptable according to the Accept-Encoding header, then the service should respond with status code 406 . Services should not return responses gzip encoded if the Accept-Encoding header is not present in the request.
Accept-Language	No	No	RFC 7231	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	Conditional	RFC 7231	Describes the type of representation used in the message body. Content-Type shall be required in requests that include a request body. Services shall accept Content-Type values of application/json or application/json; charset=utf-8. It is recommended that Clients use these values in requests since other values may result in an error.

Header	Service Requirement	Client Requirement	Supported Values	Description
Content-Length	No	No	RFC 7231	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 .
OData-MaxVersion	No	No	4.0	Indicates the maximum version of OData that an odata-aware client understands
OData-Version	Yes	No	4.0	Services shall reject requests that specify an unsupported OData version. If a service encounters a version that it does not support, the service should reject the request with status code [412] (#status-412).
Authorization	Conditional	Conditional	RFC 7235 , Section 4.2	Required for Basic Authentication . A client can access unsecured resources without using this header on systems that support Basic Authentication.
User-Agent	Yes	No	RFC 7231	Required for tracing product tokens and their version. Multiple product tokens may be listed.
Host	Yes	No	RFC 7230	Required to allow support of multiple origin hosts at a single IP address.
Origin	Yes	No	W3C CORS , Section 5.7	Used to allow web applications to consume Redfish Service while preventing CSRF attacks.
Via	No	No	RFC 7230	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.

Header	Service Requirement	Client Requirement	Supported Values	Description
Max-Forwards	No	No	RFC 7231	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
If-Match	Conditional	No	RFC 7232	If-Match shall be supported on PUT and PATCH requests for resources for which the service returns ETags, to ensure clients are updating the resource from a known state. While not required for clients, it is highly recommended for PUT and PATCH operations.
If-None-Match	No	No	RFC 7232	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 .
Last-Event-ID	No	No	HTML5 SSE	Used by the client to request history event data. See the Server-Sent Events section for more information.

- 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	Service Requirement	Client Requirement	Supported Values	Description
X-Auth-Token	Yes	Conditional	Opaque encoded octet strings	Used for authentication of user sessions. The token value shall be indistinguishable from random. While it is required for services to support, a client can access unsecured resources without establishing a session.

6.4.2. Read requests (GET)

The GET method is used to retrieve a representation of a resource. The service shall return the representation using one of the media types specified in the Accept header, subject to requirements in the Media Types clause [Media Types](#). If the Accept header is not present, the service shall return the resource's representation 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.

6.4.2.1. 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.

Services shall not require authentication in order to retrieve the service root and `"/redfish"` documents.

6.4.2.2. 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 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.

6.4.2.3. 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.

6.4.2.4. Resource retrieval requests

Clients request resources by issuing GET requests to the URI for the individual resource. The URI for a resource may be obtained from a [resource identifier property](#) returned in a previous request (for example, within the [Links Property](#) 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.

6.4.2.4.1. Query parameters

Clients can add query parameters to request additional features from the service. These features include

pagination, selection, filtering and expansion, and are explained below.

Attribute	Description	Example
\$skip	Integer indicating the number of Members in the Resource Collection to skip before retrieving the first resource.	<code>http://resourcecollection?\$skip=5</code>
\$top	Integer indicating the number of Members to include in the response. The minimum value for this parameter is 1. The default behavior is to return all Members.	<code>http://resourcecollection?\$top=30</code>
only	For Resource Collections, if there is exactly one Member in the collection, return that member's resource in place of the Resource Collection. This parameter shall be ignored when combined with '\$expand' or for resources that are not Resource Collections.	<code>http://resourcecollection?only</code>
\$expand	Include data from hyperlinks in the resource inline within the current payload, depending	<code>http://resourcecollection?\$expand=. (\$levels=1)</code>

Attribute	Description	Example
	on the value of the expand	
\$select	Include a subset of the properties of a resource based on the expression specified in the query parameters for this option.	<code>http://resource?\$select=SystemType,Status</code>
\$filter	Include a subset of the members of a collection based on the expression specified in the query parameters for this option	<code>http://resourcecollection?\$filter=SystemType eq 'Physical'</code>
excerpt	Include a subset of the properties of a resource based on the presence of the 'Excerpt' schema annotation in its definition. If no excerpt properties exist, the entire resource shall be returned.	<code>http://resource?excerpt</code>

- Services should support the \$top, \$skip, "only", and "excerpt" query parameters.
- Service may support the \$expand, \$filter and \$select query parameters.
- When the service supports query parameters, the service shall include the ProtocolFeaturesSupported object in the service root.
- 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.
- Implementation shall return the [400](#), Bad Request, status code for any query parameters containing values that are invalid, or applied to parameters with no values defined (e.g., 'excerpt' or 'only').
- Implementations shall ignore unknown or unsupported query parameters that do not begin with

"\$".

- Query parameters shall only be supported on GET operations.
- The contents of the response body shall be as if the query parameters were evaluated in the following order:
 - Prior to service side pagination: \$filter, \$skip, \$top
 - After applying any service side pagination: \$expand, \$select
- Implementations may accept multiple query parameters in a single request using "&" to separate each query parameter

Query parameters for Resource Collections

When the resource addressed is a Resource Collection, the client may use the \$top and \$skip paging query options to specify that a subset of the Members of that Resource Collection be returned. These paging query options apply specifically to the "Members" array property within a Resource Collection.

By adding the "only" parameter when addressing a Resource Collection, the client will achieve more efficient retrieval of collection members. If the target collection contains exactly one member (in the Members[] array), a GET operation including the "only" query parameter shall return the member's resource as if the GET operation was for that resource. If the collection contains either zero members or more than one member, the collection resource is returned as expected.

Query parameters for Expand

The \$expand parameter indicates to the implementation that it should include a hyperlink as well as the contents of that hyperlink in the current response as if a GET had been performed and included inline with that hyperlink. In CSDL terms, any Entries associated with an Entity or Collection of Entities through the use of NavigationProperty is capable of being expanded and thus included in the response body. The \$expand query parameter has a set of possible values that determine which hyperlinks (Navigation Properties) are to be expanded.

The following table represents the Redfish allowable values that shall be supported for \$expand if \$expand is implemented:

Value	Description	Example
* (asterisk)	Indicates all hyperlinks (Navigation Properties) shall be expanded if expand is supported.	<code>http://resourcecollection?\$expand=*</code>
. (period)	Indicates all subordinate	<code>http://resourcecollection?\$expand=.</code>

Value	Description	Example
	hyperlinks (Navigation Properties) shall be expanded if expand is supported. Subordinate hyperlinks are those that are directly referenced (i.e., not in the Links Property section of the resource).	
~ (tilde)	Indicates all dependent hyperlinks (Navigation Properties) shall be expanded if expand is supported. Dependent hyperlinks are those that are not directly referenced (i.e., in the Links Property section of the resource).	<code>http://resourcecollection?\$expand=~</code>
\$levels	Indicates how many levels the service should cascade the expand operation. Thus a \$levels=2 will not only expand the current resource (level=1) but also the expanded resource (level=2).	<code>http://resourcecollection?\$expand=.\$levels=2)</code>

Examples of the use of expand might be:

- GET of a SoftwareInventoryCollection. By including expand, the client can request multiple SoftwareInventory resources in a single request instead of fetching them one at a time.
- GET of a ComputerSystem. By specifying levels, collection such as Processors, Memory and other resources could be included in a single GET request.
- GET all of the UUIDs in the ComputerSystem collection. This would be combining \$select with \$expand on the URI. The syntax for this would be GET /redfish/v1/Systems?\$select=UUID&\$expand=.((\$levels=1))

When performing \$expand, Services may omit some of the properties of the referenced resource.

When using expand, clients should be aware that the payload may increase beyond what can be sent in a single response. If a service is unable to return the payload due to its size, it shall return HTTP Status code [507](#).

It should be noted that Resources containing Referenceable Members that are already expanded using the `AutoExpand` annotation, such as the "Temperature" object found in the "Thermal" Resource, are considered part of the current Resource, and therefore are at the same level as that Resource being queried.

Any other supported syntax for \$expand is outside the scope of this specification.

Query parameters for Select

The \$select parameter indicates to the implementation that it should return a subset of the properties of the resource based on the value of the select clause. The \$select clause shall not affect the resource itself. The value of the \$select clause is a comma-separated list of the properties to be returned in the body of the response. The syntax to represent properties in complex types shall be the property names concatenated with a slash ("/"). Note that the default behavior when the select option is not specified is to return all properties.

An example of the use of select might be:

- GET /redfish/v1/Systems/1\$select=Name,SystemType,Status/State

When performing \$select, Services shall return all of the requested properties of the referenced resource. The "[@odata.id](#)" and "[@odata.type](#)" properties shall be in the response payload and contain the same values as if \$select was not performed. If the "[@odata.context](#)" property is supported, it shall be in the response payload and should be in the recommended format specified by the [Context property section](#). If the "[@odata.etag](#)" property is supported, it shall be in the response payload and contain the same values as if \$select was not performed.

Any other supported syntax for \$select is outside the scope of this specification.

Query parameters for Filter

The \$filter parameter indicates to the implementation that it should include a subset of the members of a

collection based on the expression specified as the value of the filter clause. The \$filter query parameter is a set of properties and literals with an operator. A literal value can be a string enclosed in single quotes, a number, or a boolean value. The service should reject \$filter requests if the literal value does not match the data type for the property specified by responding with HTTP Status code [400](#). The \$filter section of the OData ABNF components specification contains the grammar for the allowable syntax of the \$filter query parameter with the additional restriction that only built-in filter operations (expressed below) are supported.

The following table represents the Redfish allowable operators that shall be supported for \$filter if \$filter is implemented:

Value	Description	Example
eq	Equal comparison operator	ProcessorSummary/Count eq 2
ne	Not equal comparison operator	SystemType ne 'Physical'
gt	Great than comparison operator	ProcessorSummary/Count gt 2
ge	Greater than or equal to comparison operator	ProcessorSummary/Count ge 2
lt	Less than comparison operator	MemorySummary/TotalSystemMemoryGiB lt 64
le	Less than or equal to comparison operator	MemorySummary/TotalSystemMemoryGiB le 64
and	Logical and operator	ProcessorSummary/Count eq 2 and MemorySummary/TotalSystemMemoryGiB gt 64
or	Logical or operator	ProcessorSummary/Count eq 2 or ProcessorSummary/Count eq 4
not	Logical negation operator	not (ProcessorSummary/Count eq 2)
()	Precedence grouping operator	(Status/State eq 'Enabled' and Status/Health eq 'OK') or SystemType eq 'Physical'

Services shall use the following operator precedence when evaluating expressions: grouping, logical negation, relational comparison (gt, ge, lt, le that all have equal precedence), equality comparison (eq, ne that both have equal precedence), logical and, then logical or.

Any other supported syntax for \$filter is outside the scope of this specification. If the service receives a

\$filter query parameter that is not supported, it shall reject the request and return HTTP Status code [501](#).

6.4.2.4.2. Retrieving Resource Collections

Retrieving a Resource Collection is done by sending the HTTP GET method to the URI for that resource. The response includes properties of the Resource Collection including an array of its Members. 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 the Members array instances using paging.

- Clients shall not make assumptions about the URIs for the Members of a Resource Collection.
- Retrieved Resource Collections shall always include the [count](#) property to specify the total number of entries in its "Members" array.
- Regardless of paging, see [partial results](#), the total number of resources referenced by the Members array shall be returned in the [count](#) property.

6.4.3. 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 form of HTTP response headers.
- Services may support the HEAD method in order to verify hyperlink 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.

6.4.4. 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.

6.4.4.1. Success responses to modification requests

For Create operations, the response from the service after successful processing of the create request should be one of the following:

- HTTP Status code of [201](#) with a body containing the JSON representation of the newly created resource after the request has been applied.
- HTTP Status code of [202](#) with a location header set to the URI of a Task Monitor when the

processing of the request will require additional time to be completed. In this case a response with the HTTP code 201 and the created resource may be returned in response to request to the Task Monitor URI after processing is complete.

- HTTP Status code of [204](#) with empty payload in the event that service is unable to return a representation of the created resource.

For Update, Replace, or Delete operations, the response from the service after successful modification should be one of the following:

- HTTP Status code of [200](#) with a body containing the JSON representation of the targeted resource after the modification has been applied, or in the case of Delete operation, a representation of the deleted resource.
- HTTP Status code of [202](#) with a location header set to the URI of a Task Monitor when the processing of the modification will require additional time. In this case a response with the HTTP code 200 and the modified resource may be returned in response to request to the Task Monitor URI after processing is complete.
- HTTP Status code of [204](#) with empty payload in the event that service is unable to return a representation of the modified or deleted resource.

For details about success responses to Action requests, see [Action](#).

6.4.4.2. Failure responses to modification requests

Services may return an HTTP status code [405](#) if the specified resource exists but does not support the requested operation. Otherwise, if a client (4xx) or service (5xx) [status code](#) is returned, this indicates the service encountered an error and the resource shall not have been modified or created as a result of the operation.

6.4.4.3. Update (PATCH)

The PATCH method is the preferred method used to perform updates on preexisting resources. Changes to one or more properties within the resource addressed by the request Uri are sent in the request body. Properties not specified in the request body are not directly changed by the PATCH request. When modification is successful, the response may contain a representation of the resource after the update was done as described in [Success responses to modification requests](#). The implementation may reject the update operation on certain fields based on its own policies and in this case, not process any of the requested modifications.

- Services shall support the PATCH method to update properties within a resource.
- If the resource or all properties can never be updated, HTTP status code [405](#) shall be returned.
- If the client specifies a PATCH request against a Resource Collection, HTTP status code [405](#) should be returned.
- In the case of a request including modification to several properties, if one or more properties in the request can never be updated, such as when a property is read only, unknown, or unsupported, an HTTP status code of [200](#) shall be returned along with a representation of the

resource containing a Message [annotation](#) specifying the non-updatable properties. In this success case, other properties may be updated in the resource.

