Document Identifier: DSP0266

Date: 2019-08-08

Version: 1.8.0

Redfish Specification

Supersedes: 1.7.1

Document Class: Normative

Document Status: Published

Document Language: en-US

CONTENTS

Foreword

The Redfish Forum of the DMTF develops the Redfish standard.

DMTF is a not-for-profit association of industry members that promotes enterprise and systems management and interoperability. For information about the DMTF, see https://www.dmtf.org/.

Acknowledgments

The DMTF acknowledges the following individuals for their contributions to the Redfish standard, including this document and Redfish Schemas, interoperability profiles, and Message Registries:

- Rafiq Ahamed - Hewlett Packard Enterprise
- Richelle Ahlvers - Broadcom Inc.
- Jeff Autor - Hewlett Packard Enterprise
- Jeff Bobzin - Insyde Software Corp.
- David Black - Dell Inc.
- Patrick Boyd - Dell Inc.
- David Brockhaus - Vertiv
- Richard Brunner - VMware Inc.
- Sean Byland - Cray Inc.
- Lee Calcote - Seagate Technology
- P Chandrasekhar - Dell Inc.
- Barbara Craig - Hewlett Packard Enterprise
- 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
- Kevin Ferguson - Vertiv
- 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
- Cactus Jiang - Vertiv
- Barry Kittner - Intel Corporation
- Steven Krig - Intel Corporation
- Jennifer Lee - Intel Corporation
- John Leung - Intel Corporation
- Steve Lyle - Hewlett Packard Enterprise
- Jagan Molleti - Dell Inc.
- Milena Natanov - Microsoft Corporation

- Scott Phuong - Cisco Systems, Inc.
- Michael Pizzo - Microsoft Corporation
- Chris Poblete - Dell Inc.
- Michael Raineri - Dell Inc.
- Irina Salvan - Microsoft Corporation
- Bill Scherer - Hewlett Packard Enterprise
- Hemal Shah - Broadcom Inc.
- Jim Shelton - Vertiv
- Tom Slaight - Intel Corporation
- Donnie Sturgeon - Vertiv
- Pawel Szymanski - Intel Corporation
- Paul Vancil - Dell Inc.
- Joseph White - Dell Inc.
- Linda Wu - Super Micro Computer, Inc.

1. Abstract

Redfish is a standard that uses RESTful interface semantics to access a schema based data model to conduct management operations. It is suitable for a wide range of devices, from stand-alone servers, to composable infrastructures, and to large-scale cloud environments.

The initial Redfish scope targeted servers.

The DMTF and its alliance partners expanded that scope to cover most data center IT equipment and other solutions, and both in- and out-of-band access methods.

Additionally, the DMTF and other organizations that use Redfish as part of their industry standard or solution have added educational material.

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.

- Architectural Styles and the Design of Network-based Software Architectures, R. Fielding, 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- DMTF DSP0270 Redfish Host Interface Specification, https://www.dmtf.org/sites/default/files/standards/documents/DSP0270_1.0.pdf
- HTML Living Standard: Server-sent events https://html.spec.whatwg.org/multipage/server-sent-events.html
- ISO 639-1:2002 ISO 639-1:2002 Codes for the representation of names of languages -- Part 1: Alpha-2 code
- IETF RFC 1738 T. Berners-Lee et al, Uniform Resource Locators (URL), https://www.ietf.org/rfc/rfc1738.txt
- IETF RFC 3986 T. Berners-Lee et al, Uniform Resource Identifier (URI): Generic Syntax, https://www.ietf.org/rfc/rfc3986.txt
- IETF RFC 4627, D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), https://www.ietf.org/rfc/rfc4627.txt
- IETF RFC 5280, D. Cooper et al, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, https://www.ietf.org/rfc/rfc5280.txt
- IETF RFC 5789, L. Dusseault et al, PATCH Method for HTTP, https://www.ietf.org/rfc/rfc5789.txt
- IETF RFC 5988, M. Nottingham, Web Linking, https://www.ietf.org/rfc/rfc5988.txt
- IETF RFC 6585, M. Nottingham, et al, Additional HTTP Status Codes, https://www.ietf.org/rfc/rfc6585.txt

- IETF RFC 6901, P. Bryan, Ed. et al, JavaScript Object Notation (JSON) Pointer, https://www.ietf.org/rfc/rfc6901.txt
- IETF RFC 6906, E. Wilde, The 'profile' Link Relation Type, https://www.ietf.org/rfc/rfc6906.txt
- IETF RFC 7230, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, https://www.ietf.org/rfc/rfc7230.txt
- IETF RFC 7231, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, https://www.ietf.org/rfc/rfc7231.txt
- IETF RFC 7232, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, https://www.ietf.org/rfc/rfc7232.txt
- IETF RFC 7234, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Caching, https://www.ietf.org/rfc/rfc7234.txt
- IETF RFC 7235, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Authentication, https://www.ietf.org/rfc/rfc7235.txt
- IETF RFC 7578, L. Masinter et al., Returning Values from Forms: multipart/form-data, https://www.ietf.org/rfc/rfc7578.txt
- ISO/IEC Directives, ISO/IEC Directives, Part 2 (English), https://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtypeH
- JSON Schema: A Media Type for Describing JSON Documents 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-handrews-json-schema-validation-01, https://tools.ietf.org/html/draft-handrews-json-schema-validation-01
- OData Version 4.0 Plus Errata 03: Core Vocabulary. 10 March 2016. https://docs.oasis-open.org/odata/odata/v4.0/errata03/csd01/complete/vocabularies/Org.OData.Core.V1.xml
- OData JSON Format Version 4.0. 24 February 2014. https://docs.oasis-open.org/odata/odata-json-format/v4.0/os/odata-json-format-v4.0-os.html
- OData Version 4.0 Part 1: Protocol. 24 February 2014. https://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. 24 February 2014. https://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). 24 February 2014. https://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html
- OData Version 4.0 Plus Errata 03: Units of Measure Vocabulary. 10 March 2016. https://docs.oasis-open.org/odata/odata/v4.0/errata03/csd01/complete/vocabularies/Org.OData.Measures.V1.xml
- The OpenAPI Specification https://github.com/OAI/OpenAPI-Specification
- Redfish Schema: RedfishExtensions. https://redfish.dmtf.org/schemas/v1/RedfishExtensions_v1.xml
- Simple Service Discovery Protocol/1.0 Operating without an Arbiter. 28 October 1999. https://tools.ietf.org/html/draft-cai-ssdp-v1-03
- SNIA TLS Specification for Storage Systems. 20 November 2014. https://www.snia.org/tech_activities/standards/curr_standards/tls
- The Unified Code for Units of Measure. https://www.unitsofmeasure.org/ucum.html
- W3C Cross-Origin Resource Sharing. 16 January 2014. https://www.w3.org/TR/cors/

3. Terms and definitions

Some terms and phrases in this document have specific meanings beyond their typical English meanings. This clause defines those terms and phrases.

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, Clause 7. 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, Clause 7 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 6.

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.

This document defines these additional terms:

Term	Definition
Baseboard management controller (BMC)	Embedded device or service. Typically an independent microprocessor or system-on-chip with associated firmware in a computer system that completes out-of-band systems monitoring and management-related tasks.
Collection	See Resource Collection.
CRUD	Basic **C**reate, **R**ead, **U**pdate, and **D**elete operations that any interface can support.
Event	Data structure that corresponds to one or more alerts.
Excerpt	A subset of data from one Resource copied and presented in another Resource. This concept is used to provide data in convenient locations without duplication of entire Resources.
Hypermedia API	API that you navigate through URIs that a service returns.
Managed system	System that provides information, status, or control via a Redfish-defined interface.
Member	Single resource instance in a Resource Collection.

Term	Definition
Message	Complete HTTP- or HTTPS-formatted request or response. In the REST-based Redfish protocol, every request should result in a response.
OData	Open Data Protocol, as defined in OData-Protocol.
OData service document	Resource that provides information about the service root for generic OData clients.
Operation	The HTTP request methods that map generic CRUD operations. These are POST, GET, PUT/PATCH, HEAD and DELETE.
Property	Name-and-value pair in a Redfish-defined request or response. A property can be any valid JSON data type.
Redfish client	Communicates with a Redfish Service and accesses one or more of the service's resources or functions.
Redfish event receiver	Software that runs at the event destination that receives events from a Redfish Service.
Redfish protocol	Discovers, connects to, and inter-communicates with a Redfish Service.
Redfish Schema	Defines Redfish Resources according to OData schema representation. You can directly translate a Redfish Schema to a JSON Schema representation.
Redfish Service	Implementation of the protocols, resources, and functions that deliver the interface that this specification defines and its associated behaviors for one or more managed systems. Also known as the *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 a designed 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.

Term	Definition
Request	Message from a client to a service.
Resource	Addressable by a URI and represents a Redfish data structure.
Resource Collection	Resource that contains a set of like resources where the number of instances can shrink or grow.
Resource tree	Tree structure of resources accessible through 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.
Response	Message from a service to a client in response to a request message.
Service root	Resource that 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	Registration of a destination to receive events.

4. Acronyms

This document uses these acronyms:

Acronym	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

Acronym	Definition
KVM-IP	Keyboard, Video, Mouse redirection over IP
NIC	Network Interface Card
PCI	Peripheral Component Interconnect
PCIe	PCI Express
TCP	Transmission Control Protocol
XSS	Cross-Site Scripting

5. Overview

Redfish is a management standard that uses a data model representation with a RESTful interface.

Being RESTful, Redfish is easier to use and implement.

Being model-oriented, it can express the relationships between components and the semantics of the services and components within them. The model is also easy to extend.

By requiring JSON representation, Redfish enables easy integration with programming environments. It is also easy to interpret by humans.

The model is defined by an interoperable Redfish Schema. It is published in OpenAPI YAML, OData CSDL, and JSON Schema, and is freely available.

5.1. Scope

This specification defines the required protocols, data model, behaviors, and other architectural components for an interoperable, multivendor, remote, and out-of-band capable interface. This interface meets the cloud-based and web-based IT professionals' expectations for scalable platform management. While large and hyperscale environments are the primary focus, clients can use the specification for individual system management.

The specifications defines elements that are mandatory for all Redfish implementations, as well as optional elements that can be chosen by system vendors or manufacturers. This specification also defines points at which extensions specific to the OEM (system vendor) can be provided by a given implementation.

The specification sets normative requirements for Redfish Services and associated materials, such as Redfish Schema files. In general, the specification does not set requirements for Redfish clients, but

indicates what a Redfish client should do to successfully and effectively access and use a Redfish Service.

The specifications do not require that implementation of the Redfish interfaces and functions require particular hardware or firmware.

5.2. Goals

As an architecture, data representation, and definition of protocols that enable a client to access Redfish Services, Redfish has these goals:

Goal	Purpose
Scalable	Stand-alone machines and racks of equipment.
Flexible	Can be implemented with existing hardware, or entirely as a software service.
Extensible	New and vendor-specific capabilities can be easily added to the data model.
Backward-compatible	Additional capabilities can be added while preserving investments in implementations of earlier versions of the specification.
Interoperable	Consistent functionality across multiple vendor implementations.
Standards-based	Built on ubiquitous and secure protocols and leveraging other standards where applicable.
Simple	Usable without the need for highly specialized programming skills or systems knowledge.
Lightweight	Designed to reduce complexity and implementation cost, as well as minimizing required footprint for implementations.

5.3. Design tenets

To deliver these goals, Redfish adheres to these design tenets:

- Provide a RESTful interface using a JSON payload and a data model.
- Separate protocol from data model, allowing them to be revised and used independently.
- Specify versioning rules for protocols and schema.
- Leverage strength of ubiquitous standards where it meets architectural requirements, such as JSON, HTTP, OData, OpenAPI, and the RFCs referenced by this document.
- Organize the data model to present value-add features, clearly demarcated, while in the same payload as standardized items.
- Make data in payloads as obvious in context as possible.

- Maintain implementation flexibility. Do not tie the interface to any particular underlying implementation or architecture.
- Focus on most widely used capabilities. Avoid adding complexity to address functions that are only valued by a small percentage of users.

5.4. Limitations

Redfish minimizes the need for clients to complete upgrades by using strict versioning and forward-compatibility rules, and separation of the protocols from the data model. However, Redfish does not guarantee that clients never need to update their software. For example, clients might need to upgrade for managing new types of systems or components, as well as updates to the data model.

Interoperable does not mean identical. Many elements of Redfish are optional. Clients should be prepared to discover the optional elements using the built in discovery methods.

The resource tree reflects the topology of the system and its devices. Consequently, different hardware or device types result in different resource trees, even for identical systems from the same manufacturer. References between resources may result in a graph instead of a tree. Clients that traverse the resource tree should provide logic to avoid infinite loops.

Additionally, not all Redfish Resources use simple REST read-and-write semantics. Different use cases may follow other types of client logic. For example, clients cannot simply read user credentials or certificates from one service and write them to another service.

Finally, the hyperlink values between resources and other elements can vary across implementations. Clients should not assume that they can reuse hyperlinks across different Redfish Service instances.

5.5. Additional design background and rationale

5.5.1. REST-based

Redfish exposes many service applications as RESTful interfaces. This document defines a RESTful interface.

Redfish defines a RESTful interface because it:

- Enables a lightweight implementation, using fewer layers than previous standards.
- Is a prevalent access method in the industry.
- Is easy to learn, document, and implement in modern programming languages.
- Has a number of development environments and a healthy tooling ecosystem.
- Fits with the design goal of simplicity.
- Equally applies to software application space as it does to embedded environments, which enables convergence and sharing of code within the management ecosystem.

- Adapts well to any data modeling language.
- Has industry-provided security and discovery mechanisms.

5.5.2. Data oriented

The Redfish data model is developed by focusing on the contents of the payload. By concentrating on the contents of the payload first, Redfish payloads are easily mapped to schema definition languages and encoding types. The data model is defined in various schema languages, including OpenAPI YAML, OData CSDL, and JSON Schema.

5.5.3. Separation of protocol from data model

Redfish separates the protocol operations from the data model and versions the protocol independently from the data model. This enables clients to extend and change the data model as needed without requiring the protocol version to change.

5.5.4. Hypermedia API service root

Redfish has a single service root URI and clients can discover all other resources through referenced URIs.

5.5.5. OpenAPI v3.0 support

The OpenAPI Specification v3.0 provides a rich ecosystem of tools for using RESTful interfaces that meet the design requirements of that specification. Starting with v1.6.0 of the Redfish Specification, the Redfish Schemas support the OpenAPI YAML file format, and URI patterns that conform to the OpenAPI Specification were defined. Conforming Redfish Services that support the Redfish protocol version v1.6.0 or later implement those URI patterns to enable use of the OpenAPI ecosystem.

5.5.6. 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 few common conventions between them.

To provide for interoperability between APIs, OData defines a set of common RESTful conventions and annotations. Redfish adopts OData conventions for describing schema, URL conventions, and definitions for typical properties in a JSON payload.

5.6. Service elements

5.6.1. Synchronous and asynchronous operation support

Some operations can take 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.

To determine whether an operation was completed synchronously or asynchronously, clients can review the HTTP status codes. For more information, see the Asynchronous operations clause.

5.6.2. Eventing mechanism

Redfish provides the ability to send messages outside the normal request and response paradigm to clients. The service uses these messages, or *events*, to asynchronously notify the client of a state change or error condition, usually of a time critical nature.

Two styles of eventing are currently defined by this specification - push style eventing, and Server-Sent Events (SSE).

In push style eventing, when the service 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.

In SSE style eventing, the client opens an SSE connection to the service by performing a GET on the URI specified by the `ServerSentEventUri` in the Event Service.

For information, see the Eventing clause.

5.6.3. Actions

Actions are Redfish operations that do not easily map to RESTful interface semantics. These types of operations may not directly affect properties in the Redfish Resources. The Redfish Schema defines certain standard actions for common Redfish Resources. For these standard actions, the Redfish Schema contains the normative language on the behavior of the action.

5.6.4. Service discovery

While the service itself is at a well-known URI, clients need to discover the network address of the service. Like UPnP, Redfish uses SSDP for discovery. A wide variety of devices, such as printers and client operating systems, support SSDP. It is simple, lightweight, IPv6 capable, and suitable for implementation in embedded environments.

For more information, see the Discovery clause.

5.6.5. Remote access support

Remote management functionality typically includes access mechanisms for redirecting operator interfaces such as serial console, keyboard video and mouse (KVM-IP), command shell (i.e., command line interface), and virtual media. While these mechanisms are critical functionality, they cannot be reasonably implemented as a RESTful interface. Therefore, this standard does not define the protocols or access mechanisms for those services, but encourages implementations that leverage existing standards. However, the Redfish schema includes resources and properties allowing client discovery of these capabilities and describing access mechanisms to enable interoperability.

5.7. Security

The challenge of remote interface security is to protect both the interface and exchanged data. To accomplish this, Redfish provides authentication and encryption. As part of this security, Redfish defines and requires minimum levels of encryption.

For more information, see the Security details clause.

6. Protocol details

In this document, the Redfish protocol refers to the RESTful mapping to HTTP, TCP/IP and other protocol, transport, and messaging layer aspects. 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.

The Redfish protocol is designed around a web service based interface model. This provides network and interaction efficiency for both user interface (UI) and automation usage. Specifically, the ability to leverage existing tool chains.

The Redfish protocol uses:

- HTTP methods for common CRUD operations and to retrieve header information.
- Actions, which are limited in use, to expand operations beyond CRUD-type operations.
- Media types to negotiate the type of data sent in the message body.
- HTTP status codes to indicate the success or failure of the server's request.
- Error responses to return more information than HTTP error codes.
- TLS for sending secure messages. See Security.
- Asynchronous semantics for long running operations.

A Redfish interface shall be exposed through a web service endpoint implemented using HTTP, version 1.1 (RFC7230, RFC7231, RFC7232).

The subsequent clauses describe how the Redfish interface uses and adds constraints to HTTP to ensure interoperability of Redfish implementations.

6.1. Universal Resource Identifiers

A Universal Resource Identifier (URI) identifies a resource, including the service root and all Redfish Resources.

- A URI shall identify each unique instance of a resource.
- URIs shall not include any RFC1738-defined unsafe characters.
  - For example, the `{`, `}`, `|`, `^`, `~`, `[`, `]`, `` ` ``, and `\` characters are unsafe because gateways and other transport agents can sometimes modify these characters.
  - Do not use the `#` character for anything other than the start of a fragment.
- URIs shall not include any percent-encoding of characters. This restriction does not apply to the query parameters portion of the URI.

Performing a GET operation on a URI yields a representation of the resource containing properties and hyperlinks to associated resources. The service root 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 practice, known as hypermedia, allows for the discovery of resources by following hyperlinks.

Redfish considers the RFC3986-defined scheme, authority, root service and version, and unique resource path component parts of the URI.

For example, the `https://mgmt.vendor.com/redfish/v1/Systems/1` URI contains these component parts:

Component part	In the example
The scheme.	`https:`
The authority, which defines the authority to which to delegate the URI.	`//mgmt.vendor.com`
The root service and version.	`/redfish/v1/`
The resource path, which provides a unique identifier for the resource.	`Systems/1`

In a URI:

- The scheme and authority component parts are not part of the unique resource path because redirection capabilities and local operations may cause the connection portion to vary.
- The service and resource path component parts *uniquely identify* the resource in a Redfish Service.