- In the case of a request modifying a single property, if the property in the request can never be updated, such as when the property is read only, unknown, or unsupported, an HTTP status code of [400](#) shall be returned along with a representation of the resource containing a Message [annotation](#) specifying the non-updatable property.
- The PATCH operation should be idempotent in the absence of outside changes to the resource, though the original ETag value may no longer match.
- Services may accept a PATCH with an empty JSON object. An empty JSON object in this context means no changes to the resource are being requested.

Services may have null entries for properties that are JSON arrays to show the number of entries a client is allowed to use in a PATCH request. Within a PATCH request, unchanged members within a JSON array may be specified as empty JSON objects, and clearing members within a JSON array may be specified with null.

OData annotations (such as [resource identifiers](#), [type](#), and [etag](#)) shall be ignored by the service on Update. This includes any annotations matching the forms "*PropertyName*@odata.*TermName*" or "@odata.*TermName*", where *PropertyName* is the name of the property being annotated, and *TermName* is the specific OData annotation term. If an Update request only contains OData annotations, the service should return the NoOperation message defined in the Base Message Registry. In order to gain the protection semantics of an ETag, the If-Match or If-None-Match header shall be used for that protection and not the @odata.etag property value.

6.4.4.4. Replace (PUT)

The PUT method is used to completely replace a resource. Properties omitted from the request body, required by the resource definition, or normally supplied by the Service, may be added by the Service to the resulting resource. When the replace operation is successful, the response may contain a representation of the resource after the replacement was done as described in [Success responses to modification requests](#).

- Services may support the PUT method to replace a resource in whole.
- If a service does not implement this method, a status code [405](#) shall be returned.
- Services may reject requests that do not include properties required by the resource definition (schema).
- Services should return status code [405](#) if the client specifies a PUT request against a Resource 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.

6.4.4.5. 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. When the create operation is successful, the response may contain a representation of the resource after the update was done as described in [Success responses to modification requests](#). The body of the create request contains a representation of the object to be created. The service may ignore any service controlled properties (e.g., Id), forcing those properties to be overridden by the service. Additionally, the service shall set the Location header in the response to the URI of the newly created resource.

- Submitting a POST request to a Resource Collection is equivalent to submitting the same request to the Members property of that Resource Collection. Services that support adding Members to a Resource Collection shall support both forms.
 - For example, if a client is adding a new Member to the Resource Collection found at `"/redfish/v1/EventService/Subscriptions"`, it can do so by sending a POST request to either `"/redfish/v1/EventService/Subscriptions"` or `"/redfish/v1/EventService/Subscriptions/Members"`.
- 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.
- Services shall support POST operations on a URL that references a Resource Collection instance.
- Services shall also support POST operations on a URL that references an Action (see [Actions \(POST\)](#)).
- The POST operation shall not be idempotent.
- Services may allow the `"@Redfish.OperationApplyTime"` property to be included in the body of the request. See the [Operation apply time](#) clause for further information.

6.4.4.6. Delete (DELETE)

The DELETE method is used to remove a resource. When the delete operation is successful, the response may contain a representation of the resource after the deletion was done as described in [Success responses to modification requests](#).

- 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 should return HTTP status code [405](#) if the client specifies a DELETE request against a Resource Collection.
- Services may return HTTP status code [404](#) or a success code if the resource has already been deleted.
- Services may allow the `"@Redfish.OperationApplyTime"` property to be included in the body of the request. See the [Operation Apply Time](#) section for further information.

6.4.4.7. 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.
- Services may allow the "@Redfish.OperationApplyTime" property to be included in the body of the request. See the [Operation apply time](#) clause for further information.

Actions are requested on a resource by sending the HTTP POST method to the URI of the action. The "target" property within the [actions property](#) of a resource shall contain the URI of the action. The URI of the action shall be in the form of:

```
*ResourceUri*/Actions/*QualifiedActionName*
```

where

- *ResourceUri* is the URI of the resource that 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 qualified name of the action, including namespace.

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. The resource may provide a separate ActionInfo resource to describe the parameters and values supported by a particular instance or implementation. The ActionInfo resource is specified using the "@Redfish.ActionInfo" annotation, containing a URI to the ActionInfo resource for the action.

For instance, if a Redfish Schema document `http://redfish.dmtf.org/schemas/v1/ComputerSystem_v1.xml` defines a Reset action in the ComputerSystem namespace, bound to the ComputerSystem.v1_0_0.Actions type, such as this example:

```
<Schema Namespace="ComputerSystem">
  ...
  <Action Name="Reset" IsBound="true">
    <Parameter Name="Resource" Type="ComputerSystem.v1_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",
          "ForceRestart",
          "Nmi",
          "ForceOn",
          "PushPowerButton"
        ]
      }
    },
    ...
  }

```

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

```

POST /redfish/v1/Systems/1/Actions/ComputerSystem.Reset HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
  "ResetType": "On"
}

```

Using the same Reset example, a computer system resource may utilize an ActionInfo annotation and resource to convey the allowable values:

```

{
  "Actions": {
    "#ComputerSystem.Reset": {
      "target": "/redfish/v1/Systems/1/Actions/ComputerSystem.Reset",
      "@Redfish.ActionInfo": "/redfish/v1/Systems/1/ResetActionInfo"
    }
  },
  ...
}

```

Then, the ResetActionInfo resource would contain a more detailed description of the parameters and the supported values.

```

{
  "@odata.context": "/redfish/v1/$metadata#ActionInfo.ActionInfo",
  "@odata.id": "/redfish/v1/Systems/1/ResetActionInfo",
  "@odata.type": "#ActionInfo.v1_0_0.ActionInfo",
  "Id": "ResetActionInfo",
  "Name": "Reset Action Info",
  "Parameters": [
    {
      "Name": "ResetType",
      "Required": true,
      "DataType": "String",
      "AllowableValues": [
        "On",
        "ForceOff",
        "ForceRestart",
        "Nmi",
        "ForceOn",
        "PushPowerButton"
      ]
    }
  ]
}

```

In cases where the processing of the Action may require extra time to be completed, the service may respond with an HTTP Status code of [202](#) with a location header in the response set to the URI of a Task Monitor. Otherwise the response from the service after processing an Action may return a response with one of the following HTTP Status codes:

- HTTP Status Code [200](#) indicates the Action request was successfully processed, with the JSON message body as described in [Error Responses](#) and providing a message indicating success or any additional relevant messages.
 - If the Action was successfully processed and completed without errors, warnings, or other notifications for the client, the service should use the Success message defined in the Base Message Registry in the response body for the "code" property.
- HTTP Status Code [204](#) indicates the Action is successful and is returned without a message body.
- In the case of an error, a valid HTTP status code in the range 400 or above indicating an error was detected and the Action was not processed. In this case, the body of the response may contain a JSON object as described in [Error Responses](#) detailing the error or errors encountered.

Actions may have required parameters as defined in the [Resource actions section](#). If a client does not provide all required parameters, the service shall reject the request with HTTP Status Code [400](#). If an Action does not have any required parameters, the service should accept an empty JSON object in the HTTP body for the Action request. If a client provides a parameter that the service does not support, the

service shall either reject the request with HTTP Status Code [400](#) or ignore the unknown parameters.

If an Action requested by the client will have no effect, such as performing a Reset of a ComputerSystem where the parameter "ResetType" is set to "On" and the ComputerSystem is already "On", the service should respond with an HTTP Status Code [200](#) and return the NoOperation message defined in the Base Message Registry.

Example successful Action response:

```
{
  "error": {
    "code": "Base.1.0.Success",
    "message": "Successfully Completed Request",
    "@Message.ExtendedInfo": [
      {
        "@odata.type": "#Message.v1_0_0.Message",
        "MessageId": "Base.1.0.Success",
        "Message": "Successfully Completed Request",
        "Severity": "OK",
        "Resolution": "None"
      }
    ]
  }
}
```

6.4.5. Operation apply time

Services may accept the "@Redfish.OperationApplyTime" annotation in the body of a [Create](#), [Delete](#), or [Action](#) operation. This is to give the client control as to when a particular operation is carried out. For example, if the client wants to delete a particular Volume resource, but can only safely do so when a reset is taking place, the client can use this annotation to instruct the service to delete the Volume on the next reset. If multiple operations are pending, the service shall process them in the order in which they were received.

Services that support the "@Redfish.OperationApplyTime" annotation for create and delete operations on a particular Resource Collection shall include the "@Redfish.OperationApplyTimeSupport" response annotation for the given Resource Collection. Below is an example response for a Resource Collection that supports the "@Redfish.OperationApplyTime" annotation in the request for create and delete operations.

```
{
  "@odata.context": "/redfish/v1/$metadata#VolumeCollection.VolumeCollection",
  "@odata.id": "/redfish/v1/Systems/1/Storage/SATAEmbedded/Volumes",
```

```

"@odata.type": "#VolumeCollection.VolumeCollection",
"Name": "Storage Volume Collection",
>Description": "Storage Volume Collection",
"Members@odata.count": 2,
"Members": [
  {
    "@odata.id": "/redfish/v1/Systems/1/Storage/SATAEmbedded/Volumes/1"
  },
  {
    "@odata.id": "/redfish/v1/Systems/1/Storage/SATAEmbedded/Volumes/2"
  }
],
"@Redfish.OperationApplyTimeSupport": {
  "@odata.type": "#Settings.v1_2_0.OperationApplyTimeSupport",
  "SupportedValues": [ "Immediate", "OnReset" ]
}
}

```

In the above example, a client is allowed to annotate their request body when performing a create operation on the VolumeCollection itself, or when performing a delete operation on the Volumes within the VolumeCollection. Below is a sample request to delete a particular Volume on the next reset.

```

DELETE /redfish/v1/Systems/1/Storage/SATAEmbedded/Volumes/2 HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
  "@Redfish.OperationApplyTime": "OnReset"
}

```

Services that support the "@Redfish.OperationApplyTime" annotation for a given action shall include the "@Redfish.OperationApplyTimeSupport" response annotation for the given action. Below is an example response for a ComputerSystem resource that supports the "@Redfish.OperationApplyTime" annotation in the request for the reset action.

```

{
  "@odata.context": "/redfish/v1/$metadata#ComputerSystem.ComputerSystem",
  "@odata.id": "/redfish/v1/Systems/1",
  "@odata.type": "#ComputerSystem.v1_5_0.ComputerSystem",
  "Actions": {
    "#ComputerSystem.Reset": {
      "target": "/redfish/v1/Systems/1/Actions/ComputerSystem.Reset",

```

```

    "ResetType@Redfish.AllowableValues": [
        "On",
        "ForceOff",
        "ForceRestart",
        "Nmi",
        "ForceOn",
        "PushPowerButton"
    ],
    "@Redfish.OperationApplyTimeSupport": {
        "@odata.type": "#Settings.v1_2_0.OperationApplyTimeSupport",
        "SupportedValues": [ "Immediate", "AtMaintenanceWindowStart" ],
        "MaintenanceWindowStartTime": "2017-05-03T23:12:37-05:00",
        "MaintenanceWindowDurationInSeconds": 600,
        "MaintenanceWindowResource": {
            "@odata.id": "/redfish/v1/Systems/1"
        }
    }
},
...
}

```

In the above example, a client is allowed to annotate their request body when performing a reset action operation on the ComputerSystem represented by the payload. Below is a sample request to perform a reset at the start of the next maintenance window.

```

POST /redfish/v1/Systems/1/Actions/ComputerSystem.Reset HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
    "ResetType": "ForceRestart",
    "@Redfish.OperationApplyTime": "AtMaintenanceWindowStart"
}

```

Services that support the "@Redfish.OperationApplyTime" for a given Resource Collection or action shall create a [Task](#), and respond with an HTTP Status code of [202](#) with a location header set to the URI of a Task resource, if the client's request body contains "@Redfish.OperationApplyTime" in the request.

The structure of the "@Redfish.OperationApplyTimeSupport" object and the values "@Redfish.OperationApplyTime" annotation are defined in the "Settings" Redfish Schema.

6.5. 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 resource that represents a Resource Collection.
- [Error Responses](#) - Top level JSON response providing additional information in the case of an HTTP error.

6.5.1. 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 7231	Describes the type of representation used in the message body. Services shall specify a Content-Type of <code>application/json</code> when returning resources as JSON, and <code>application/xml</code> when returning metadata as XML. <code>; charset=utf-8</code> shall be appended to the Content-Type if specified in the chosen media-type in the Accept header for the request.
Content-Encoding	No	RFC 7231	Describes the encoding that has been performed on the media type.
Content-Length	No	RFC 7231	Describes the size of the message body. An optional means of indicating size of the body uses

Header	Required	Supported Values	Description
			Transfer-Encoding: chunked, that 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 7232	An identifier for a specific version of a resource, often a message digest. ETags shall be included on responses to GETs of ManagerAccount objects.
Server	Yes	RFC 7231	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 clause on Link Headers .
Location	Conditional	RFC 7231	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 that create user sessions.
Cache-Control	Yes	RFC 7234	This header shall be supported and is meant to indicate whether a response can be cached or not.
Via	No	RFC 7230	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.
Max-Forwards	No	RFC 7231	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
Access-Control-Allow-Origin	Yes	W3C CORS , Section 5.1	Prevents or allows requests based on originating domain. Used to prevent CSRF attacks.
Allow	Yes	POST, PUT, PATCH, DELETE, GET,	Shall be returned with a 405 (Method Not Allowed) response to indicate the valid methods for the specified Request URI. Shall be returned with any GET or HEAD operation to indicate the other allowable operations for this resource.