In an implementation:

- The resource path component part shall be unique.
- A relative reference in the body and HTTP headers payload can identify a resource in that same implementation.
- An absolute URI in the body and HTTP headers payload can identify a resource in a different implementation.

For the absolute URI definition, see RFC3986.

For example, a POST operation may return the `/redfish/v1/Systems/2` URI in the `Location` header of the response, which points to the POST-created resource.

Assuming that the client connects through the `mgmt.vendor.com` appliance, the client accesses the resource through the `https://mgmt.vendor.com/redfish/v1/Systems/2` absolute URI.

RFC3986-compliant URIs may also contain the query, `?query`, and frag, `#frag`, components. For information about queries, see Query parameters. When a URI includes a fragment (`frag`) to submit an operation, the server ignores the fragment.

If a property in a response is a reference to another property within a resource, use the RFC6901-defined URI Fragment Identifier Representation format. If the property is as a reference property in 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 in the `/redfish/v1/Chassis/MultiBladeEncl/Thermal` resource:

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

For requirements on constructing Redfish URIs, see the Resource URI Patterns annotation clause.

6.2. HTTP methods

The following table describes the mapping of HTTP methods to the operations that are supported by Redfish. The "required" column specifies whether the method is supported by a Redfish interface.

- If the value is "yes", the HTTP method shall be supported.
- If the value is "no", the value may be supported.

For HTTP methods not supported by the Redfish Service or not listed in the table, an HTTP 405 status code shall be returned by the Redfish Service.

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 Cross-Origin Resource Sharing (CORs) preflight	No

6.3. HTTP redirect

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

- All Redfish clients shall correctly handle HTTP redirect.

NOTE: Refer to the Security details clause for security implications of HTTP Redirect.

6.4. Media types

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

In HTTP messages, the media type is specified in the `Content-Type` header. To tell a service to send the response through certain media types, the client sets the HTTP `Accept` header to a list of the media types.

- All resources shall be available through the JSON `application/json` media type.
- Redfish Services shall make every resource available in a JSON-based representation, as specified in RFC4627. Receivers shall not reject a JSON-encoded message, and shall offer at least one JSON-based response representation. An implementation may offer additional non-JSON media type representations.

To request compression, clients specify an `Accept-Encoding` request header.

When requested by the client, services should support gzip compression.

6.5. ETags

To reduce unnecessary RESTful accesses to resources, the Redfish Service should support the association of a separate entity tag (ETag) with each resource.

- Implementations should support the return of ETag properties for each resource.
- Implementations should support the return of ETag headers for each single-resource response.
- Implementations shall support the return of ETag headers for GET requests of `ManagerAccount` resources.

Because the service knows whether the new version of the object is substantially different, the service generates and provides the ETag as part of the resource payload.

The ETag mechanism supports both ***strong*** and ***weak*** validation. If a resource supports an ETag, it shall use the RFC7232-defined ETag strong validator.

This specification does not mandate a particular algorithm for ETag creation, but ETags should be highly collision-free.

An ETag can be:

- A hash
- A generation ID
- A time stamp
- Some other value that changes when the underlying object changes

If a client calls PUT or PATCH to update a resource, it should include an ETag from a previous GET in the HTTP `If-Match` or `If-None-Match` header. If a service supports the return of the ETag header on a resource, the service may respond with HTTP 428 status code if the `If-Match` or `If-None-Match` header is missing from the PUT or PATCH request for the same resource, as specified in RFC6585.

In addition to the return of the ETag property on each resource, a Redfish Service should return the ETag header on:

- A client PUT, POST, or PATCH operation
- A GET operation for an individual resource

The format of the ETag header is:

```
ETag: "<string>"
```

6.6. Protocol version

The protocol version is separate from the resources' version or the Redfish Schema version that the resources support.

Each Redfish protocol version is strongly typed by using the URI of the Redfish Service in combination with the resource obtained at that URI, called the `ServiceRoot` resource.

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

The URI defines the major version of the protocol.

The `RedfishVersion` property of the `ServiceRoot` resource defines the protocol version, which includes the major version, minor version, and errata version of the protocol, as defined in the Redfish Schema for that resource.

The protocol version is a string in the format:

```
MajorVersion.MinorVersion.ErrataVersion
```

where

Variable	Type	Version	Description
MajorVersion	Integer	Major	A backward-compatible class change.
MinorVersion	Integer	Minor	A minor update. Redfish introduces new functionality but does not remove any functionality. The minor version preserves compatibility with earlier minor versions.
ErrataVersion	Integer	Errata	A fix in the earlier version.

Any resource that a client discovers through hyperlinks that the root service or any root service-referenced service or resource returns shall conform to the same protocol version that the root service supports.

A GET operation on the `/redfish` resource shall return this response body:

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

6.7. Redfish-defined URIs and relative reference rules

A Redfish Service shall support these Redfish-defined URIs:

URI	Returns
`/redfish`	The version. A major update that does not preserve compatibility with earlier minor versions.
`/redfish/v1/`	The Redfish service root.
`/redfish/v1/odata`	The Redfish OData service document.
`/redfish/v1/$metadata`	The Redfish metadata document.

A Redfish Service should support these Redfish-defined URIs:

URI	Returns
`/redfish/v1/openapi.yaml`	The Redfish OpenAPI YAML document.

In addition, the service shall process the following URI without a trailing slash in one of these ways:

- Redirect it to the associated Redfish-defined URI.
- Treat it as the equivalent URI to the associated Redfish-defined URI:

URI	Associated Redfish-defined URI
`/redfish/v1`	`/redfish/v1/`

All other Redfish Service-supported URIs shall match the Resource URI patterns definitions, except the supplemental resources that the `@Redfish.Settings`, `@Redfish.ActionInfo`, and `@Redfish.CollectionCapabilities` payload annotations reference. The client shall treat the URIs for these supplemental resources as opaque.

All Redfish Service-supported URIs are reserved for future standardization by DMTF and DMTF alliance partners, except OEM extension URIs, which shall conform to the OEM resource URI requirements.

All relative references (see RFC3986) that the service uses shall start with either:

- A double forward slash (`//`) and include the authority (network-path), such as `//mgmt.vendor.com/redfish/v1/Systems`.

- A single forward slash (`/`) and include the absolute-path, such as `/redfish/v1/Systems`.

7. Service requests

This clause describes the requests that clients can send to Redfish Services.

7.1. Request headers

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

For Redfish Services:

- Redfish Services shall 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", or if the value is set to "Conditional" under the conditions noted in the Description column.
- Redfish Services should 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".

For Redfish Clients (sending the HTTP requests):

- 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", or if the value in the Client Requirement column is set to "Conditional" under the conditions noted in the Description column.
- 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	RFC7231	Communicates to the server the media type or types that this client is prepared to accept. Services shall support Resource requests with `Accept` header values of `application/json` or `application/json;charset=utf-8`

Header	Service requirement	Client requirement	Supported values	Description
				Services shall support metadata requests with `Accept` header values of `application/xml` or `application/xml;charset=utf-8` Services shall support any request with `Accept` header values of `application/*`, `application/*;charset=utf-8`, `*/*`, or `*/*;charset=utf-8`
`Accept-Encoding`	No	No	RFC7231	Indicates whether the client can handle gzip-encoded responses. If a service cannot send an acceptable response to a request with this header, it shall respond with the HTTP 406 status code. If the request omits this header, the service shall not return gzip-encoded responses.
`Accept-Language`	No	No	RFC7231	The languages that the client accepts in the response. If the request omits this header, uses the service's default language for the response.
`Authorization`	Conditional	Conditional	RFC7235, Section 4.2	Required for Basic authentication. A client can access unsecured resources without this header on

Header	Service requirement	Client requirement	Supported values	Description
				systems that support basic authentication.
`Content-Length`	No	No	RFC7231	The size of the message body. To indicate the size of the body, a client can use the `Transfer-Encoding: chunked` header. If a service must use `Content-Length` and does not support `Transfer-Encoding`, it responds with the HTTP 406 status code.
`Content-Type`	Conditional	Conditional	RFC7231	The request format. Required for operations with a request body. Services shall accept the `Content-Type` header set to either `application/json` or `application/json;charset=utf-8`. It is recommended that clients use these values in requests because other values can cause an error.
`Host`	Yes	No	RFC7230	Enables support of multiple origin hosts at a single IP address.
`If-Match`	Conditional	No	RFC7232	To ensure that clients update the resource from a known state, PUT and PATCH requests for resources

Header	Service requirement	Client requirement	Supported values	Description
				for which a service returns ETags shall support `If-Match`. While not required for clients, it is highly recommended for PUT and PATCH operations.
`If-None-Match`	No	No	RFC7232	A service only returns the resource if the current ETag of that resource does not match the ETag sent in this header. If the ETag in this header matches the resource's current ETag, the GET operation returns the HTTP 304 status code.
`Last-Event-ID`	No	No	HTML5 SSE	The event source's last `id` field from the SSE stream. Requests history event data. See Server-Sent Events.
`Max-Forwards`	No	No	RFC7231	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
`OData-MaxVersion`	No	No	4.0	The maximum OData version that an OData-aware client understands.
`OData-Version`	Yes	No	4.0	The OData version.

Header	Service requirement	Client requirement	Supported values	Description
				Services shall reject requests that specify an unsupported OData version. If a service encounters an unsupported OData version, it should reject the request with the HTTP 412 status code.
`Origin`	Yes	No	W3C CORS, Section 5.7	Enables web applications to consume a Redfish Service while preventing CSRF attacks.
`User-Agent`	Yes	No	RFC7231	Traces product tokens and their versions. The header can list multiple product tokens.
`Via`	No	No	RFC7230	Defines the network hierarchy and recognizes message loops. Each pass inserts its own `Via` header.

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 ***Yes***.

Header	Service requirement	Client requirement	Supported values	Description
`X-Auth-Token`	Yes	Conditional	Opaque encoded octet strings	Authenticates user sessions. The token value shall be indistinguishable from random. While services shall support this

Header	Service requirement	Client requirement	Supported values	Description
				header, a client can access unsecured resources without establishing a session.

7.2. GET (read requests)

The GET operation is used to retrieve resources from a Redfish Service. Clients make a GET request to the individual resource URI. Clients may obtain the resource URI from published sources, such as the OpenAPI document, or from a Resource identifier property in a previously retrieved resource response, such as the Links Property.

The service shall return the resource representation using one of the media types listed in the `Accept` header, subject to the media types' requirements. If the `Accept` header is absent, the service shall return the resource's representation as `application/json`. Services may, but are not required to, support the convention of retrieving individual properties within a resource by appending a segment containing the property name to the URI of the resource.

- The HTTP GET operation shall 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.

7.2.1. Resource Collection requests

Clients retrieve a Resource Collection by making a GET request to the Resource Collection URI. The response includes the Resource Collection's properties and an array of its `Members`.

No requirements are placed on implementations to return a consistent set of members when a series of requests that use paging query parameters are made over time to obtain the entire set of members. It is possible that these calls can result in missed or duplicate elements if multiple GETs are used to retrieve the `Members` array instances through 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 the next link property or paging, the count property shall return the total number of resources that the `Members` array references.

A subset of the members can be retrieved using client paging query parameters.

A service may not be able to return all of the contents of a Resource Collection request in a single

response body. In this case, the response can be paged by the service. If a service pages a response to a Resource Collection request, the following rules shall apply:

- Responses may contain a subset of the members of the full Resource Collection.
- Individual members shall not be split across response bodies.
- A next link annotation in the response body that has the URI to the next set of members in the collection shall be supplied.
- The next link property shall adhere to the rules in the Next link property clause.
- GET Operations on the next link shall return the subsequent section of the Resource Collection response.

7.2.2. Service root request

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

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

Services shall not require authentication to retrieve the service root and `/redfish` resources.

7.2.3. OData service and $metadata document requests

Redfish Services expose two OData-defined documents at specific URIs to enable generic OData clients to navigate the Redfish Service.

- Service shall expose an OData $metadata Document at the `/redfish/v1/$metadata` URI.
- Service shall expose an OData Service Document at the `/redfish/v1/odata` URI.
- Service shall not require authentication to retrieve the OData $metadata Document or the OData Service Document.

7.3. Query parameters

To paginate, retrieve subsets of resources, or expand the results in a single response, clients can include the query parameters. Some query parameters apply only to Resource Collections.

Services:

- Shall only support query parameters on GET operations.
- Should support the `$top`, `$skip`, `only`, and `excerpt` query parameters.
- May support the `$expand`, `$filter`, and `$select` query parameters.
- Shall include the `ProtocolFeaturesSupported` object in the service root if the service supports query parameters.
    - This is to indicate which parameters and options have been implemented.

- Shall ignore unknown or unsupported query parameters that do not begin with `$`.
- Shall use the `&` operator to separate multiple query parameters in a single request

Services shall return:

- The HTTP 501 Not Implemented status code for any unsupported query parameters that start with `$`.
- An extended error that indicates the unsupported query parameters for this resource.
- The HTTP 400 Bad Request status code for any query parameters that contain values that are invalid, or values applied to query parameters without defined values (e.g., `excerpt` or `only`).

Services should return:

- The HTTP 400 Bad Request status code with the `QueryNotSupportedOnResource` message from the Base Message Registry for any implemented query parameters that are not supported on a Resource in the request.
- The HTTP 400 Bad Request status code with the `QueryNotSupportedOnResource` message from the Base Message Registry for any supported query parameters that apply only to Resource Collections, but are used on singular Resources. This includes query parameters such as `$filter`, `$top`, `$skip`, and `only`.
- The HTTP 400 Bad Request status code with the `QueryNotSupportedOnOperation` message from the Base Message Registry for any supported query parameters that are used on operations other than GET.

The response body shall reflect the evaluation of the query parameters in this order:

- Prior to service side pagination: `$filter`, `$skip`, `$top`
- After applying any service side pagination: `$expand`, `$select`

Query parameter	Description and example
`excerpt`	Returns a subset of the resource's properties that match the defined `Excerpt` schema annotation. If no `Excerpt` schema annotation is defined for the resource, the entire resource is returned. **Example:** `http://resource?excerpt`

Query parameter	Description and example

`$expand=<string>`	Returns a hyperlink and its contents in-line with retrieved resources, as if a GET call response was included in-line with that hyperlink.
	
	See Use of the $expand query parameter.
	
	Example:
	
	`http://resource?$expand=*($levels=3)`
	
	`http://resourcecollection?$expand=.($levels=1)`

`$filter=<string>`	Applies to Resource Collections. Returns a subset of collection members that match the `$filter` expression. See Use of the $filter query parameter. **Example:** `http://resourcecollection?$filter=SystemType eq 'Physical'`
`only`	Applies to Resource Collections. If the target Resource Collection contains exactly one member, clients can use this query parameter to return that member's resource. If the collection contains either zero members or more than one member, the response returns the collection resource, as expected. Services should return the HTTP 400 Bad Request with the `QueryCombinationInvalid` message from the Base Message Registry if `only` is being combined with other query parameters. **Example:** `http://resourcecollection?only`
`$select=<string>`	Returns a subset of the resource's properties that match the `$select`

Query parameter	Description and example
	expression. See Use of the $select query parameter. **Example:** `http://resource?$select=SystemType,Status`
`$skip=<integer>`	Applies to Resource Collections. Returns a subset of the members in a Resource Collection. This paging query parameter defines the number of Members in the Resource Collection to skip. **Example:** `http://resourcecollection?$skip=5`
`$top=<integer>`	Applies to Resource Collections. Defines the number of members to show in the response. Minimum value is `1`. By default, returns all members. **Example:** `http://resourcecollection?$top=30`

7.3.1. Use of the $expand query parameter

The `$expand` query parameter allows a client to request a response that includes not only the requested resource, but also includes the contents of the subordinate or hyperlinked resources. The definition of this query parameter follows the OData-Protocol specification.

The `$expand` query parameter has a set of possible options that determine which hyperlinks in a resource are included in the expanded response. Some resources may already be expanded due to the resource's schema annotation `AutoExpand`, such as the `Temperature` object in the `Thermal` resource.

The Redfish-supported options for the `$expand` query parameter are listed in the following table. The service may implement some of these options but not others. Any other supported syntax for `$expand` is outside the scope of this specification.

Option	Description	Example
asterisk (`*`)	Shall expand all hyperlinks.	`http://resource?$expand=*`
`$levels`	The number of levels the service should cascade the `$expand` operation. The default level shall be 1. For example, `$levels=2` expands both: • The hyperlinks in the current resource (level 1). • The hyperlinks in the resulting expanded resources (level 2).	`http://resourcecollection?$expand=.($levels=2)`
period (`.`)	Shall expand all hyperlinks **not** in the Links Property section of the resource.	`http://resourcecollection?$expand=.`
tilde (`~`)	Shall expand all hyperlinks found in the Links Property section of the resource.	`http://resourcecollection?$expand=~`

Examples of `$expand` usage include:

- GET of a `SoftwareInventoryCollection`.

  With `$expand`, the client can request multiple `SoftwareInventory` collection member resources in one request rather than fetching them one at a time.

- GET of a `ComputerSystem`.

  With `$levels`, a single GET request can include the subordinate Resource Collections, such as `Processors` and `Memory`.

- GET all UUIDs in Members of the `ComputerSystem` collection.

  To accomplish this result, include both `$select` and `$expand` on the URI.

  The syntax is `GET /redfish/v1/Systems?$select=UUID&$expand=.($levels=1)`

When services execute `$expand`, they may omit some of the referenced resource's properties.

When clients use `$expand`, they should be aware that the payload may increase beyond what can be sent in a single response.

If a service cannot return the payload due to its size, it shall return HTTP 507 status code.

The following is an example showing the `RoleCollection` resource being expanded with the level set to 1:

```
{
    "@odata.id": "/redfish/v1/AccountService/Roles",
    "@odata.type": "#RoleCollection.RoleCollection",
    "Name": "Roles Collection",
    "Members@odata.count": 3,
    "Members": [
        {
            "@odata.id": "/redfish/v1/AccountService/Roles/Administrator",
            "@odata.type": "#Role.v1_1_0.Role",
            "Id": "Administrator",
            "Name": "User Role",
            "Description": "Admin User Role",
            "IsPredefined": true,
            "AssignedPrivileges": [
                "Login",
                "ConfigureManager",
                "ConfigureUsers",
                "ConfigureSelf",
                "ConfigureComponents"
```

```
            ]
        },
        {
            "@odata.id": "/redfish/v1/AccountService/Roles/Operator",
            "@odata.type": "#Role.v1_1_0.Role",
            "Id": "Operator",
            "Name": "User Role",
            "Description": "Operator User Role",
            "IsPredefined": true,
            "AssignedPrivileges": [
                "Login",
                "ConfigureSelf",
                "ConfigureComponents"
            ]
        },
        {
            "@odata.id": "/redfish/v1/AccountService/Roles/ReadOnly",
            "@odata.type": "#Role.v1_1_0.Role",
            "Id": "ReadOnly",
            "Name": "User Role",
            "Description": "ReadOnly User Role",
            "IsPredefined": true,
            "AssignedPrivileges": [
                "Login",
                "ConfigureSelf"
            ]
        }
    ]
}
```

7.3.2. Use of the $select query parameter

The `$select` query parameter indicates that the implementation should return a subset of the resource's properties that match the `$select` expression. If a request omits the `$select` query parameter, the response returns all properties by default. The definition of this query parameter follows the OData-Protocol specification.