Header	Required	Supported Values	Description
		HEAD	
WWW-Authenticate	Yes	RFC 7235 , Section 4.1	Required for Basic and other optional authentication mechanisms. See the Security clause for details.
X-Auth-Token	Yes	Opaque encoded octet strings	Used for authentication of user sessions. The token value shall be indistinguishable from random.
Retry-After	No	RFC 7231 , Section 7.1.3	Used to inform a client how long to wait before requesting the Task information again.

6.5.1.1. Link Header

The [Link Header](#) provides metadata information on the accessed resource in response to a HEAD or GET operation. The information can describe things such as hyperlinks from the resource and JSON Schemas that describe the resource.

Below is an example of the Link Headers of a ManagerAccount with a role of Administrator that has a Settings Annotation.

- The first Link Header is an example of a hyperlink that comes from the resource. It describes hyperlinks within the resource. This type of header is outside the scope of this specification.
- The second Link Header is an example of an Annotation Link Header as it references the JSON Schema that describes the annotation and does not have rel=describedby. This example references the public copy of the annotation on the DMTF's Redfish Schema repository.
- The third Link Header is an example for the JSON Schema that describes the actual resource.
- Note that the URL can reference an unversioned JSON Schema (since the @odata.type in the resource indicates the appropriate version) or reference the versioned JSON Schema (which according to previous normative statements would need to match the version specified in the @odata.type property of the resource).

```
Link: </redfish/v1/AccountService/Roles/Administrator>; path=/Links/Role
Link: <http://redfish.dmtf.org/schemas/Settings.json>
Link: </redfish/v1/JsonSchemas/ManagerAccount.v1_0_2.json>; rel=describedby
```

A Link Header containing rel=describedby shall be returned on GET and HEAD requests. If the

referenced JSON Schema is a versioned schema, it shall match the version contained in the value of the "@odata.type" property returned in this resource.

A Link Header satisfying Annotations should be returned on GET and HEAD requests.

6.5.2. 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 should return the extended error resource as described in this specification in the response body when a status code [400](#) or greater is returned. Services may return the extended error resource as described in this specification in the response body when other status codes are returned for those codes and operations that allow a response body.
- Extended error messages MUST NOT provide privileged information when authentication failures occur.

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

The following table lists HTTP status codes that have meaning or usage defined for a Redfish service, or are otherwise referenced by this specification. Other codes may be returned by the service as appropriate, and their usage is implementation-specific. See the Description column for usage 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 multipass 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	The request has been accepted for processing, but the processing has not been

HTTP Status Code	Description
Accepted	completed. The Location header shall be set to the URI of a Task Monitor 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 clause Error responses .
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, that 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 properties that work in a linked manner using

HTTP Status Code	Description
	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 hyperlink 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	The precondition (such as OData-Version, If-Match or If-Not-Modified headers) check failed.
415 Unsupported Media Type	The request specifies a Content-Type for the body that is not supported.
428 Precondition Required	The request did not provide the required precondition, such as an If-Match or If-None-Match header.
431 Request Header Field Too Large	The server is unwilling to process the request because either an individual header field, or all the header fields collectively, are too large.
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 clause Error Responses .
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. A service may use this response to indicate that the request URI is valid, but the service is performing initialization or other maintenance on the resource. It may also use this response to indicate the service itself is undergoing maintenance, such as finishing initialization steps after reboot

HTTP Status Code	Description
	of the service.
507 Insufficient Storage	The server is unable to build the response for the client due to the size of the response.

6.5.3. Metadata responses

Metadata describes resources, Resource 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.

6.5.3.1. 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>
```

6.5.3.1.1. Referencing other schemas

The service metadata shall include the namespaces for each of the Redfish resource types, along with the "RedfishExtensions.v1_0_0" namespace. These references may use the standard URI for the hosted Redfish Schema definitions (i.e., on <http://redfish.dmtf.org/schemas>) or a URI 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_v1.xml">
  <edmx:Include Namespace="AccountService"/>
  <edmx:Include Namespace="AccountService.v1_0_0"/>
</edmx:Reference>
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/ServiceRoot_v1.xml">
  <edmx:Include Namespace="ServiceRoot"/>
  <edmx:Include Namespace="ServiceRoot.v1_0_0"/>
```

```

</edmx:Reference>

...

<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/VirtualMedia_v1.xml">
  <edmx:Include Namespace="VirtualMedia"/>
  <edmx:Include Namespace="VirtualMedia.v1_0_0"/>
</edmx:Reference>
<edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/RedfishExtensions_v1.xml">
  <edmx:Include Namespace="RedfishExtensions.v1_0_0" Alias="Redfish"/>
</edmx:Reference>

```

The service's [Metadata Document](#) shall include an EntityContainer that defines the top-level resources and Resource Collections. An implementation may extend the ServiceContainer defined in the ServiceRoot schema for the implementation's [OData Service Document](#).

```

<edmx:DataServices>
  <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Service">
    <EntityContainer Name="Service" Extends="ServiceRoot.v1_0_0.ServiceContainer"/>
  </Schema>
</edmx:DataServices>

```

6.5.3.1.2. 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 Resource Collections.

```

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

```

6.5.3.2. 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 Resource Collections) or `"EntitySet"` for top-level Resource Collections, and a `"url"` property whose value is the relative URL for the top-level resource.

6.5.4. Resource responses

Resources are returned as JSON payloads, using the MIME type `application/json`. Resource property names match the case specified in the [Schema](#).

Responses that represent a single resource shall contain the ["Id" property](#) and the ["Name" property](#). Responses that represent a single resource may contain the ["Description" property](#).

See also [Resource Collection responses](#).

6.5.4.1. Context property

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

The context URL for a resource should be of the following form:

MetadataUrl##ResourceType

where

- *MetadataUrl* = the metadata url of the service (/redfish/v1/\$metadata)
- *ResourceType* = the fully qualified name of the unversioned resource type. For many Redfish implementations, this is just the namespace for the resource type concatenated with a period followed by the resource type again.

For example, the following context URL specifies that the result contains a single ComputerSystem resource:

```
{
  "@odata.context": "/redfish/v1/$metadata#ComputerSystem.ComputerSystem",
  ...
}
```

The context URL for a resource may be of the following form:

MetadataUrl##ResourcePath/\$entity

where

- *MetadataUrl* = the metadata url of the service (/redfish/v1/\$metadata)
- *ResourceType* = the fully qualified name of the unversioned resource type
- *ResourcePath* = the path from the service root to the singleton or Resource Collection containing the resource
- *\$entity* = a designator that the response is a single resource from either an entity set or specified by a navigation property.

While both formats are allowable, services should use the *MetadataUrl##ResourceType* format for the "@odata.context" property values as there are additional constraints required by the [OData-Protocol](#) when partial or expanded results are returned that pose an additional burden on services.

6.5.4.2. 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.

Resources identifiers shall be represented in JSON payloads as strings that conform to the rules for URI paths as defined in Section 3.3, Path of [RFC3986](#). Resources within the same authority as the request URI shall be represented according to the rules of path-absolute defined by that specification. That is, they shall always start with a single forward slash ("/"). Resources within a different authority as the request URI shall start with a double slashes ("//") followed by the authority and path to the resource.

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

6.5.4.3. Type property

All resources in a response shall include a type property named "@odata.type". If support of generic OData clients is desired, all [structured properties](#) in a response should include a type property named "@odata.type." The value of the type property shall be a URL fragment that specifies the type of the resource as defined within, or referenced by, the [metadata document](#) 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.

6.5.4.4. 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.

6.5.4.5. 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.Duration	String, formatted as specified in Duration Values
Edm.Decimal	Number, optionally containing a decimal point
Edm.Double	Number, optionally containing a decimal point and optionally containing an

Type	JSON Representation
	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.

6.5.4.5.1. 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*] (Z | (+ | - ) *hh*:*mm*)
```

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.

In cases where the time of day is unknown or serves no purpose, the service shall report "00:00:00Z" for the time of day portion of the DateTime value.

6.5.4.5.2. Duration values

Duration values shall be returned as JSON strings according to the ISO 8601 "duration" format of the form:

```
P [*Y*Y] [*M*M] [*W*W] [*D*D] [T [*h*H] [*m*M] [*s* [.S*] S]]
```

where

- Y is the number of years.
- M is the number of months.
- W is the number of weeks.
- D is the number of days.
- h is the number of hours.

- *m* is the number of minutes.
- *s* is the number of seconds.
- *S* is the fractional seconds.

Each field is optional and may contain more than one digit. Below are some examples:

- "P3D" specifies a duration of 3 days.
- "PT6H" specifies a duration of 6 hours.
- "PT10S" specifies a duration of 10 seconds.
- "PT0.001S" specifies a duration of 0.001 seconds.
- "PT1H30M" specifies a duration of 1 hour and 30 minutes.

6.5.4.6. 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.

Some structured properties inherit from the definition "Resource.v1_0_0.ReferenceableMember". Structured properties that follow this definition shall contain the ["MemberId" property](#) as well as the [resource identifier property](#).

Since the definition of structured properties can evolve over time, clients need to be aware of the inheritance model used by the different structured property definitions. For example, the "Location" definition found in Resource_v1.xml has gone through several iterations since the original introduction in the "Resource.v1_1_0" namespace, and each iteration inherits from the previous version so that existing references found in other schemas can leverage the new additions. There are two types of structured property references that need to be resolved: local references and external references.

A local reference is when a resource has a structured property within its own schema, such as "ProcessorSummary" in the "ComputerSystem" resource. In these cases, the [type property](#) for the resource is used as a starting point for resolving the structured property definition. The [version of the resource](#) can be stepped backwards until the latest applicable version is found. For example, if a service returns "#ComputerSystem.v1_4_0.ComputerSystem" as the resource type, a client can go backwards from "ComputerSystem.v1_4_0", to "ComputerSystem.v1_3_0", "ComputerSystem.v1_2_0", and so on, until the structured property definition of "ProcessorSummary" is found.

An external reference is when a resource has a property that references a definition found in a different schema, such as "Location" in the "Chassis" resource. In these cases, the latest version of the external schema file will be used as a starting point for resolving the structured property definition. For example, if the latest version of Resource_v1.xml is 1.6.0, a client can go backward from "Resource.v1_6_0", to "Resource.v1_5_0", "Resource.v1_4_0", and so on, until the structured property definition of "Location" is found.

6.5.4.7. Actions property

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

6.5.4.7.1. 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 shall be a JSON object containing a property named "target" whose value is a relative or absolute URL used to invoke the action. The JSON object for the action may contain a property named "title" whose value is a string containing the action's name. The "target" and "title" properties are defined in the [OData JSON Format](#) specification.

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",
    "title": "Computer System 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"  
}
```

6.5.4.7.2. 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.

6.5.4.8. 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 hyperlinks.

6.5.4.8.1. 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"  
    }  
  }  
}
```

```

    }
  }
}

```

6.5.4.8.2. Array of references to related resources

A reference to a set 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"
      }
    ]
  }
}

```

6.5.4.9. OEM property

OEM-specific properties are nested under an [OEM property](#).

6.5.4.10. Partial resource results

Responses representing a single resource shall not be broken into multiple results. See [partial results](#) for details about the format of these responses.

6.5.4.11. 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](#).

6.5.4.11.1. 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#SerialInterface.SerialInterface",
  "@odata.id": "/redfish/v1/Managers/1/SerialInterfaces/1",
  "@odata.type": "#SerialInterface.v1_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 operation failed."
    }
  ]
}

```

The value of the property is an array of [message objects](#).

6.5.4.11.2. 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#SerialInterface.SerialInterface",
  "@odata.id": "/redfish/v1/Managers/1/SerialInterfaces/1",
  "@odata.type": "#SerialInterface.v1_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 Contoso 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 resubmit the request if the operation failed."
      }
    ]
  }
}

```

The value of the property is an array of [message objects](#).

6.5.4.12. 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.

Services shall limit the annotation usage to the "odata", "Redfish", and "Message" namespaces. The

"odata" namespace is defined as part of the [OData JSON Format](#) specification. The "Redfish" namespace is an alias for the "RedfishExtensions.v1_0_0" namespace.

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.

6.5.5. Resource Collection responses

Resource Collections are returned as a JSON payloads, using the MIME type `application/json`. Resource property names match the case specified in the [Schema](#). Resource Collection schema shall be derived from the Resource Schema and thus Resource Collection responses shall contain the following properties:

- The ["Name" property](#)
- The [Resource Identifier property](#)
- The [Type property](#)
- An array of [Members](#)
- A [resource count](#)

Responses for Resource Collections may contain the following properties:

- The ["Description" property](#)
- The [Context property](#)
- An [Etag property](#)
- A [Next Link Property](#) for partial results
- An [OEM property](#)

Responses for Resource Collections shall not contain any property not explicitly defined in this clause of this specification.

6.5.5.1. Context property

Responses 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 Collection according to [OData-Protocol](#).

The context URL for a Resource Collection is of one of the following two forms:

MetadataUrl.#CollectionResourceType MetadataUrl.#CollectionResourcePath

where

- *MetadataUrl* = the metadata url of the service (`/redfish/v1/$metadata`)
- *CollectionResourceType* = the fully qualified name of the unversioned type of resources within

the Resource Collection

- *CollectionResourcePath* = the path from the service root to the Resource Collection

6.5.5.2. Count property

The total number of resources (members) available in the Resource Collection is represented through the count property. The count property shall be named "Members@odata.count" and its value shall be the total number of members available in the Resource Collection. This count is not affected by the \$top or \$skip [query parameters](#).

6.5.5.3. Members property

The Members of the Resource Collection of resources are returned as a JSON array, where each element of the array is a JSON object whose type is specified in the Redfish Schema document describing the containing type. The name of the property representing the members of the collection shall be "Members". The Members property shall not be null. Empty collections shall be returned in JSON as an empty array.

6.5.5.4. Next Link Property and partial results

Responses may contain a subset of the members of the full Resource Collection. For partial Resource Collections the response includes a Next Link Property named "Members@odata.nextLink". The value of the Next Link Property shall be an opaque URL to a resource, with the same @odata.type, containing the next set of partial members. The Next Link Property shall only be present if the number of Members in the Resource Collection is greater than the number of members returned, and if the payload does not represent the end of the requested Resource Collection.

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

6.5.5.5. Additional annotations

A JSON object representing a Resource Collection 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.

Services shall limit the annotation usage to the "odata", "Redfish", and "Message" namespaces. The "odata" namespace is defined as part of the [OData JSON Format](#) specification. The "Redfish" namespace

is an alias for the "RedfishExtensions.v1_0_0" namespace.

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.

6.5.6. 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.

A Redfish Service may provide multiple error responses in the HTTP response in order to provide the client with as much information about the error situation as it can. Additionally, the service may provide Redfish standardized errors, OEM-defined errors or both depending on the implementation's ability to convey the most useful information about the underlying error.

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.
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" : "#Message.v1_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 operation failed"
  },
  {
    "@odata.type" : "#Message.v1_0_0.Message",
    "MessageId": "Base.1.0.PropertyNotWritable",
    "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 operation failed"
  }
]
}
}

```

6.5.6.1. 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	A 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

Property	Description
	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 characters.

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 lightweight implementation (since large strings need not be included with the implementation).

"Base.1.0.GeneralError" should be used only in the code property of the error object if there is no better message or if there are multiple messages in the ExtendedInfo array. The use of "Base.1.0.GeneralError" as a MessageId in ExtendedInfo is discouraged. When it is used in ExtendedInfo, implementations are expected to include a Resolution property with this error to indicate how to resolve the problem.

7. Data model and Schema

One of the key tenets of the Redfish interface is the separation of protocol and data model. This clause 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](#).

7.1. Schema, registry and profile repository

All Redfish schemas, registries, and profiles published or re-published by the DMTF's Redfish Forum are available from the DMTF website <http://redfish.dmtf.org/> for download. The files are organized on the site in the following manner:

URL	Folder contents
redfish.dmtf.org/schemas	Current (most recent minor or errata) release of each schema file in CSDL, JSON Schema, and/or OpenAPI formats.
redfish.dmtf.org/schemas/v1	Durable URL for programmatic access to all v1.xx schema files. Every v1.xx minor or errata release of each schema file in CSDL, JSON Schema, OpenAPI formats.
redfish.dmtf.org/schemas/archive	Subfolders contain schema files specific to a particular version release.
redfish.dmtf.org/registries	Current (most recent minor or errata) release of each registry file.
redfish.dmtf.org/registries/v1	Durable URL for programmatic access to all v1.xx registry files. Every v1.xx minor or errata release of each registry file.
redfish.dmtf.org/registries/archive	Subfolders contain registry files specific to a particular version release.
redfish.dmtf.org/profiles	Current release of each Redfish Interoperability Profile (.json) file and associated documentation.
redfish.dmtf.org/profiles/v1	Durable URL for programmatic access to all v1.xx Redfish Interoperability Profile (.json) files.
redfish.dmtf.org/profiles/archive	Subfolders contain profile files specific to a particular profile version or release.

7.1.1. Schema, registry, and profile file naming conventions

Standard Redfish schema, registry, and profile files published in the repository, or those created by others and republished, shall follow a set of naming conventions. These conventions are intended to ensure consistent naming and eliminate naming collisions. Spaces shall not be part of file names.

7.1.1.1. CSDL (XML) schema file naming

Redfish CSDL schema files shall be named using the [Type Name](#) value, followed by "_v" and the major version of the schema. As a single CSDL schema file contains all minor revisions of the schema, only the major version is used in the file name. The file name shall be formatted as:

Type Name_vMajorVersion.xml

For example, version 1.3.0 of the Chassis schema would be named "Chassis_v1.xml".

7.1.1.2. JSON Schema file naming

Redfish JSON Schema files shall be named using the [Type identifiers](#), following the format:

ResourceType Name.vMajorVersion_MinorVersion_Errata.json

For example, version 1.3.0 of the Chassis schema would be named "Chassis.v1_3_0.json".

7.1.1.3. OpenAPI file naming

Redfish OpenAPI files shall be named using the [Type identifiers](#), following the format:

ResourceType Name.vMajorVersion_MinorVersion_Errata.yaml

For example, version 1.3.0 of the Chassis schema would be named "Chassis.v1_3_0.yaml".

7.1.1.4. Registry file naming

Redfish message or privilege registry files shall be named using the Registry name, following the format:

RegistryName.MajorVersion.MinorVersion.Errata.json

For example, version 1.0.2 of the Base message registry would be named "Base.1.0.2.json".

7.1.1.5. Profile file naming

The JSON document describing a Profile follows the Redfish Schema file naming conventions. The file name format for Profiles shall be formatted as:

ProfileName.vMajorVersion_MinorVersion_Errata.json

For example, version 1.2.0 of the BasicServer profile would be named "BasicServer.v1_2_0.json". The file name shall include the Profile name and Profile version matching those property values within the document.

7.1.1.6. OEM schema file naming

To avoid namespace collisions with current or future standard Redfish schema files, third parties defining Redfish schemas should prepend an organization name to the Namespace as the file name. For example, "ContosoDisk_v1.xml" or "ContosoDisk.v1.0.4.json".

7.1.2. Programmatic access to schema, registry, or profile files

Programs may access the Schema Repository using the durable URLs listed at the redfish.dmtf.org repository, as these folders will contain every released version of each file. Programs incorporating remote repository access should implement a local cache to reduce latency, program requirements for Internet access and undue traffic burden on the DMTF website.

7.2. 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 where the type is defined
- *TypeName* = the name of the type

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

```
*ResourceTypeName*.v*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 minor versions.
- *Errata* = integer: something in the prior version was broken and needed to be fixed.

An example of a valid type namespace might be "ComputerSystem.v1_0_0", and an example of a corresponding Type URI would be "#ComputerSystem.v1_0_0.ComputerSystem".

7.2.1. Type identifiers in JSON Schema

In JSON Schema definitions for Redfish schema, the JSON Schema-defined "title" property shall contain the Type URI used to identify the schema.

For example, the "title" property for the ComputerSystem schema would be:

```
"title": "#ComputerSystem.v1_0_0.ComputerSystem"
```

7.2.2. 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](#) clause for security implications of Data Model and Schema.

7.3. 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 uppercase with spaces between words shall be removed (e.g., PowerState, SerialNumber.)
- No underscores are used.
- Both characters are capitalized for two-character acronyms (e.g., IPAddress, RemoteIP).
- Only the first character of acronyms with three or more characters is capitalized, except the first word of a Pascal-cased identifier (e.g., Wwn, VirtualWwn). If a single acronym (or mixed-case name) is used alone as a name (e.g. RDMA, iSCSI, SNMP), then the value should follow the capitalization commonly used for that name.

Exceptions are allowed for the following cases:

- Well-known technology names like "iSCSI" (e.g., "iSCSITarget")
- Product names like "iLO"
- Well-known abbreviations or acronyms
- OEM appears as "Oem" in resource or property names (alone or as a portion of a name), but should be "OEM" when used alone as a constant.
- Enumeration values should be named for readability as they may appear unmodified on user interfaces, whereas property or resource names should follow the conventions above and strive for consistency in naming with existing Redfish resources or properties.

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

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

7.4. Localization considerations

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

7.5. Schema definition

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

7.5.1. Common annotations

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

7.5.1.1. 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>.

7.5.1.2. 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. Where the LongDescriptions in the Redfish schema files contain "shall" references, the service shall be required to conform with the statement.

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

7.5.2. 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 and OpenAPI representations of the Redfish Schema capable of validating JSON payloads.

7.5.2.1. Schema modification rules

Schema referenced from the implementation, either from the OData Service Document or from the JSON Schema File representations, may vary from the canonical definitions of those Schema defined by the Redfish Schema or other entities, provided they adhere to the rules in the list below. Clients should take this into consideration when attempting operations on the resources defined by schema.

- Modified schema may constrain a read/write property to be read only.
- Modified schema may remove properties.
- Modified schema may change any "Reference Uri" to point to Schema that adheres to the modification rules.
- Other modifications to the Schema shall not be allowed.

7.5.2.2. Schema version requirements

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>
```

7.5.2.3. 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 and OpenAPI representations, 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.v1_0_0` 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="Measures"/>
  </edmx:Reference>
  <edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/RedfishExtensions_v1.xml">
    <edmx:Include Namespace="RedfishExtensions.v1_0_0" Alias="Redfish"/>
  </edmx:Reference>
  <edmx:Reference Uri="http://redfish.dmtf.org/schemas/v1/Resource_v1.xml">
    <edmx:Include Namespace="Resource"/>
    <edmx:Include Namespace="Resource.v1_0_0"/>
  </edmx:Reference>

```

7.5.2.4. 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.v1_0_0">

    <!-- Type definitions go here -->

  </Schema>
</edmx:DataService>

```

7.5.3. 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.v1_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.v1_0_0.Resource">
  <Annotation Term="OData.Description" String="This is the description of TypeA."/>
  <Annotation Term="OData.LongDescription" String="This is the specification of
TypeA."/>

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

</EntityType>

```

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

7.5.4. Resource capabilities

The capabilities of a resource are expressed using the `Capabilities.InsertRestrictions`, `Capabilities.UpdateRestrictions`, and `Capabilities.DeleteRestrictions` terms.

- `Capabilities.InsertRestrictions` is used to show whether or not a client is able to perform a POST on the resource.
- `Capabilities.UpdateRestrictions` is used to show whether or not a client is able to perform a PATCH or PUT on the resource.
- `Capabilities.DeleteRestrictions` is used to show whether or not a client is able to perform a DELETE on the resource.
- A service may only implement a subset of the capabilities that are set to true.

```

<EntityType Name="ManagerAccount" BaseType="Resource.v1_0_0.Resource"
Abstract="true">

  <!-- Other definitions for the EntityType go here -->

  <Annotation Term="Capabilities.InsertRestrictions">
    <Record>
      <PropertyValue Property="Insertable" Bool="false"/>
    </Record>
  </Annotation>
  <Annotation Term="Capabilities.UpdateRestrictions">
    <Record>
      <PropertyValue Property="Updatable" Bool="true"/>
      <Annotation Term="OData.Description" String="Manager Accounts can be updated
to change the password and other writable properties."/>
    </Record>
  </Annotation>
  <Annotation Term="Capabilities.DeleteRestrictions">
    <Record>

```

```

        <PropertyValue Property="Deletable" Bool="true"/>
        <Annotation Term="OData.Description" String="Manager Accounts are removed with
a Delete operation."/>
    </Record>
</Annotation>
</EntityType>

```

In the above example, the `Capabilities.InsertRestrictions` term has the `Insertable` property set to false, meaning a client is not able to perform a POST on the resource. It also uses the `Capabilities.UpdateRestrictions` term with the `Updatable` property set to true, meaning a client is able to perform a PATCH or PUT on the resource, assuming the client has the right privilege to perform these operations. It also uses the `Capabilities.DeleteRestrictions` term with the `Deletable` property set to true, meaning a client is able to perform a DELETE on the resource, assuming the client has the right privilege to perform this operation.

7.5.5. Resource URI pattern definitions

The URI patterns allowed for a given Redfish Resource are expressed using the `Redfish.Uris` annotation within the `EntityType` element.

```

<EntityType Name="ManagerAccount" BaseType="Resource.v1_0_0.Resource"
Abstract="true">

    <!-- Other definitions for the EntityType go here -->

    <Annotation Term="Redfish.Uris">
        <Collection>
            <String>/redfish/v1/AccountService/Accounts/{ManagerAccountId}</String>
        </Collection>
    </Annotation>
</EntityType>

```

In the above example, the `Redfish.Uris` annotation describes a single URI pattern. This definition means the client can expect to find the `ManagerAccount` resource at the URI `/redfish/v1/AccountService/Accounts/{ManagerAccountId}`, if the resource is implemented by the service.

Items between the `{` and `}` characters are treated as identifiers within the URI for given instances of a Redfish resource. Clients interpret this as a string to be replaced in order to access a given resource. A URI pattern may contain multiple identifier terms to support multiple levels of nested Resource Collections. The identifier term in the URI pattern shall match the "Id" string property for the corresponding Redfish resource, or the "MemberId" string property for the corresponding object within a Redfish resource. Using the above example, "{ManagerAccountId}" would be replaced by the "Id"

property of the corresponding ManagerAccount resource. If the "Id" property for a given ManagerAccount resource is "John", then the full URI for that resource would be "/redfish/v1/AccountService/Accounts/John".

All Redfish Resources and Redfish Resource Collections shall be annotated with `Redfish.Uris`.

All Redfish Resources and Redfish Resource Collections implemented by a service shall match the URI pattern described by the `Redfish.Uris` annotation for their given `EntityType` definition.

7.5.6. 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.

Property names in the Request and Response JSON Payload shall match the casing of the value of the `Name` attribute.

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="OData.Description" String="This is a property of TypeA."/>
  <Annotation Term="OData.LongDescription" String="This is the specification of
Property1."/>
  <Annotation Term="OData.Permissions" EnumMember="OData.Permission/Read"/>
  <Annotation Term="Redfish.Required"/>
  <Annotation Term="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 `OData.Permission/Read`.

Properties that are writable are annotated with the [Permissions annotation](#) with a value of `OData.Permission/ReadWrite`.

- A service may implement a writable property as read-only.

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](#).

7.5.6.1. Property types

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.

7.5.6.1.1. 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

7.5.6.1.2. 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 that specifies the base type, if any.

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

```
<ComplexType Name="PropertyTypeA">
  <Annotation Term="OData.Description" String="This is type used to describe a
structured property."/>
  <Annotation Term="OData.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.

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

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

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

7.5.6.1.4. 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.

7.5.6.2. 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 value of `"False"`, shall 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>.

7.5.6.3. 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">
```

7.5.6.4. Permissions of properties

The `Permissions` annotation term can be applied to a property with the value of `OData.Permission/Read` in order to specify that it is read-only.