The `$select` expression shall not affect the resource itself.

The `$select` expression defines a comma-separated list of properties to return in the response body.

The syntax for properties in object types shall be the object and property names concatenated with a slash (`/`).

An example of `$select` usage is:

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

When services execute `$select`, they shall return all requested properties of the referenced resource. The `@odata.id` and `@odata.type` properties shall be in the response payload and contain the same values as if `$select` was not performed. If the `@odata.context` property is supported, it shall be in the response payload and should be in the Context property recommended format. 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.

7.3.3. Use of the $filter query parameter

The `$filter parameter` allows a client to request a subset of the Resource Collection's members based on the `$filter` expression. The definition of this query parameter follows the OData-Protocol specification.

The `$filter` query parameter defines a set of properties and literals with an operator.

A literal value can be:

- A string enclosed in single quotes.
- A number.
- A boolean value.

If the literal value does not match the data type for the specified property, the service should reject `$filter` requests with the HTTP 400 status code.

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 are supported.

The following table lists the Redfish-supported values for the `$filter` query parameter. Any other supported syntax for `$filter` is outside the scope of this specification.

Value	Description	Example
`()`	Precedence grouping operator.	`(Status/State eq 'Enabled' and Status/Health eq 'OK') or SystemType eq 'Physical'`
`and`	Logical and operator.	`ProcessorSummary/Count eq 2 and MemorySummary/TotalSystemMemoryGiB gt 64`

Value	Description	Example
`eq`	Equal comparison operator.	`ProcessorSummary/Count eq 2`
`ge`	Greater than or equal to comparison operator.	`ProcessorSummary/Count ge 2`
`gt`	Great than comparison operator.	`ProcessorSummary/Count gt 2`
`le`	Less than or equal to comparison operator.	`MemorySummary/TotalSystemMemoryGiB le 64`
`lt`	Less than comparison operator.	`MemorySummary/TotalSystemMemoryGiB lt 64`
`ne`	Not equal comparison operator.	`SystemType ne 'Physical'`
`not`	Logical negation operator.	`not (ProcessorSummary/Count eq 2)`
`or`	Logical or operator.	`ProcessorSummary/Count eq 2 or ProcessorSummary/Count eq 4`

When evaluating expressions, services shall use the following operator precedence:

- Grouping
- Logical negation
- Relational comparison. `gt`, `ge`, `lt`, and `le` all have equal precedence.
- Equality comparison. `eq` and `ne` both have equal precedence.
- Logical `and`
- Logical `or`

If the service receives an unsupported `$filter` query parameter, it shall reject the request and return the HTTP 501 status code.

7.4. HEAD

The HEAD method differs from the GET method in that it shall not return message body information.

However, the HEAD method completes the same authorization checks and returns all the same meta information and status codes in the HTTP headers as a GET method.

Services may support the HEAD method to:

- Return meta information in the form of HTTP response headers.
- Verify hyperlink validity.

Services may support the HEAD method 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.

7.5. Data modification requests

To create, modify, and delete resources, clients issue the following operations:

- POST (create)
- PATCH (update)
- PUT (replace)
- DELETE (delete)
- POST (action) on the resource

The following clauses describe the success and error response requirements common to all data modification requests.

7.5.1. Modification success responses

For create operations, the response from the service, after the create request succeeds, should be one of these responses:

- HTTP 201 status code with a body that contains the JSON representation of the newly created resource after the request has been applied.
- HTTP 202 status code with a `Location` header set to the URI of a task monitor when the processing of the request requires additional time to be completed.
    - After processing of the task is complete, the created resource may be returned in response to a request to the task monitor URI with the HTTP 201 status code.
- HTTP 204 status code with empty payload in the event that the service cannot return a representation of the created resource.

For update, replace, and delete operations, the response from the service, after successful modification, should be one of the following responses:

- HTTP 200 status code with a body that contains the JSON representation of the targeted resource after the modification has been applied, or, for the delete operation, a representation of the deleted resource.

- HTTP 202 status code with a `Location` header set to the URI of a task monitor when the processing of the modification requires additional time.
    - After processing of the task is complete, the modified resource may be returned in response to a request to the task monitor URI with the HTTP 200 status code.
- HTTP 204 status code with an empty payload in the event that service cannot return a representation of the modified or deleted resource.

For details on success responses to action requests, see POST (action).

7.5.2. Modification error responses

If the resource exists but does not support the requested operation, services may return the HTTP 405 status code.

Otherwise, if the service returns a client 4xx or service 5xx status code, the service encountered an error and the resource shall not have been modified or created as a result of the operation.

7.6. PATCH (update)

To update a resource's properties, the service shall support the PATCH method.

The request body defines the changes to make to one or more properties in the resource that the request URI references. The PATCH request does not change any properties that are not in the request body. The service shall ignore OData annotations in the request body, such as Resource identifier, type, and ETag properties. Services may accept a PATCH with an empty JSON object, which indicates that the service should make no changes to the resource.

When modification succeeds, the response may contain a representation of the updated resource. See Modification success responses.

To gain the protection semantics of an ETag, the service shall use the `If-Match` or `If-None-Match` header and not the `@odata.etag` property value for that protection.

The implementation may reject the update on certain properties based on its own policies and, in this case, not perform the requested update. For the following exception cases, services shall return the following HTTP status codes and other information:

Exception case	The service returns
Modify several properties where one or more properties can never be updated. For example, when a property is read-only, unknown, or	• The HTTP 200 status code. • A resource representation with a message annotation that lists the non-updatable properties.

Exception case	The service returns
unsupported.	• The service may update other properties in the resource.
Modify a single property that can never be updated. For example, a property that is read-only, unknown, or unsupported.	• The HTTP 400 status code. • A resource representation with a message annotation that shows the non-updatable property.
Modify a resource or all properties that can never be updated.	• The HTTP 405 status code.
A client PATCH request against a Resource Collection.	• The HTTP 405 status code.
A client only provides OData annotations.	• The HTTP 400 status code with the `NoOperation` message from the Base Message Registry or one of the modification success responses.

In the absence of outside changes to the resource, the PATCH operation should be idempotent, although the original `ETag` value may no longer match.

7.6.1. PATCH on array properties

There are three possible styles of array properties in a Resource, which are detailed in the Array properties clause.

Within a PATCH request, the service shall accept `null` to remove an element, and accept an empty object `{}` to leave an element unchanged. Array properties using the fixed or variable length style will remove those elements, while array properties using the rigid style will replace removed elements with `null` elements. A service may indicate the maximum size of an array by padding `null` elements at the end of the array sequence.

When processing a PATCH request, the order of operations shall be:

- Modifications
- Deletions
- Additions

A PATCH request with fewer elements than currently exist in the array shall remove the remaining elements of the array.

For example, a fixed length style array of 'Flavors' indicates the service supports a maximum of six elements (by padding the array with `null` elements), with four populated.

```
"Flavors": [ "Chocolate", "Vanilla", "Mango", "Strawberry", null, null ]
```

A client could issue the following PATCH request to remove `Vanilla`, replace `Strawberry` with `Cherry`, and add `Coffee` and `Banana` to the array, while leaving the other elements unchanged.

```
"Flavors": [ {}, null, {}, "Cherry", "Coffee", "Banana" ]
```

The resulting array after the PATCH is:

```
"Flavors": [ "Chocolate", "Mango", "Cherry", "Coffee", "Banana", null ]
```

7.7. PUT (replace)

To completely replace a resource, services may support the PUT method. The service may add properties to the response resource that the client omits from the request body, the resource definition requires, or the service normally supplies.

The PUT operation should be idempotent in the absence of outside changes to the resource, with the possible exception that the operation might change ETag values.

When the replace operation succeeds, the response may contain a resource representation after the replacement occurs. See Modification success responses.

The following list contains the exception cases for PUT:

- If a service does not implement this method, the service shall return the HTTP 405 status code.
- Services may reject requests that do not include properties that the resource definition (schema) requires.
- If the client makes a PUT request against a Resource Collection, services should return the HTTP 405 status code.

7.8. POST (create)

To create a new resource, services shall support the POST method on Resource Collections.

The POST request is submitted to the Resource Collection to which the new resource will belong. When the create operation succeeds, the response may contain the new resource representation. See Modification success responses.

The body of the create request contains a representation of the object to create. The service may ignore any service-controlled properties, such as `Id`, which would force the service to overwrite those properties. Additionally, the service shall set the `Location` header in the response to the URI of the new 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 the addition of `Members` to a Resource Collection shall support both forms.
    - For example, if a client adds a new member to the Resource Collection at `/redfish/v1/EventService/Subscriptions`, it can send a POST request to either `/redfish/v1/EventService/Subscriptions` or `/redfish/v1/EventService/Subscriptions/Members`.
- If the service does not allow for the creation of new resources, the service shall return the HTTP 405 status code.
- The POST operation shall not be idempotent.
- Services may allow the inclusion of `@Redfish.OperationApplyTime` property in the request body. See Operation Apply Time.

7.9. DELETE (delete)

To remove a resource, the service shall support the DELETE method.

When the delete operation succeeds, the response may contain the resource representation after the deletion occurs. See Modification success responses.

- If the resource can never be deleted, the service shall return the HTTP 405 status code.
- If the resource was already deleted, the service may return HTTP status code 404 or a success code.
- The service may allow the inclusion of the `@Redfish.OperationApplyTime` property in the request body. See Operation Apply Time.

7.10. POST (Action)

Services shall support the POST method to send actions to Resources.

- The POST operation may not be idempotent.