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

The `Permissions` annotation term can be applied to a property with the value of `OData.Permission/ReadWrite` in order to specify that it is writable.

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

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

7.5.6.5. Required properties

The `Required` annotation is used to specify that a property is required to be supported by services. Required properties shall be annotated with the `Required` annotation. All other properties 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_v1.xml.

7.5.6.6. 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_v1.xml.

7.5.6.7. 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.

The value of the annotation should be a string that contains the case-sensitive "(c/s)" symbol of the unit of measure as listed in the [Unified Code for Units of Measure \(UCUM\)](#), unless the symbolic representation does not reflect common usage (e.g., "RPM" is commonly used to report fan speeds in revolutions-per-minute, but has no simple UCUM representation). For units with prefixes (e.g., Mebibyte (1024² bytes), which have the UCUM prefix "Mi" and symbol "By"), the case-sensitive "(c/s)" symbol for the prefix as listed in UCUM should be prepended to the unit symbol. For values that also include rate information (e.g., megabits per second), the rate unit's symbol should be appended and use a "/" slash character as a separator (e.g., "Mbit/s").

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

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

7.5.7. 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="OData.Description" String="This property references a related
resource."/>
  <Annotation Term="OData.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="OData.Description" String="This property represents a collection
of related resources."/>
  <Annotation Term="OData.LongDescription" String="This is the specification of the
related property."/>
  <Annotation Term="OData.AutoExpandReferences"/>
</NavigationProperty>
```

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

7.5.7.1. 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 policies, sensors) for this chassis."/>
  <Annotation Term="OData.LongDescription" String="The value of this property shall
be a reference to the resource that represents the power characteristics of this
chassis and shall be of type Power."/>
  <Annotation Term="OData.AutoExpandReferences"/>
```

```
</NavigationProperty>
```

7.5.7.2. 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>.

7.5.7.3. 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 Resource Collection in responses. Reference properties annotated with this term shall be expanded by the service, even if not requested by the client.

```
<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>.

7.5.8. 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 [structured 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. Parameters containing the term `Nullable="false"` are required to be provided in the `Action` request.

```
<Action Name="MyAction" IsBound="true">
  <Parameter Name="Thing" Type="MyType.Actions"/>
  <Parameter Name="Parameter1" Type="Edm.Boolean"/>
  <Parameter Name="Parameter2" Type="Edm.String" Nullable="false"/>
</Action>
```

In the above example, three parameters are defined:

- Thing: This is the binding parameter that is not provided in the request by the client
- Parameter1: A boolean parameter used in the client payload for the request
- Parameter2: A string parameter used in the client payload for the request and is also required to be provided by the client

7.5.9. Resource extensibility

Companies, OEMs, and other organizations can define additional [properties](#), [hyperlinks](#), and [actions](#) for common Redfish resources using the `Oem` property on resources, the [Links Property](#), 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.

7.5.9.1. Oem property

In the context of this clause, 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 predefined 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 Namespace="Contoso.v1_2_0">
  ...
  <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 clauses.

```
{
  "Oem": {
    "Contoso": {
      "@odata.type": "#Contoso.v1_2_0.AnvilType1",
      "slogan": "Contoso anvils never fail",
      "disclaimer": "* Most of the time"
    }
  },
  ...
}
```

7.5.9.2. Oem property format and content

Each property contained within the [Oem property](#) shall be a JSON object. The name of the object (property) shall uniquely identify the OEM or organization that defines the properties contained by that object. This is described in more detail in the following clause. The OEM-specified object 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.

7.5.9.3. 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), with all dot '.' separators replaced with underscores '_', *or an IANA-assigned Enterprise Number prefaced with "EID"*. DEPRECATED: 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 an underscore and any additional string to allow further creation of namespaces of OEM-specified objects as desired by the OEM, e.g., "Contoso_xxxx" or "EID_412_xxxx". The form and meaning of any text that follows the trailing underscore 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 suborganization "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 "Contoso_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.

7.5.9.4. URIs for OEM resources

Companies, OEMs, and other organizations can define additional resources and link to them using a

`NavigationProperty` from an [Oem property](#) found in a standard Redfish Resource. In order to avoid URI collisions with other OEM resources and future Redfish standard resources, the URIs for OEM resources shall be in the form of:

```
*BaseUri*/Oem/*OemName*/*ResourceName*
```

where

- *BaseUri* is the URI segment of the standard Redfish Resource where the "Oem" property is used.
- *OemName* is the name of the OEM, that follows the same naming as defined in the [Oem property format and content section](#).
- *ResourceName* is the name of the resource defined by the OEM.

For example, if Contoso defined a new resource called "AccountServiceMetrics" to be linked via the "Oem" property found at the URI "/redfish/v1/AccountService", the OEM resource would have the URI "/redfish/v1/AccountService/Oem/Contoso/AccountServiceMetrics".

7.5.10. 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": "#Contoso.v1_2_1.AnvilTypes1",
      "slogan": "Contoso anvils never fail",
      "disclaimer": "* Most of the time"
    },
    "Contoso_biz": {
      "@odata.type": "#ContosoBiz.v1_1.RelatedSpeed",
      "speed": "ludicrous"
    },
    "EID_412_ASB_123": {
      "@odata.type": "#OtherSchema.v1_0_1.powerInfoExt",
      "readingInfo": {
        "readingAccuracy": "5",
        "readingInterval": "20"
      }
    },
    "Contoso_customers_customerA": {
      "@odata.type": "#ContosoCustomer.v2015.slingPower",
```



```

        "AvailableTargets" : [ "rabbit", "duck", "runner" ],
        "launchPowerOptions" : [ "low", "medium", "eliminate" ],
        "powerSetting" : "eliminate",
        "targetSetting" : "rabbit"
    }
},
...
}

```

7.5.10.1. OEM 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>

```

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

```

{
  "Actions": {
    "Oem": {
      "#Contoso.Ping": {
        "target": "/redfish/v1/Systems/1/Actions/Oem/Contoso.Ping"
      }
    }
  },
  ...
}

```

The URI of the OEM action in the "target" property shall be in the form of:

ResourceUri*/Actions/Oem/*QualifiedActionName

where

- *ResourceUri* is the URI of the resource that supports invoking the action.
- "Actions" is the name of the property containing the actions for a resource.
- "Oem" is the name of the OEM property within the Actions property.
- *QualifiedActionName* is the qualified name of the action, including namespace.

7.6. Common Redfish resource properties

This clause contains a set of common properties across all Redfish resources. The property names in this clause 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_v1.xml. For JSON Schema representations, this is in Resource.v1_0_0.json. For OpenAPI representations, this is in Resource.v1_0_0.yaml.

7.6.1. Id

The Id property of a resource uniquely identifies the resource within the Resource Collection that contains it. The value of Id shall be unique across a Resource Collection.

7.6.2. Name

The Name property is used to convey a human-readable moniker for a resource. The type of the Name property shall be string. The value of Name is NOT required to be unique across resource instances within a Resource Collection.

7.6.3. Description

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

7.6.4. MemberId

The MemberId property of an object within a resource uniquely identifies the object within an array that contains it. The value of MemberId shall be unique across the array within the resource.

7.6.5. 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.

7.6.6. Links

The [Links Property](#) represents the hyperlinks 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.

7.6.7. Members

The [Members](#) property of a Resource Collection identifies the members of the collection.

7.6.8. RelatedItem

The [RelatedItem](#) property represents hyperlinks 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 subelements 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).

7.6.9. Actions

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

7.6.10. OEM

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

7.7. 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 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

7.7.1. 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 and/or PUT to a separate [Settings resource](#). For resources that can be modified immediately, the Allow header shall contain PATCH and/or PUT in the GET response. When a resource is read-only, the Allow header shall not contain PATCH or PUT in the GET response.

7.7.2. Settings

A Settings resource represents the future state and configuration of the resource. For resources that support a future state and configuration, the response shall contain a property with the "@Redfish.Settings" annotation. While the resource represents the current state, the Settings resource represents the future intended state.

The Settings resource includes several properties to help clients monitor when the resource is consumed by the service and determine the results of applying the values, which may or may not have been successful. The Messages property is a collection of Messages that represent the results of the last time the values of the Settings resource were applied. The ETag property contains the ETag of the Settings resource that was last applied. The Time property indicate the time when the Settings resource was last applied.

Below is an example body for a resource that supports a Settings resource. A client is able to locate the URI of the Settings resource using the "SettingsObject" property.

```
{
  "@Redfish.Settings": {
    "@odata.type": "#Settings.v1_0_0.Settings",
    "SettingsObject": {
      "@odata.id": "/redfish/v1/Systems/1/Bios/SD"
    },
    "Time": "2017-05-03T23:12:37-05:00",
    "ETag": "A89B031B62",
    "Messages": [
      {
        "MessageId": "Base.1.0.PropertyNotWritable",
        "RelatedProperties": [
          "#/Attributes/ProcTurboMode"
        ]
      }
    ]
  }
}
```

```

    ]
  },
  ...
}

```

The values in the Settings resource are applied to the resource either directly, such as with a POST of an action (such as Reset) or a PUT/PATCH request, or indirectly, such as when a user reboots a machine outside of the Redfish Service. A client may indicate its preference on when to apply the future configuration by including the "@Redfish.SettingsApplyTime" annotation in the request body when configuring the Settings resource. If a service supports configuring when to apply the future settings, the response body that represents the Settings resource shall contain a property with the "@Redfish.SettingsApplyTime" annotation. See properties defined in the "Settings" Redfish Schema for details.

Below is an example request body that shows a client configuring when the values in the Settings resource are to be applied:

```

{
  "@Redfish.SettingsApplyTime": {
    "@odata.type": "#Settings.v1_1_0.PreferredApplyTime",
    "ApplyTime": "OnReset",
    "MaintenanceWindowStartTime": "2017-05-03T23:12:37-05:00",
    "MaintenanceWindowDurationInSeconds": 600
  },
  ...
}

```

7.7.3. 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.

7.7.4. 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.

7.8. Special resource situations

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

7.8.1. Absent resources

Resources may be either absent or their state unknown at the time a client requests information about that resource. For resources that represent removable or optional components, absence provides useful information to clients, as it indicates a capability (e.g., an empty PCIe slot, DIMM socket, or drive bay) that would not be apparent if the resource simply did not exist. This also applies to resources that represent a limited number of items or unconfigured capabilities within an implementation, but this usage should be applied sparingly and should not apply to resources limited in quantity due to arbitrary limits (e.g., an implementation that limits "SoftwareInventory" to a maximum of 20 items should not populate 18 absent resources when only two items are present).

For resources that provide useful data in an absent state, and where the URI is expected to remain constant (such as when a DIMM is removed from a memory socket), the resource should exist, and should represent the State property of the Status object as "Absent". In this circumstance, any required properties for which there is no known value shall be represented as null. Properties whose support is based on the configuration choice or the type of component installed (and therefore unknown while in the Absent state), should not be returned. Likewise, subordinate resources for a absent resource should not be populated until their support can be determined (e.g., the "Power" and "Thermal" resources under a "Chassis" resource should not exist for an absent Chassis).

Client software should be aware that when absent resources are later populated, the updated resource may represent a different configuration or physical item, and previous data (including read-only properties) obtained from that resource may be invalid. For example, the "Memory" resource shows details about a single DIMM socket and the installed DIMM. When that DIMM is removed, the Memory resource remains to indicate the empty DIMM socket (with an "Absent" State). Later, an upgraded DIMM is installed, and the Memory resource then contains data about this new DIMM, which could have completely different characteristics.

7.8.2. 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 Redfish Service may reference 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 that specifies limits to the element. This is an override for the schema only to be used when necessary.

A Redfish Service 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.

8. Service details

8.1. Eventing

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

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

8.1.1. Event subscription types

The Redfish Service requires a client or administrator to create subscriptions to receive events. There are two methods of creating a subscription: directly by sending an HTTP POST to the subscription collection, or indirectly when a [Server-Sent Events \(SSE\)](#) connection is opened for the Event Service.

8.1.1.1. POST to the subscription collection method

The client locates the Event Service by traversing the Redfish Service interface. The Event Service is found off of the Service Root as described in the Redfish Schema for that service.

When the service has been discovered, clients subscribe to messages by sending a HTTP POST to the URI of the Resource Collection for "Subscriptions" in the Event Service. 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. The specific syntax of the subscription body is found in the Redfish Schema definition for "EventDestination".

On success, the Event Service shall return an 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 conforming to the "EventDestination" schema. 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.

- 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 preconfigured thresholds.
- Services shall respond to a request to create a subscription with an error if the body of the request is conflicting. For instance, if parameters in the request are not supported, an HTTP status 400 shall be returned.
- Services shall respond to a request to create a subscription with an error if the body of the request contains both RegistryPrefixes and MessageIds, and shall return an HTTP status code of 400. These properties are considered mutually exclusive.
- 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 code [404](#).

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.

8.1.1.2. SSE method

A service may support the "ServerSentEventUri" property within the Event Service resource. If a client performs a GET on the URI specified by the "ServerSentEventUri", an SSE connection will be opened for the client. See the [Server-Sent Events: EventService section](#) for details on this method.

8.1.2. EventType based eventing

There are three types of events generated in a Redfish Service - life cycle, alert, and metric report. This method of eventing has been deprecated in the Redfish Schema.

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 properties 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.

Metric report events happen when the TelemetryService has generated a new Metric Report or updated an existing Metric Report. These types of events shall be generated as specified by the MetricReportDefinition resources found subordinate to the TelemetryService. This can be defined to be done on a periodic basis, on demand, or when changes in the metric properties are detected. See the Redfish MetricReportDefinition Schema for full details.

8.1.3. Ways to register for events

Event subscriptions can be subscribed to by specifying a RegistryPrefixes, ResourceTypes, OriginResources (including SubordinateResources) property in order to filter events to any EventDestination. An EventFormatType can also be specified.

The RegistryPrefixes property has the list of message registries that the service provides and that the subscriber would like messages corresponding to. The values of this property are the values of the RegistryPrefix and can be standard or OEM message registries. It acts like a filter, only sending messages to the subscriber if the RegistryPrefix in the subscription matches the RegistryPrefix of the registry. This value does not include the version of the registry. If this value is empty when subscribing, the subscriber can receive messages from any registry.

The ResourceTypes property has the list of Resource Types that the service provides events on which the subscriber can use in the ResourceType property of the EventDestination. The values of this property is an array of Resource Types and can be standard or OEM schema Resource Types. It acts like a filter, only sending messages to the subscriber if the ResourceType in the subscription matches the Resource Type of the OriginOfCondition. This value does not include the version of the schema (thus there are no periods). For example, if the normal Resource Type is "Task.v1_2_0.Task", then the value in this property is just "Task". If this value is empty when subscribing, the subscriber can receive messages from any resource.

OriginResources can be specified to limit the events sent to the destination to the resource list (in URI format) specified. Leaving this property empty indicates that events from any resources are acceptable. The property SubordinateResources can be specified to indicate those resources as well as subordinate ones, regardless of depth.

EventFormatType can be specified in the subscription as well. The service advertises the list of formats

that can be sent using the `EventFormatTypes` property in the `EventService`. This value represents the format of the payload sent to the Event Destination. If the value is not specified, then the payload will correspond to the Event Schema.

8.1.4. Event formats

There are two formats of events:

- [Metric report message objects](#): This format shall be when the `TelemetryService` has generated a new Metric Report or updated an existing Metric Report.
- [Event message objects](#): This format shall be used for all other types of events.

8.1.4.1. 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.

Event messages may also have an `EventGroupId` property. The purpose of this property is to let clients know that different messages may be from the same event. For instance, if a LAN cable is disconnected, they may get a specific message from one registry about the LAN cable being disconnected, another message from a general registry about the resource changing, perhaps a message about resource state change and maybe even more. In order for the client to be able to tell all of these have the same root cause, these messages would have the same value for the `EventGroupId` property.

8.1.4.2. Metric report message objects

Metric report message objects sent to the specified client endpoint shall contain the properties as

described in the Redfish MetricReport Schema.

8.1.5. OEM Extensions

OEMs can extend both messages and message registries. There are OEM sections defined in any individual message (per the message registry schema definition). Thus if OEMs wish to provide additional information or properties, this can be done using the OEM section. OEMs shall not supply additional message arguments beyond those in a standard message registry. OEMs may substitute their own message registry for the standard registry in order to provide the OEM section within the registry but shall not change the standard values (such as Messages) in such registries.

8.2. Asynchronous operations

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

The Task service is used to describe the service that handles tasks. It contains a Resource 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 been 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 the Task Monitor and may include the Retry-After header to specify the amount of time the client should wait before querying status of the operation.

The Task Monitor is an opaque URL generated by the service intended to be used by the client that initiated the request. The client queries the status of the operation by performing a GET request on the Task Monitor.

The client should not include the mime type application/http in the Accept Header when performing a GET request to the Task 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 Task Monitor returned in the location header. If a service supports canceling a task, it shall have DELETE in the Allow header for the Task Monitor.

The client may cancel the operation by performing a DELETE on the Task Monitor URL. The service determines when to delete the associated Task resource object.

The client may also cancel the operation by performing a DELETE on the Task resource. Deleting the Task resource object may invalidate the associated Task Monitor and subsequent GET on the Task Monitor URL returns either [410](#) (Gone) or [404](#) (Not Found).

In the unlikely event that a [202](#) (Accepted) is returned on the DELETE of the Task Monitor or Task resource, an additional Task shall not be started and instead the existing Task resource may be monitored for status of the cancellation request. When the Task has finally completed cancellation, operations to the Task Monitor and Task resource shall return a [404](#) (Not Found).

After the operation has been completed, the service shall update the TaskState with the appropriate value. The values indicating that a task has been completed are defined in the Task schema.

After the operation has been completed, the Task Monitor shall return the appropriate status code (such as, but not limited to, OK [200](#) for most operations, Created [201](#) for POST to create a resource) and include the headers and response body of the initial operation, as if it had been 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) or [404](#) (Not Found) if the operation has been completed and the service has already deleted the task. This can occur if the client waits too long to read the Task Monitor.

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 Task Monitor associated with the activity. The response may also include the Retry-After 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 GET, PUT, PATCH, POST, and DELETE methods.

8.3. 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.

8.4. Discovery

Automatic discovery of managed devices supporting the Redfish Scalable Platform Management API may be 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.

8.4.1. UPnP compatibility

For compatibility with general purpose SSDP client software, primarily UPnP, UDP 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.

8.4.2. USN format

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

8.4.3. M-SEARCH response

The Redfish Service Search Target (ST) is defined as: urn:dmmtf-org:service:redfish-rest:1

The managed device shall respond to M-SEARCH queries searching for Search Target (ST) of the Redfish Service as well as "ssdp:all". For UPnP compatibility, the managed device should respond to M-SEARCH queries searching for Search Target (ST) of "upnp:rootdevice".

The URN provided in the ST header 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 backward-compatible with previous minor revisions, then that minor version shall be appended and preceded 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".

The managed device shall provide clients with the AL header pointing to the Redfish Service Root URL.

For UPnP compatibility, the managed device should provide clients with the LOCATION header pointing to the UPnP XML descriptor.

An example response to an M-SEARCH multicast or unicast query shall follow the format shown below. A service may provide additional headers for UPnP compatibility. Fields in brackets are placeholders for device-specific values.

```
HTTP/1.1 200 OK
CACHE-CONTROL:max-age=<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:
```

8.4.4. 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.

8.5. Server-Sent Events

Server-Sent Events (SSE), as defined by the Web Hypertext Application Technology Working Group, allows for a client to open a connection with a web service, and the web service can continuously push data to the client as needed. Resource responses for SSE shall have a Content-Type header set as "text/event-stream;charset=UTF-8". A service may occasionally send a comment within a stream to keep the connection alive. The following clauses describe how this is used by Redfish in different contexts of the Redfish data model. Details about SSE can be found in the [HTML5 Specification](#).

8.5.1. EventService

A service's implementation of the "EventService" resource may contain a property called "ServerSentEventUri". If a client performs a GET on the URI specified by the "ServerSentEventUri", the service shall keep the connection open and conform to the [HTML5 Specification](#) until the client closes the socket. Events generated by the service shall be sent to the client using the open connection.

When a client opens an SSE stream for the EventService, the service shall create an EventDestination instance in the Subscriptions collection for the EventService to represent the connection. The "Context" property in the EventDestination resource shall be an opaque string generated by the service. The service shall delete the corresponding EventDestination instance when the connection is closed. The service shall close the connection if the corresponding EventDestination is deleted.

There are two formats of SSE streams:

- [Metric report SSE stream](#): This format shall be used when the TelemetryService has generated a new Metric Report or updated an existing Metric Report.
- [Event message SSE stream](#): This format shall be used for all other types of events.

The service should support using the \$filter query parameter provided in the URI for the SSE stream by the client in order to reduce the amount of data returned to the client. The following table shows the properties that the service should allow to be filtered. The \$filter syntax shall follow the format specified in the [Query parameters for Filter clause](#).

Property	Description	Example
EventType	The service shall only send Events of the matching EventType. See the EventType enum defined by the Redfish Event Schema for the values allowed.	<code>http://sseuri?\$filter=EventType eq StatusChange</code>
MetricReportDefinition	The service shall only send MetricReports generated from the specified MetricReportDefinition.	<code>http://sseuri?\$filter=MetricReportDefinition eq '/redfish/v1/TelemetryService/MetricReportDefinitions/PowerMetrics'</code>
RegistryPrefix	The service shall only send Events with Messages that are	<code>http://sseuri?\$filter=(RegistryPrefix eq Resource) or (RegistryPrefix eq Task)</code>

Property	Description	Example
	part of the specified Registry Prefix.	
ResourceType	The service shall only send Events for resources matching the type.	<code>http://sseuri?\$filter=(ResourceType eq 'Power') or (ResourceType eq 'Thermal')</code>
EventFormatType	The service shall only send event payloads of the matching type. See the EventFormatType enum defined by the Redfish EventService Schema for the values allowed.	<code>http://sseuri?\$filter=EventFormatType eq Event</code>
MessageId	The service shall only send Events containing the matching Message Id.	<code>http://sseuri?\$filter=MessageId eq 'Contoso.1.0.TempAssert'</code>
OriginResource	The service shall only send Events for the specified resource.	<code>http://sseuri?\$filter=OriginResource eq '/redfish/v1/Chassis/1/Thermal'</code>

8.5.1.1. Event message SSE stream

The service shall use the "id" field in the SSE stream to uniquely indicate an event. The value of the "id" field shall be the same as the "Id" property in the event payload. The value of the "Id" property should be a positive integer value and should be generated in a sequential manner. A service should accept the "Last-Event-ID" header from the client in order to allow a client to restart the event stream in case the connection is interrupted.

The service shall use the "data" field in the SSE stream to include the JSON representation of the Event object as defined in the [Event message objects section](#).

The example payload below shows a stream containing a single event with the "id" field set to 1, and the "data" field containing a single Event object.


```

id: 1
data: {
  data: "@odata.context": "/redfish/v1/$metadata#Event.Event",
  data: "@odata.type": "#Event.v1_1_0.Event",
  data: "Id": "1",
  data: "Name": "Event Array",
  data: "Context": "ABCDEFGH",
  data: "Events": [
    data: {
      data: "MemberId": "1",
      data: "EventType": "Alert",
      data: "EventId": "ABC132489713478812346",
      data: "Severity": "Warning",
      data: "EventTimestamp": "2017-11-23T17:17:42-0600",
      data: "Message": "The LAN has been disconnected",
      data: "MessageId": "Alert.1.0.LanDisconnect",
      data: "MessageArgs": [
        data: "EthernetInterface 1",
        data: "/redfish/v1/Systems/1"
      ],
      data: "OriginOfCondition": {
        data: "@odata.id": "/redfish/v1/Systems/1/EthernetInterfaces/1"
      },
      data: "Context": "ABCDEFGH"
    }
  ]
}
data: {}

```

8.5.1.2. Metric report SSE stream

The service shall use the "id" field in the SSE stream to uniquely indicate a metric report transmission. The value of the "id" field shall be the same as the "ReportSequence" property in the metric report payload. The value of the "ReportSequence" property should be a positive integer value and should be generated in a sequential manner. A service should accept the "Last-Event-ID" header from the client in order to allow a client to restart the metric report stream in case the connection is interrupted.

The service shall use the "data" field in the SSE stream to include the JSON representation of the Metric Report object as defined in the [Metric report message objects section](#).

The example payload below shows a stream containing a metric report with the "id" field set to 127, and the "data" field containing the Metric Report object.

```

id: 127
data: {

```

```

data:    "@odata.id": "/redfish/v1/TelemetryService/MetricReports/
AvgPlatformPowerUsage",
data:    "@odata.context": "/redfish/v1/$metadata#MetricReport.MetricReport",
data:    "@odata.type": "#MetricReport.v1_0_0.MetricReport",
data:    "Id": "AvgPlatformPowerUsage",
data:    "Name": "Average Platform Power Usage metric report",
data:    "ReportSequence": "127",
data:    "MetricReportDefinition": {
data:        "@odata.id": "/redfish/v1/TelemetryService/MetricReportDefinitions/
AvgPlatformPowerUsage"
data:    },
data:    "MetricValues": [
data:        {
data:            "MetricId": "AverageConsumedWatts",
data:            "MetricValue": "100",
data:            "Timestamp": "2016-11-08T12:25:00-05:00",
data:            "MetricProperty": "/redfish/v1/Chassis/Tray_1/Power#/0/
PowerConsumedWatts"
data:        },
data:        {
data:            "MetricId": "AverageConsumedWatts",
data:            "MetricValue": "94",
data:            "Timestamp": "2016-11-08T13:25:00-05:00",
data:            "MetricProperty": "/redfish/v1/Chassis/Tray_1/Power#/0/
PowerConsumedWatts"
data:        },
data:        {
data:            "MetricId": "AverageConsumedWatts",
data:            "MetricValue": "100",
data:            "Timestamp": "2016-11-08T14:25:00-05:00",
data:            "MetricProperty": "/redfish/v1/Chassis/Tray_1/Power#/0/
PowerConsumedWatts"
data:        }
data:    ]
data:}

```

9. Security

9.1. Protocols

9.1.1. TLS

Implementations shall support TLS v1.1 or later.

Implementations should support the latest version of the TLS v1.x specification.

Implementations should support the [SNIA TLS Specification for Storage Systems](#).

9.1.2. Cipher suites

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

Redfish implementations should consider supporting ciphers similar to below that 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."

Redfish implementations should support the following additional ciphers.