- Services may allow the inclusion of the `@Redfish.OperationApplyTime` property in the request body. See Operation Apply Time.

To request actions on a Resource, send the HTTP POST method to the URI of the action. The `target` property in the Resource's `Actions` property shall contain the URI of the action. The URI of the action shall be in the format:

```
ResourceUri/Actions/QualifiedActionName
```

where

Variable	Description
ResourceUri	The URI of the resource that supports the action.
`Actions`	The name of the property that contains the actions for a resource, as defined by this specification.
QualifiedActionName	The qualified name of the action. Includes the namespace.

To determine the available actions and the valid parameter values for those actions, clients can query a Resource directly.

Clients provide parameters for the action as a JSON object within the request body of the POST operation. See the `Actions` property clause for information about the structure of the request and required parameters. Some parameter information may require that the client examine the Redfish Schema that corresponds to the Resource.

The service may ignore unsupported parameters provided by the client. 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.

To indicate the success or failure of the action request processing, the service may return a response with one of the following HTTP status codes and additional information:

To indicate	HTTP status code	Additional information
The action request succeeds.	200	The JSON message body, as described in Error responses, with a message that indicates 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 return the `Success` message from the Base

To indicate	HTTP status code	Additional information
		Message Registry in the `code` property in the response body.
The action request may require extra time to process.	202	A `Location` response header set to the URI of a Task Monitor.
The action request succeeds.	204	No JSON message body.
The client did not provide all required parameters.	400	The response may contain a JSON object, as described in Error responses, which details the error or errors.
The client provides a parameter that the service does not support, and the service does not ignore unsupported parameters.	400	The response may contain a JSON object, as described in Error responses, which details the error or errors.
An error was detected and the action request was not processed.	400 or greater	The response may contain a JSON object, as described in Error responses, which details the error or errors.

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 200 status code and return the `NoOperation` message from the Base Message Registry.

Example successful action response:

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

7.11. Operation apply time

Services may accept the `@Redfish.OperationApplyTime` annotation in the POST (create), DELETE (delete), or POST (action) request body. This annotation enables the client to control when an 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 occurs, 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 the service receives them.

Services that support the `@Redfish.OperationApplyTime` annotation for create and delete operations on a Resource Collection shall include the `@Redfish.OperationApplyTimeSupport` response annotation for the Resource Collection.

The following example response for a Resource Collection supports the `@Redfish.OperationApplyTime` annotation in the create and delete requests:

```
{
    "@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 previous example, a client can annotate their create request body on the `VolumeCollection` itself, or a delete operation on the `Volumes` within the `VolumeCollection`.

The following sample request deletes a `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 an action shall include the `@Redfish.OperationApplyTimeSupport` response annotation for the action.

The following example response for a `ComputerSystem` resource supports the `@Redfish.OperationApplyTime` annotation in the reset action request:

```
{
    "@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 previous example, a client can annotate their reset action request body on the `ComputerSystem` in the payload.

The following sample request completes 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` annotation for a Resource Collection or action shall create a Task, and respond with the HTTP 202 status code 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 `Settings` Redfish Schema defines the structure of the `@Redfish.OperationApplyTimeSupport` object and the `@Redfish.OperationApplyTime` annotation value.

8. Service responses

This clause describes the responses that Redfish Services can send to clients.

8.1. 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 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 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	The OData version of the payload to which the response conforms.
`Content-Type`	Yes	RFC 7231	The type of representation used in the message body. Services shall specify a `Content-Type` of `application/json` when returning resources as JSON, and `application/xml` when returning metadata as XML. `;charset=utf-8` 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	The encoding that has been performed on the media type.
`Content-Length`	No	RFC 7231	The size of the message body. An optional means of indicating size of the body uses `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 shall respond with status code 411.

Header	Required	Supported values	Description
`ETag`	Conditional	RFC 7232	An identifier for a specific version of a resource, often a message digest. The `ETag` header shall be included on responses to GETs of `ManagerAccount` resources.
`Server`	Yes	RFC 7231	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	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	Shall be supported and indicates whether a response can or cannot be cached.
`Via`	No	RFC 7230	Defines the network hierarchy and recognizes message loops. Each pass inserts its own `Via` header.
`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, HEAD	Shall be returned with a 405 (Method Not Allowed) response to indicate the valid methods for the request URI. Shall be returned with any GET or HEAD operation to indicate the other allowable operations for this resource.
`WWW-`	Yes	RFC 7235,	Required for Basic and other optional

Header	Required	Supported values	Description
`Authenticate`		Section 4.1	authentication mechanisms. See the Security details 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.

8.1.1. Link header

The Link header provides metadata information on the accessed resource in response to a HEAD or GET operation. The information can contain items 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.

```
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
```

- 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).

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.

8.2. Status codes

HTTP defines status codes that are used in responses. The status codes themselves provide general information about how the request was processed, such as whether the request was successful, if the client provided bad information, or the service encountered an error when performing the request.

- When the service returns a status code in the 4xx or 5xx range, services should return an extended error response in the response body.
    - This is to provide the client more meaningful and deterministic error semantics.
- When the service returns a status code in the 2xx range and the response contains a representation of a resource, services may use extended information to convey additional information about the resource.
- Extended error messages shall not provide privileged information when authentication failures occur.

NOTE: Refer to the Security details 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 the status codes in the table below as defined in description column.
- Redfish Services should not return the status code 100. Using the HTTP protocol for a multipass data transfer should be avoided, except for the upload of extremely large data.
- The HTTP 400 Bad Request status code should be used as the default status code for client-side errors if no other status code in the 4xx range is appropriate.
- The HTTP 500 Internal Server Error status code should be used as the default status code for service-side errors if no other status code in the 5xx range is appropriate.

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 was 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

HTTP Status Code	Description
	response body.
202 Accepted	The request has been accepted for processing, but the processing has not been completed. The `Location` header shall be set to the URI of a Task 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 service recognized the credentials in the request, but those credentials do not possess authorization to perform this request. This code is also returned when the user credentials provided need to be changed before access to the service can be granted. See the Security clause for more details.
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 URI in the client request.
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

HTTP Status Code	Description
	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 incompatible values).
410 Gone	The requested resource is no longer available at the service and no forwarding address is known. This condition is expected to be considered permanent. Clients with hyperlink editing capabilities should delete references to the URI in the client request after user approval. If the service 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 service 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 service 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 service does not (currently) support the functionality required to fulfill the request. This is the appropriate response when the service does not recognize the request method and is not capable of supporting the method for any resource.

HTTP Status Code	Description
503 Service Unavailable	The service is currently unable to handle the request due to temporary overloading or maintenance of the service. 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 of the service.
507 Insufficient Storage	The service is unable to build the response for the client due to the size of the response.

8.3. OData metadata responses

OData metadata describes resources, Resource Collections, capabilities, and service-dependent behavior to generic OData consumers 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, clients are not required to request metadata to request and interpret a JSON representation of a resource that this specification defines.

A client is able to access the OData metadata via the `/redfish/v1/$metadata` URI.

A client is able to access the OData service document via the `/redfish/v1/odata` URI.

8.3.1. OData $metadata

The OData metadata describes top-level service resources and resource types according to OData-Schema. The OData metadata is represented as an XML document with an `Edmx` root element in the `https://docs.oasis-open.org/odata/ns/edmx` namespace with an OData version attribute set to `4.0`.

The service shall return the OData metadata document as an XML document by using the `application/xml` or `application/xml;charset=utf-8` MIME types.

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

8.3.1.1. Referencing other schemas

The OData metadata shall include the namespaces for each of the Redfish Resource types, along with the `RedfishExtensions.v1_0_0` namespace.

These references may use either:

- The standard URI for the published Redfish Schema definitions, such as on `http://redfish.dmtf.org/schemas`.
- A URI to a local version of the Redfish Schema.

```
<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 OData metadata document shall include an `EntityContainer` that defines the top-level resources and Resource Collections.

8.3.1.2. Referencing OEM extensions

The OData metadata document may reference additional schema documents that describe OEM-specific extensions that the service uses.

For example, the OData metadata document may reference custom types for additional Resource Collections.

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

8.3.2. OData service document

The OData service document serves as a top-level entry point for generic OData clients. More information about the OData service document can be found in the OData JSON Format specification.

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

The service shall return the OData service document as a JSON object by using the `application/json` MIME type.

The JSON object shall contain the `@odata.context` context property set to `/redfish/v1/$metadata`.

The JSON object shall include a `value` property set to a JSON array that contains an entry for the service root and each resource that is a direct child of the service root.

Each JSON object entry includes:

Property	Defines
`name`	A user-friendly resource name of the resource.
`kind`	The type of resource. Value is `Singleton` for all cases defined by Redfish.
`url`	The relative URL for the top-level resource.

8.4. Resource responses

Services return resources and Resource Collections as JSON payloads by using the `application/`

`json` MIME type. A service shall not break responses for a single resource into multiple results.

The format of these payloads is defined by the Redfish schema. See the Data model and Schema definition languages clauses for rules about the Redfish schema, and how it maps to JSON payloads.

8.5. Error responses

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

To provide the client with as much information about the error as possible, a Redfish Service may provide multiple error responses in the HTTP response. 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.

An extended error response, which is a single JSON object, defines the error responses, with an `error` property, which contains the following properties.

Property	Description
`code`	String. Defines a `MessageId` from the Message Registry.
`message`	Displays a human-readable error message that corresponds to the Message in the Message Registry.
`@Message.ExtendedInfo`	Displays an array of Message objects. Describes one or more error messages.