```
TLS_RSA_WITH_AES_128_CBC_SHA
```

References to RFCs:

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

9.1.3. Certificates

Redfish implementations shall support replacement of the default certificate if one is provided.

Redfish implementations shall use certificates that are compliant with X.509 v3 certificate format, as defined in [RFC5280](#).

9.2. 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.

9.2.1. HTTP header security

- All write requests to Redfish objects 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 information when authentication failures occur.
- Redfish objects shall not be available unauthenticated, except for
 - The root object that is needed to identify the device and service locations
 - The \$metadata object that is needed to retrieve resource types
 - The OData Service Document that is needed for compatibility with OData clients
 - The version object located at /redfish
- External services linked via external references are not part of this specification, and may have other security requirements.

9.2.1.1. HTTP redirect

- When there is a HTTP Redirect, the privilege requirements for the target resource shall be enforced.
- Generally if the location is reachable without authentication, but only over https, the server should issue a redirect to the https version of the resource. For cases where the resource is only accessible with authentication, a [404](#) should be returned.

9.2.2. Extended error handling

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

9.2.3. 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.

9.2.3.1. BASIC authentication

HTTP BASIC authentication as defined by [RFC7235](#) shall be supported, and shall only use compliant TLS connections to transport the data between any third-party authentication service and clients.

9.2.3.2. Request/Message level authentication

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

9.2.4. Session management

9.2.4.1. Session life cycle management

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

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

9.2.4.2. 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 Property](#) under the Sessions property. Both URIs shall be the same.

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

9.2.4.3. Session login

A Redfish session is created, without requiring an authentication header, by an HTTP POST to the SessionService's Sessions Resource Collection, including the following POST body:

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

- An X-Auth-Token header that contains a "session auth token" that the client can use in subsequent requests
- A Location header that contains a hyperlink to the newly created session resource
- The JSON response body that contains a full representation of the newly created session object (example below)

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

{
  "@odata.context": "/redfish/v1/$metadata#Session.Session",
  "@odata.id": "/redfish/v1/SessionService/Sessions/1",
  "@odata.type": "#Session.v1_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 hyperlink 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 hyperlink 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 hyperlink. 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.

9.2.4.4. 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. A request via POST to create a new session using the HTTP port should redirect to the HTTPS port if both HTTP and HTTPS are enabled.

9.2.4.5. Session lifetime

Note that Redfish sessions "time-out" as opposed to having a token expiration time like some token-based methods use. For Redfish sessions, as long as 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, the session is automatically terminated.

9.2.4.6. 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 hyperlink 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.

When a session is terminated, the service shall not affect independent connections established originally by this session for other purposes, such as connections for [Server-Sent Events](#) or transferring an image for the Update Service.

9.2.5. 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 that do not include an ETag.
- User Management activity is atomic.
- Extended error messages shall NOT provide privileged information when authentication failures occur.

9.2.6. 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.

9.2.7. Event subscriptions

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

9.2.8. 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 predefined roles. The predefined roles shall be created as follows (where Role Name is the value of the Id property for the role resource):
 - Role Name = "Administrator"
 - AssignedPrivileges = Login, ConfigureManager, ConfigureUsers, ConfigureComponents, ConfigureSelf
 - Role Name = "Operator"
 - AssignedPrivileges = Login, ConfigureComponents, ConfigureSelf
 - Role Name = "ReadOnly"
 - AssignedPrivileges = Login, ConfigureSelf
 - Implementations shall support all of the predefined roles.
 - The predefined Roles may include OEM privileges.
 - The privilege array defined for the predefined roles shall not be modifiable.
 - A service may optionally define additional "Custom" roles.
 - A service may allow users to create custom roles by issuing a POST to the "Roles" Resource Collection.
 - A predefined role or a custom role shall be assigned to a user when a user is created.
 - The client shall provided the "RoleId" property when creating a new Manager Account in order to select the predefined role or a custom role.
- 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.

9.2.9. Redfish Service Operation-to-Privilege mapping

For every request made by a Redfish client to a Redfish service, the Redfish service shall determine that the authenticated identity of the requester has the authorization to perform the requested operation on the resource specified in the request. Using the role and privileges authorization model, where an authenticated identity context is assigned a role and a role is a set of privileges, the service will typically check a HTTP request against a mapping of the authenticated requesting identity role/privileges and determine whether the identity privileges are sufficient to perform the operation specified in the request.

9.2.9.1. Why specify Operation-to-Privilege mapping

Initial versions of the Redfish specifications specified several Role-to-Privilege mappings for standardized Roles and normatively identified several Privilege labels but did not normatively define what these privileges meant in detail or how privilege-to-operations mappings could be specified or represented in a normative fashion. The lack of a methodology to define what privilege(s) are required to perform a specific requested operation against the URI specified in the request puts at risk the interoperability between Redfish service implementations that Redfish clients may encounter due to variances in privilege requirements between implementations. Also, a lack of methodology for specifying and representing the operation-to-privilege mapping prevents the Redfish Forum or other governing organizations to normatively define privilege requirements for a service.

9.2.9.2. Representing Operation-to-Privilege mappings

A Redfish service should provide a Privilege Registry file in the service Registry Collection. The Privilege Registry file represents the Privilege(s) required to perform an operation against a URI specified in a HTTP request to the service. The Privilege Registry is a single JSON document that contains a Mappings array of PrivilegeMapping entity elements where there is an individual element for every schema entity supported by the service. The operation-to-privilege mapping is defined for every entity schema and applies to every resource the service implements for the applicable schema. There are several situations where specific resources or elements of resources may have differing operation-to-privilege mappings

than the entity mappings and the entity level mappings have to be overridden. The methodology for specifying entity level operation-to-privilege mappings and related overrides are defined in the PrivilegeRegistry schema.

If a Redfish service provides a Privilege Registry document, the service shall use the Redfish Forum's Redfish Privilege Mapping Registry definition as a base operation-to-privilege mapping definition for operations that the service supports in order to promote interoperability for Redfish clients.

9.2.9.3. OperationMap syntax

An operation map defines the set of privileges required to perform a specific operation on an entity, entity element, or resource. The operations mapped are GET, PUT, PATCH, POST, DELETE and HEAD. Privilege mapping are defined for each operation irrespective of whether the service or the API data model support the specific operation on the entity, entity element or resource. Privilege labels used may be the Redfish standardized labels defined in the Privilege.PrivilegeType enumeration and they may be OEM-defined privilege labels. The privileges required for an operation can be specified with logical AND and OR behavior as required (see the [Privilege AND and OR syntax](#) clause for more information). The following example defines the privileges required for various operations on Manager entity. Unless mapping overrides to the OperationMap array are defined (syntax explained in next clause), the specified operation-to-privilege mapping would represent behavior for all Manager resources in a service implementation.

```
{
  "Entity": "Manager",
  "OperationMap": {
    "GET": [
      {
        "Privilege": [ "Login" ]
      }
    ],
    "HEAD": [
      {
        "Privilege": [ "Login" ]
      }
    ],
    "PATCH": [
      {
        "Privilege": [ "ConfigureManager" ]
      }
    ],
    "POST": [
      {
        "Privilege": [ "ConfigureManager" ]
      }
    ],
  },
}
```

```

    "PUT": [
      {
        "Privilege": [ "ConfigureManager" ]
      }
    ],
    "DELETE": [
      {
        "Privilege": [ "ConfigureManager" ]
      }
    ]
  }
}

```

9.2.9.4. Mapping overrides syntax

Several situations occur where operation-to-privilege mapping varies from what might be specified at an entity schema level. These situations are:

- **Property Override** - Where a property has different privilege requirements that the resource (document) it is in. For example, the Password property in the ManagerAccount resource requires the "ConfigureSelf" or the "ConfigureUsers" privilege to change in contrast to the "ConfigureUsers" privilege required for the rest of the properties in ManagerAccount resources.
- **Subordinate Override** - Where an entity is used in context of another entity and the contextual privileges need to govern. For example, the privileges for PATCH operations on EthernetInterface resources depends on whether the resource is subordinate to Manager (ConfigureManager is required) or ComputerSystem (ConfigureComponents is required) resources.
- **Resource URI Override** - Where a specific resource instance has different privilege requirements for operation that those defined for the entity schema. The overrides are defined in the context of the operation-to-privilege mapping for an entity.

9.2.9.5. Property override example

In the following example, the Password property on the ManagerAccount resource requires the "ConfigureSelf" or the "ConfigureUsers" privilege to change in contrast to the "ConfigureUsers" privilege required for the rest of the properties on ManagerAccount resources.

```

{
  "Entity": "ManagerAccount",
  "OperationMap": {
    "GET": [
      {
        "Privilege": [ "ConfigureManager" ]
      }
    ]
  }
}

```

```

        },
        {
            "Privilege": [ "ConfigureUsers" ]
        },
        {
            "Privilege": [ "ConfigureSelf" ]
        }
    ],
    "HEAD": [
        {
            "Privilege": [ "Login" ]
        }
    ],
    "PATCH": [
        {
            "Privilege": [ "ConfigureUsers" ]
        }
    ],
    "POST": [
        {
            "Privilege": [ "ConfigureUsers" ]
        }
    ],
    "PUT": [
        {
            "Privilege": [ "ConfigureUsers" ]
        }
    ],
    "DELETE": [
        {
            "Privilege": [ "ConfigureUsers" ]
        }
    ]
},
"PropertyOverrides": [
    {
        "Targets": [ "Password" ],
        "OperationMap": {
            "GET": [
                {
                    "Privilege": [ "ConfigureManager" ]
                }
            ],
            "PATCH": [
                {
                    "Privilege": [ "ConfigureManager" ]
                }
            ]
        }
    }
]

```

```

        {
            "Privilege": [ "ConfigureSelf" ]
        }
    ]
}
]
}

```

9.2.9.6. Subordinate override

The Targets property within SubordinateOverrides lists a hierarchical representation for when to apply the override. In the following example, the override for an EthernetInterface entity is applied when it is subordinate to an EthernetInterfaceCollection entity, which in turn is subordinate to a Manager entity. If a client were to PATCH an EthernetInterface entity that matches this override condition, it would require the "ConfigureManager" privilege; otherwise, the client would require the "ConfigureComponents" privilege.

```

{
    "Entity": "EthernetInterface",
    "OperationMap": {
        "GET": [
            {
                "Privilege": [ "Login" ]
            }
        ],
        "HEAD": [
            {
                "Privilege": [ "Login" ]
            }
        ],
        "PATCH": [
            {
                "Privilege": [ "ConfigureComponents" ]
            }
        ],
        "POST": [
            {
                "Privilege": [ "ConfigureComponents" ]
            }
        ],
        "PUT": [
            {
                "Privilege": [ "ConfigureComponents" ]
            }
        ],
    },
}

```

```

        "DELETE": [
            {
                "Privilege": [ "ConfigureComponents" ]
            }
        ],
    },
    "SubordinateOverrides": [
        {
            "Targets": [
                "Manager",
                "EthernetInterfaceCollection"
            ],
            "OperationMap": {
                "GET": [
                    {
                        "Privilege": [ "Login" ]
                    }
                ],
                "PATCH": [
                    {
                        "Privilege": [ "ConfigureManager" ]
                    }
                ]
            }
        }
    ]
}

```

9.2.9.7. ResourceURI Override

In the following example, use of the ResourceURI Override syntax for representing operation privilege variations for specific resource URIs is demonstrated. The example specifies both ConfigureComponents and OEMAdminPriv privileges are required in order to perform a PATCH operation on the two resource URIs listed as Targets.

```

{
    "Entity": "ComputerSystem",
    "OperationMap": {
        "GET": [
            {
                "Privilege": [ "Login" ]
            }
        ],
        "HEAD": [
            {

```

```

        "Privilege": [ "Login" ]
    },
],
"PATCH": [
    {
        "Privilege": [ "ConfigureComponents" ]
    }
],
"POST": [
    {
        "Privilege": [ "ConfigureComponents" ]
    }
],
"PUT": [
    {
        "Privilege": [ "ConfigureComponents" ]
    }
],
"DELETE": [
    {
        "Privilege": [ "ConfigureComponents" ]
    }
]
},
"ResourceURIOverrides": [
    {
        "Targets": [
            "/redfish/v1/Systems/VM6",
            "/redfish/v1/Systems/Sys1"
        ],
        "OperationMap": {
            "GET": [
                {
                    "Privilege": [ "Login" ]
                }
            ],
            "PATCH": [
                {
                    "Privilege": [ "ConfigureComponents", "OEMSysAdminPriv" ]
                }
            ]
        }
    }
]
}

```

9.2.9.8. Privilege AND and OR syntax

Logical combinations of privileges required to perform an operation on an entity, entity element or resource are defined by the array placement of the privilege labels in the OperationMap GET, HEAD, PATCH, POST, PUT, DELETE operation element arrays. For OR logical combinations, the privilege label is placed in the operation element array as individual elements. In the following example, either Login or OEMPrivilege1 privileges are required to perform a GET operation.

```
{
  "GET": [
    {
      "Privilege": [ "Login" ]
    },
    {
      "Privilege": [ "OEMPrivilege1" ]
    }
  ]
}
```

For logical AND combinations, the privilege label is placed in the Privilege property array within the operation element. In the following example, both ConfigureComponents and OEMSysAdminPriv are required to perform a PATCH operation.

```
{
  "PATCH": [
    {
      "Privilege": [ "ConfigureComponents", "OEMSysAdminPriv" ]
    }
  ]
}
```

10. Redfish Host Interface

The Redfish Host Interface Specification defines how software executing on a host computer system can interface with a Redfish service that manages the host. See [DSP0270](#) for details.

11. Redfish Composability

A service may implement the CompositionService resource off of ServiceRoot to support the binding of resources together. One example is disaggregated hardware, which allows for independent components,

such as processors, memory, I/O controllers, and drives, to be bound together to create logical constructs that operate together. This allows for a client to dynamically assign resources for a given application.

11.1. Composition requests

A service that implements the CompositionService (as defined by the CompositionService schema) shall support one or more of the following types of composition requests:

- [Specific Composition](#)

A service that supports removing a composed resource shall support the DELETE method on the composed resource.

11.1.1. Specific Composition

A Specific Composition is when a client has identified an exact set of resources in which to build a logical entity. A service that supports Specific Composition requests shall implement the ResourceBlock resource (ResourceBlock schema) and the ResourceZone resource (Zone schema) for the CompositionService. ResourceBlocks provide an inventory of components available to the client for building compositions. ResourceZones describe the binding restrictions of the ResourceBlocks managed by the service.

The ResourceZone resource within the CompositionService shall include the CollectionCapabilities annotation in the response. The CollectionCapabilities annotation allows a client to discover which collections in the service support compositions, and how the POST request for the collection is formatted, as well as what properties are required. A service that supports Specific Compositions shall support a POST request that contains an array of hyperlinks to ResourceBlocks. The specific nesting of the ResourceBlock array is defined by the schema for the resource being composed.

A service that supports updating a composed resource shall also support the PUT and/or PATCH methods on the composed resource with a modified list of ResourceBlocks.

Example Specific Composition of a ComputerSystem:

```
POST /redfish/v1/Systems HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
  "Name": "Sample Composed System",
  "Links": {
    "ResourceBlocks": [
```

```
    { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/  
ComputeBlock0" },  
    { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/DriveBlock2"  
} ,  
    { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/NetBlock4" }  
  ]  
}  
}
```

12. ANNEX A (informative)

12.1. Change log

Version	Date	Description
1.6.1	2018-12-13	Added clause about percent encoding being allowed for query parameters.
		Changed Expand example to use SoftwareInventory instead of LogEntry.
		Added missing clause about the usage of a separator for multiple query parameters.
		Fixed '\$filter' examples to use '/' instead of '.' for property paths.
		Clarified the usage of Messages in a successful Action response; provided an example.
		Added clarification about services supporting a subset of HTTP operations on resources specified in schema.
		Added clarification about services implementing writable properties as read only.
		Added clarification about session termination not affecting connections opened by the session.
		Added "Redfish Provider" term definition.
		Updated JSON Schema references to point to Draft 7 of the JSON Schema specification.

Version	Date	Description
		Added clarifications about scenarios for when a request to add an Event Subscription contains conflicting information and how services respond.
		Removed language about ignoring the 'Links' property in PATCH requests.
		Clarified usage of ETags to show that a client is not supposed to PATCH '@odata.etag' when attempting to use ETag protection for a resource.
		Clarified usage of the 'only' query parameter to show it's not to be combined with '\$expand' and not to be used with singular resources.
		Clarified the usage of HTTP status codes with Task Monitors.
		Various spelling and grammar fixes.
1.6.0	2018-08-23	Added methods of using \$filter on the SSE URI for the EventService.
		Added support for the OpenAPI Specification v3.0. This allows OpenAPI-conforming software to access Redfish service implementations.
		Added strict definitions for the URI patterns used for Redfish resources to support OpenAPI. Each URI is now constructed using a combination of fixed, defined path segments and the values of "Id" properties for Resource Collections. Also added restrictions on usage of unsafe characters in URIs. Implementations reporting support for Redfish v1.6.0 must conform to these URI patterns.
		Added support for creating and naming Redfish schema files in the OpenAPI YAML-based format.
		Added URI construction rules for OEM extensions.
		Changed ETag usage to require strong ETag format.
		Added requirement for HTTP Allow header as a response header for GET and HEAD operations.
		Added Metric Reports as a type of event that can be produced by a Redfish Service. Added support for SSE streaming of Metric reports in support of new TelemetryService schema.
		Added Registry, Resource, Origin, or EventFormatType-based event subscription methods as detailed in the Specification and schema.

Version	Date	Description
		Added an EventFormatType to allow for additional payload types for subscription-based or streaming events. Deprecated 'EventType'-based event subscription mechanism.
		Added Event message grouping capability.
		Provided guidance for defining and using OEM extensions for Messages and Message Registries.
		Added 'excerpt' and 'only' query parameters.
		Clarified requirements for Resource Collection responses, which includes required properties that were expected, but not listed explicitly in the Specification.
		Made inclusion of the '@odata.context' annotation optional.
		Removed requirement for clients to include the 'OData-Version' HTTP header in all requests.
1.5.1	2018-08-10	Added clarifications to required properties in structured properties derived from ReferenceableMembers.
		Reorganized Eventing section to break out the different subscription methods to differentiate pub-sub from SSE.
		Removed statements referencing OData conformance levels.
		Clarified terminology to explain usage of absolute versus relative URIs throughout.
		Clarified client-side HTTP Accept header requirements.
		Added evaluation order for supported query parameters and clarified examples.
		Clarified handling of annotations in response payloads when used with \$select queries.
		Clarified service handling of annotations in PATCH requests.
		Clarified handling of various PATCH request error conditions.
		Clarified ability to create Resource Collection members by POST operations to the Resource Collection or the Members[] array within the resource.

Version	Date	Description
		Corrected several examples to show required properties in payload.
		Clarified usage of the Link header and values of 'rel=describedBy'.
		Clarified that the HTTP status code table only describes Redfish-specific behavior and that unless specified, all other usage follows the definitions within the appropriate RFCs.
		Added missing entry for HTTP status code 431.
		Added statement that HTTP status code 503 can be used during reboot/reset of a Service to indicate that the service is temporarily unavailable.
		Clarified usage of the @odata.type annotation within embedded objects.
		Added missing statements about required properties 'Name', 'Id', 'MemberId', and common property 'Description', which have always been shown as required in schema files, but were not mentioned in the Specification.
		Added guidance for the value of time/date properties when time is unknown.
		Added missing description of the 'title' property in Action requests.
		Clarified usage of the '@odata.nextLink' annotation at the end of Resource Collections.
		Added additional guidance for naming properties and enumeration values that contain 'OEM' or that include acronyms.
		Corrected requirements for Description and LongDescription schema annotations.
		Corrected name of 'ConfigureComponents' in Operation-to-Privilege mapping clause.
		Various typographical errors and grammatical improvements.
1.5.0	2018-04-05	Added support for Server-Sent Eventing for streaming events to web-based GUIs or other clients.
		Added "OperationApplyTime" annotation to provide a mechanism for specifying deterministic behavior for the application of Create, Delete or

Version	Date	Description
		Action (POST) operations.
1.4.1	2018-04-05	Updated name of the DMTF Forum from 'SPMF' to 'Redfish Forum'.
		Changed terminology for consistent usage of 'hyperlink'.
		Added example to clarify usage of \$select query parameter with \$expand, and clarified expected results when using 'AutoExpand'. Corrected order of precedence for \$filter parameter options.
		Corrected terminology for OEM-defined actions removing 'custom' in favor of OEM, and clarified that the Action 'target' property is always required for an Action, along with its usage.
		Corrected location header values for responses to Data modification requests that create a Task (Task resource vs. Task Monitor). Clarified error handling of DELETE operations on Task resources.
		Removed references to obsolete and unused 'Privilege' annotation namespace.
		Clarified usage of the 'Base.1.0.GeneralError' message in the Base Message Registry.
		Added missing durable URIs for Registries and Profiles, clarified intended usage for each folder in the Repository. Added missing file naming conventions for Registries and Profiles, and clarified file naming for Schemas.
		Added statement to clarify that additional headers may be added to M-SEARCH responses for SSDP to allow for UPnP compatibility.
		Clarified assignment requirements for predefined or custom roles when new Manager Account instances are created, using the 'RoleId' property.
1.4.0	2017-11-17	Added support for optional Query parameters ("\$expand", "\$filter", and "\$select") on requests to allow for more efficient retrieval of resources or properties from a Redfish Service.
		Clarified HTTP status and payload responses after successful processing of data modification requests. This includes POST operations for performing Actions, as well as other POST, PATCH, or PUT requests.

Version	Date	Description
		Added HTTP status code entries for 428 and 507 to clarify the proper response to certain error conditions. Added reference links to the HTTP status code table throughout.
		Updated Abstract to reflect current state of the Specification.
		Added reference to RFC 6585 and clarified expected behavior when ETag support is used in conjunction with PUT or PATCH operations.
		Added definition for "Property" term and updated text to use term consistently.
		Added "Client Requirement" column and information for HTTP headers on requests.
		Clarified the usage and expected format of the Context property value.
		Added clause detailing how Structured properties can be revised and how to resolve their definitions in schema.
		Added more descriptive definition for the Settings resource. Added an example for the "SettingsObject". Added description and example for using the "SettingsApplyTime" annotation.
		Added Action example using the ActionInfo resource in addition to the simple AllowableValues example. Updated example to show a proper subset of the available enumerations to reflect a real-world example.
		Added statement explaining the updates required to TaskState upon task completion.
1.3.0	2017-08-11	Added support for a Service to optionally reject a PATCH or PUT operation if the If-Match or If-Match-None HTTP header is required by returning the HTTP status code 428 .
		Added support for a Service to describe when the values in the Settings object for a resource are applied via the "@Redfish.SettingsApplyTime" annotation.
1.2.1	2017-08-10	Clarified wording of the "Oem" object definition.
		Clarified wording of the "Partial resource results" section.
		Clarified behavior of a Service when receiving a PATCH with an empty JSON object.

Version	Date	Description
		Added statement about other uses of the HTTP 503 status code.
		Clarified format of URI fragments to conform to RFC6901.
		Clarified use of absolute and relative URIs.
		Clarified definition of the "target" property as originating from OData.
		Clarified distinction between "hyperlinks" and the "Links Property".
		Corrected the JSON example of the privilege map.
		Clarified format of the "@odata.context" property.
		Added clauses about the schema file naming conventions.
		Clarified behavior of a Service when receiving a PUT with missing properties.
		Clarified valid values in the "Accept" header to include wildcards per RFC7231.
		Corrected "ConfigureUser" privilege to be spelled "ConfigureUsers".
		Corrected Session Login section to include normative language.
1.2.0	2017-04-14	Added support for the Redfish Composability Service.
		Clarified Service handling of the Accept-Encoding header in a request.
		Improved consistency and formatting of example requests and responses throughout.
		Corrected usage of the "@odata.type" property in response examples.
		Clarified usage of the "Required" schema annotation.
		Clarified usage of SubordinateOverrides in the Privilege Registry.
1.1.0	2016-12-09	Added Redfish Service Operation to Privilege Mapping clause. This functionality allows a Service to present a resource or even property-level mapping of HTTP operations to account Roles and Privileges.
		Added references to the Redfish Host Interface Specification (DSP0270).
1.0.5	2016-12-09	Errata release. Various typographical errors.

Version	Date	Description
		Corrected terminology usage of "Collection", "Resource Collection" and "Members" throughout.
		Added glossary entries for "Resource Collection" and "Members".
		Corrected Certificate requirements to reference definitions and requirements in RFC 5280 and added a normative reference to RFC 5280.
		Clarified usage of HTTP POST and PATCH operations.
		Clarified usage of HTTP Status codes and Error responses.
1.0.4	2016-08-28	Errata release. Various typographical errors.
		Added example of an HTTP Link Header and clarified usage and content.
		Added Schema Modification clause describing allowed usage of the Schema files.
		Added recommendation to use TLS 1.2 or later, and to follow the SNIA TLS Specification. Added reference to the SNIA TLS Specification. Added additional recommended TLS_RSA_WITH_AES_128_CBC_SHA Cipher suite.
		Clarified that the "Id" property of a Role resource must match the Role Name.
1.0.3	2016-06-17	Errata release. Corrected missing Table of Contents and Clause numbering. Corrected URL references to external specifications. Added missing Normative References. Corrected typographical error in ETag example.
		Clarified examples for ExtendedInfo to show arrays of Messages.
		Clarified that a POST to Session Service to create a new Session does not require authorization headers.
1.0.2	2016-03-31	Errata release. Various typographical errors.
		Corrected normative language for M-SEARCH queries and responses.
		Corrected Cache-Control and USN format in M-SEARCH responses.
		Corrected schema namespace rules to conform to OData namespace

Version	Date	Description
		requirements (.n.n.n becomes .vn_n_n) and updated examples throughout the document to conform to this format. File naming rules for JSON Schema and CSDL (XML) schemas were also corrected to match this format and to allow for future major (v2) versions to coexist.
		Added missing clause detailing the location of the Schema Repository and listing the durable URLs for the repository.
		Added definition for the value of the Units annotation, using the definitions from the UCUM specification. Updated examples throughout to use this standardized form.
		Modified the naming requirements for Oem Property Naming to avoid future use of colon ':' and period '.' in property names, which can produce invalid or problematic variable names when used in some programming languages or environments. Both separators have been replaced with underscore '_', with colon and period usage now deprecated (but valid).
		Removed duplicative or out-of-scope sub-clauses from the Security clause, which made unintended requirements on Redfish service implementations.
		Added missing requirement that property names in Resource Responses must match the casing (capitalization) as specified in schema.
		Updated normative references to current HTTP RFCs and added clause references throughout the document where applicable.
		Clarified ETag header requirements.
		Clarified that no authentication is required for accessing the Service Root resource.
		Clarified description of Retrieving Collections.
		Clarified usage of 'charset=utf-8' in the HTTP Accept and Content-Type headers.
		Clarified usage of the 'Allow' HTTP Response Header and added missing table entry for usage of the 'Retry-After' header.
		Clarified normative usage of the Type Property and Context Property,

Version	Date	Description
		explaining the ability to use two URL forms, and corrected the "@odata.context" URL examples throughout.
		Corrected inconsistent terminology throughout the Collection Resource Response clause.
		Corrected name of normative Resource Members Property ('Members', not 'value').
		Clarified that Error Responses may include information about multiple error conditions.
		Corrected name of Measures.Unit annotation term as used in examples.
		Corrected outdated reference to Core OData specification in Annotation Term examples.
		Added missing 'Members' property to the Common Redfish Resource Properties clause.
		Clarified terminology and usage of the Task Monitor and related operations in the Asynchronous Operations clause.
		Clarified that implementation of the SSDP protocol is optional.
		Corrected typographical error in the SSDP USN field's string definition (now '::dmtf-org').
		Added missing OPTIONS method to the allowed HTTP Methods list.
		Fixed nullability in example.
1.0.1	2015-09-17	Errata release. Various grammatical corrections.
		Clarified normative use of LongDescription in schema files.
		Clarified usage of the 'rel-describedby' link header.
		Corrected text in example of 'Select List' in OData Context property.
		Clarified Accept-Encoding Request header handling.
		Deleted duplicative and conflicting statement on returning extended error resources.
		Clarified relative URI resolution rules.

Version	Date	Description
		Clarified USN format.
1.0.0	2015-08-04	Initial release.