```
{
    "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"
            }
        ]
    }
}
```


9. Data model

One of the key tenets of Redfish is the separation of protocol from the data model. This separation makes the data both transport and protocol agnostic. By concentrating on the data transported in the payload of the protocol (in HTTP, it is the HTTP body), Redfish is also able to define the payload in any encoding and the data model is intended to be schema language agnostic. While Redfish is defined using JSON, it is intended to provide a common encoding type and helps to ensure property naming conventions that make it easier for developers in many languages such as JavaScript and Python. All of this helps the Redfish data model to be more easily accessible in modern tools and programming environments.

This clause describes common data model, resource, and Redfish Schema requirements.

9.1. Resources

A Resource is a single entity. Services return resources as JSON payloads by using the `application/json` MIME type.

Each resource shall be strongly typed and defined in a Redfish schema document, and identified in the response payload by a unique type identifier property.

Responses for a single resource shall contain the following properties:

- `@odata.id`
  - Registry Resources are not required to provide `@odata.id`
- `@odata.type`
- `Id`
- `Name`

Responses may also contain other properties defined within the schema for that resource type. Responses shall not include any properties not defined by that resource type.

9.2. Resource Collections

A Resource Collection is a set of resources that share the same schema definition. Services return Resource Collections as JSON payloads by using the `application/json` MIME type.

Resource Collection responses shall contain the following properties:

- `@odata.id`
- `@odata.type`
- `Name`
- `Members`
- `Members@odata.count`

Responses for Resource Collections may contain the following properties:

- `@odata.context`
- `@odata.etag`
- `Description`
- `Members@odata.nextLink`
- `Oem`

Responses for Resource Collections shall not contain any other properties with the exception of payload annotations.

9.3. OEM Resources

OEMs and other third parties can extend the Redfish data model by creating new resource types. This is accomplished by defining an OEM schema for each resource type, and connecting instances of those Resources to the Resource Tree.

Companies, OEMs, and other organizations can define additional properties, hyperlinks, and actions for standard Redfish Resources using the `Oem` property in 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. OEMs are encouraged to follow the design tenets and naming conventions in this specification when defining OEM Resources or properties.

9.4. Common data types

The following clause details the data types found throughout the Redfish data model.

9.4.1. Primitive types

The following are the primitive data types found in the data model:

Type	Description
Boolean	This value can be `true` or `false`.
Number	A number with optional decimal point or exponent. Number properties may restrict the representation to be an integer or a number with decimal point.
String	A sequence of characters enclosed with double quotes (`"`).
Array	A comma separated set of the above types enclosed with square braces (`[` and `]`). See the Array properties clause for additional information.
Object	A set of properties enclosed with curly braces (`{` and `}`). See the Structured properties clause for additional information.
Null	The value `null`. This is used when the Service is unable to determine the property value, or if the schema has requirements around using `null` for other conditions.

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

9.4.2. GUID/UUID values

Globally Unique Identifier (GUID)/Universally Unique Identifier (UUID) values are unique identifier strings and shall use the format:

```
([0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})
```

9.4.3. Date-Time values

Date-Time values are strings according to the ISO 8601 extended format, including the time offset or UTC suffix. Date-Time values shall use the format:

```
YYYY-MM-DDThh:mm:ss[.SSS](Z|(+|-)HH:MM)
```

where

Variable or separator	Description
YYYY	The four digit year.
MM	The two digit month (1 to 12).
DD	The two digit day (1 to 31).
`T`	The time separator. Shall be a capital `T`.
hh	The two digit hour (0 to 23).
mm	The two digit minute (0 to 59).
ss	The two digit second (0 to 59).
SSS	The decimal fraction of a second. One or more digits where the number of digits implies the precision.
`Z`	The zero offset indicator. Shall be a capital `Z`.
HH	The two digit hour offset (0 to 23).
MM	The two digit minute offset (0 to 59).

For example, the value "`2015-03-13T04:14:33+06:00`" represents March 13, 2015 at 4:14:33 with a time offset of +06:00.

When the time of day is unknown or serves no purpose, the service shall report `00:00:00Z` for the time of day value.

9.4.4. Duration values

Duration values are strings according to the ISO 8601 duration format. Duration values shall use the

format:

```
P[yY][mM][wW][dD][T[hH][mM][s[.f]S]]
```

where

Variable	Description
y	The number of years.
m	The number of months.
w	The number of weeks.
d	The number of days.
h	The number of hours.
m	The number of minutes.
s	The number of seconds.
f	The fractional seconds.

Each field is optional and may contain more than one digit.

For example, the following values represent the following durations:

Value	Specifies a duration of
`P3D`	Three days.
`PT6H`	Six hours.
`PT10S`	Ten seconds.
`PT0.001S`	0.001 seconds.
`PT1H30M`	One hour and 30 minutes.

9.4.5. Reference properties

Reference properties are used to provide a reference to another Resource in the data model. Reference properties are JSON objects that contain an `@odata.id` property. The value of `@odata.id` is the URI of

the Resource being referenced.

9.4.6. Non-resource reference properties

Non-resource reference properties are used to provide a reference to services or documents that are not Redfish-defined resources. These properties shall include the term "Uri" in their property name (e.g. `AssemblyBinaryDataUri` in the Assembly schema). The access protocol and data format of the referenced URI may be defined in schema for that property. Non-resource reference properties that refer to local HTTP/S targets shall follow the Redfish protocol (including use of Redfish Sessions and access control), unless otherwise specified by the property definition in schema.

9.4.7. Array properties

Array properties contain a set of values or objects, and appear as JSON arrays within a response body. Array elements shall all contain values of the same data type.

There are three styles of arrays, regardless of the data type of the elements:

- A fixed length array has a static number of elements, with the array size specified in the property definition or chosen by the implementation.
- A variable length array, where the array size is not specified, will vary in size among instances, and may change in size. This is the most common style.
- A rigid array, where the array index is meaningful and therefore elements do not change position (index) in the array when elements are added or removed. Empty elements within a rigid array property shall be represented by `null` elements. An element removed from a rigid array shall be replaced by a `null` element and all other elements shall remain at their current index. Any array property using this style shall indicate the rigid style in the long description of its schema definition.

Services may pad an array property with `null` elements at the end of the sequence to indicate the array size to clients. This is useful for small fixed length arrays, and for variable or rigid arrays with a restrictive maximum size. Services should not pad array properties if the maximum array size is not restrictive. For example, an array property typically populated with two elements, that a service limits to a maximum of 16 elements, should not pad the array with 14 `null` elements.

9.4.8. Structured properties

Structured properties are JSON objects within a response body.

Some structured properties inherit from the `Resource.v1_0_0.ReferenceableMember` definition. Structured properties that follow this definition shall contain the `MemberId` and Resource identifier properties.

Because the definition of structured properties can evolve over time, clients need to be aware of the

inheritance model that the different structured property definitions use.

For example, the `Location` definition in the `Resource` schema has gone through several iterations since the `Resource.v1_1_0` namespace was introduced, and each iteration inherits from the previous version so that existing references in other schemas can leverage the new additions.

Structured property references need to be resolved for both local and external references.

A local reference is a Resource that has a structured property in its own schema, such as `ProcessorSummary` in the `ComputerSystem` Resource. In these cases, the type property for the Resource is the starting point for resolving the structured property definition.

To find the latest applicable version, clients can step the version of the Resource backwards.

For example, if a service returns `#ComputerSystem.v1_4_0.ComputerSystem` as the Resource type, a client can step backwards from `ComputerSystem.v1_4_0`, to `ComputerSystem.v1_3_0`, to `ComputerSystem.v1_2_0`, and so on, until it finds the `ProcessorSummary` structured property definition.

An external reference is a Resource that has a property that references a definition found in a different schema, such as `Location` in the `Chassis` Resource.

In these cases, clients can use the latest version of the external schema file as a starting point to resolve the structured property definition.

For example, if the latest version of the `Resource` schema is `1.6.0`, a client can go backward from `Resource.v1_6_0`, to `Resource.v1_5_0`, to `Resource.v1_4_0`, and so on, until it finds the `Location` structured property definition.

9.4.9. Message object

A `Message` object provides additional information about an object, property, or error response.

A `Message` object is a JSON object with the following properties:

Property	Type	Required	Defines
`MessageId`	String	Yes	The error or message, which is not to be confused with the HTTP status code. Clients can use this code to access a detailed message from a Message Registry.
`Message`	String	No	The human-readable error message that indicates the semantics associated with the

Property	Type	Required	Defines
			error. This shall be the complete message, and not rely on substitution variables.
`RelatedProperties`	An array of JSON pointers	No	The properties in a JSON payload that the message describes.
`MessageArgs`	An array of strings	No	The substitution parameter values for the message. If the parameterized message defines a `MessageId`, the service shall include the `MessageArgs` in the response.
`Severity`	String	No	The severity of the error.
`Resolution`	String	No	The recommended actions 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 that defines the complete human-readable error message.

`MessageIds` identify specific messages that a Message Registry defines.

The `MessageId` property value shall be in the format:

```
RegistryName.MajorVersion.MinorVersion.MessageKey
```

where

Variable	Description
`RegistryName`	The name of the registry. The registry name shall be Pascal-cased.
`MajorVersion`	A positive integer. The major version of the registry.
`MinorVersion`	A positive integer. The minor version of the registry.
`MessageKey`	The human-readable key into the registry. The message key shall be Pascal-cased and shall not include spaces, periods, or special characters.

To search the Message Registry for a corresponding message, the client can use the `MessageId`.

The Message Registry approach has advantages for internationalization, because the registry can be translated easily, and lightweight implementation because large strings need not be included with the implementation.

The use of `Base.1.0.GeneralError` as a `MessageId` in `ExtendedInfo` is discouraged. If no better message exists or the `ExtendedInfo` array contains multiple messages, use `Base.1.0.GeneralError` only in the `code` property of the `error` object.

When an implementation uses `Base.1.0.GeneralError` in `ExtendedInfo`, the implementation should include a `Resolution` property with this error to indicate how to resolve the problem.

9.5. Properties

Every property included in a Redfish response payload shall be defined in the schema for that resource. The following attributes apply to all property definitions:

- Property names in the request and response payload shall match the casing of the `Name` attribute value in the defining schema.
- Required properties shall always be returned in a response.
- Properties not returned from a GET operation indicate that the property is not supported by the implementation.
- If an implementation supports a property, it shall always provide a value for that property. If a value is unknown, then the value of `null` is an acceptable value if supported by the schema definition.
- A service may implement a writable property as read-only.

This clause also 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.

9.5.1. Resource identifier (@odata.id) property

Registry Resources in a response may include an `@odata.id` property. All other Resources in a response shall include an `@odata.id` property. The value of the identifier property shall be the Resource URI.

9.5.2. Resource type (@odata.type) property

All Resources in a response shall include an `@odata.type` type property. To support generic OData clients, all structured properties in a response should include an `@odata.type` type property. The value shall be a URL fragment that specifies the type of the resource and shall be in the format:

```
#Namespace.TypeName
```

where

Variable	Description
Namespace	The full namespace name of the Redfish Schema that defines the type. For Redfish Resources, the versioned namespace name.
TypeName	The name of the Resource type.

An example of a Resource type value is `#ComputerSystem.v1_0_0.ComputerSystem`, where `ComputerSystem.v1_0_0` denotes the version 1.0.0 namespace of `ComputerSystem`, and the type itself is `ComputerSystem`.

9.5.3. Resource ETag (@odata.etag) property

ETags enable clients to conditionally retrieve or update a Resource. Resources should include an `@odata.etag` property. For a Resource, the value shall be the ETag.

9.5.4. Resource context (@odata.context) property

Responses for a single Resource may contain an `@odata.context` property that describes 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 in the format:

```
/redfish/v1/$metadata#ResourceType
```

where *ResourceType* is the fully qualified name of the unversioned resource type. Redfish Resource definitions concatenate the Resource type namespace with a period (`.`) followed by the Resource type.

For example, the following context URL specifies that the results show a single `ComputerSystem` Resource:

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

```
    ...
}
```

The context URL for a Resource may be in one of the other formats specified by OData-Protocol.

9.5.5. 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. The `Id` property shall follow the definition for `Id` in the Resource schema.

9.5.6. 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. The `Name` property shall follow the definition for `Name` in the Resource schema.

9.5.7. Description

The `Description` property is used to convey a human-readable description of the resource. The `Description` property shall follow the definition for `Description` in the Resource schema.

9.5.8. MemberId

The `MemberId` property uniquely identifies an element within an array, where the element can be referenced by a reference property. The value of `MemberId` shall be unique across the array. The `MemberId` property shall follow the definition for `MemberId` in the Resource schema.

9.5.9. Count (Members@odata.count) property

The count property defines the total number of Resources, or members, that are available in a Resource Collection. The count property shall be named `Members@odata.count` and its value shall be the total number of members available in the Resource Collection. The `$top` or `$skip` query parameters shall not affect this count.

9.5.10. Members

The `Members` property of a Resource Collection identifies the members of the collection. The `Members` property is required and shall be returned in the response for any Resource Collection. The `Members` property shall be an array of JSON objects named `Members`. The `Members` property shall not be `null`. Empty collections shall be an empty JSON array.

9.5.11. Next link (Members@odata.nextLink) property

The value of the Next Link property shall be an opaque URL to a Resource, with the same `@odata.type`, which contains the next set of partial members from the original operation. 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 `Members@odata.count` property value is the total number of Resources available if the client enumerates all pages of the Resource Collection.

9.5.12. Links

The `Links` property represents the hyperlinks associated with the Resource, as defined by that Resource's 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.

The Links property shall be named `Links` and contain a property for each related Resource.

To navigate vendor-specific hyperlinks, the `Links` property shall also include an `Oem` property.

9.5.12.1. Reference to a related Resource

A reference to a single Resource is a JSON object that contains a single Resource identifier property. The name of this reference is the name of the relationship. The value of this reference is the URI of the referenced Resource.

```
{
    "Links": {
        "ManagedBy": {
            "@odata.id": "/redfish/v1/Chassis/Encl1"
        }
    }
}
```

9.5.12.2. References to multiple related Resources

A reference to a set of zero or more related Resources is an array of JSON objects. The name of this reference is the name of the relationship. Each element of the array is a JSON object that contains a Resource identifier property with the value of the URI of the referenced resource.

```
{
    "Links": {
        "Contains": [
            {
                "@odata.id": "/redfish/v1/Chassis/1"
            },
            {
                "@odata.id": "/redfish/v1/Chassis/Encl1"
            }
        ]
    }
}
```

9.5.13. Actions

The `Actions` property contains the actions supported by a Resource.

9.5.13.1. Action representation

Each supported action is represented as a property nested under `Actions`. The property name is constructed using the unique URI that identifies the action.

This property name shall be in the format

```
#ResourceType.ActionName
```

where

Variable	Description
ResourceType	The Resource where the action is defined.
ActionName	The name of the action.

The client may use this fragment to identify the action definition in the referenced Redfish Schema document.

The property for the action is a JSON object and contains the following properties:

- The `target` property shall be present, and defines the relative or absolute URL to invoke the action.
- The `title` property may be present,and defines the action's name.

The OData JSON Format specification defines the `target` and `title` properties.

To specify the list of supported values for a parameter, the service may include the `@Redfish.AllowableValues` annotation.

For example, the following property defines the `Reset` action for a `ComputerSystem`:

```
{
    "#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:

```
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"
}
```

The Resource may provide a separate `@Redfish.ActionInfo` Resource to describe the parameters and values that a particular instance or implementation supports. Use the `@Redfish.ActionInfo` annotation to specify the `ActionInfo` resource, which contains a URI to the `@Redfish.ActionInfo` resource for the action. See the Action Info annotation clause for details.

9.5.14. Oem

The `Oem` property is used for OEM extensions as defined in Resource Extensibility.

9.5.15. Status

The `Status` property represents the status of a Resource. The `Status` property shall follow the definition for `Status` in the Resource schema.

By having a common representation of status, clients can depend on consistent semantics. The `Status` property is capable of indicating the current state, health of the Resource, and the health of subordinate Resources.

9.6. Resource, schema, property, and URI 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.

Standard Redfish Resources defined and 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. The resource name is used to construct both the type identifier property and the schema file name for each of the supported schema description languages.

Standard Redfish properties follow similar naming conventions, and should use a common definition when defined in multiple schemas across the Redfish data model. This consistency allows for code re-use across resources and increases interoperability. Existing property definitions should be leveraged for new resource definitions whenever possible.

The Resource, schema, property, and URI naming rules are as follows:

- Resource Names, Property Names, and Enumerations shall be Pascal-cased.
- The first letter of each word in a name shall be uppercase and spaces between words shall be removed (e.g., `ComputerSystem`, `PowerState`, `SerialNumber`).
- Names shall not contain spaces or underscore characters.
- Both characters should be capitalized for two-character acronyms (e.g., `IPAddress`, `RemoteIP`).
- Only the first character of acronyms with three or more characters should be 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), the value should follow the capitalization commonly used for that name.
- Enumeration values should start with a letter, and be followed by letters, numbers, or

underscores.

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 and property names (alone or as a portion of a name), but should be `OEM` when used alone as an enumeration value.
- 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, a 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`, `ClearingLogic`)

In addition, the following rules apply to URIs defined in Redfish Schema:

- URI segments should generally follow the name of the Redfish Schema that defines the Resource located at each segment.
- URI segments for Resource Collections should use the plural form of the Resource Collection schema name, with the `Collection` term omitted (e.g., `Processors` for a `ProcessorCollection`).
- For Resources that contain hyperlinks to more than one Resource or ResourceCollection of the same schema type, the URI segments for those Resources should follow the name of the property that provides the hyperlink for clarity.
- If a hyperlink to a subordindate Resource is not found at the root of the Resource, the URI segments should contain the property path.
    - For example, for the `Certificates` hyperlink found in `ManagerNetworkProtocol` within the `HTTPS` object, `HTTPS` should be one of the URI segments.

9.7. Resource extensibility

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. All Redfish-specified resources include an empty structured property called `Oem` whose value can be used to

encapsulate one or more OEM-specified structured properties. This is predefined placeholder available to contain OEM-specific property definitions.

9.7.1. 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.

9.7.2. 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 would use 'Contoso' instead of 'Contoso_com', but Contoso.org would 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.

9.7.3. OEM resource naming and URIs

Companies, OEMs, and other organizations can define additional resources and link to them from an `Oem` property found in a standard Redfish Resource. To avoid naming collisions with current or future standard Redfish schema files, the defining organization's name should be prepended to the resource (schema) name. For example, `ContosoDisk` would not conflict with a `Disk` resource or another OEM's disk-related resource.

To avoid URI collisions with other OEM resources and future Redfish standard resources, the URIs for OEM resources within the Redfish Resource Tree shall be in the form of:

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

where

- *BaseUri* is the URI segment of the standard Redfish Resource starting with `/redfish/` where the `Oem` property is used. For example, `/redfish/v1/Systems/3AZ38944T523`.
- *OemName* is the name of the OEM, that follows the same naming as defined in the `Oem` property format and content clause.
- *ResourceName* is the name of the resource defined by the OEM.

For example, if Contoso defined a new resource called `ContosoAccountServiceMetrics` 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`.

9.7.4. 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"
        }
    },
    ...
}
```

9.7.5. OEM actions

OEM-specific actions appear in the JSON payload as properties of the `Oem` object, nested under an Actions property. The name of the property that represents the action shall follow the form `#*QualifiedActionName*`, where *QualifiedActionName* is the qualified name of the action, including namespace.

```
{
    "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.

9.8. Payload annotations

Resources, objects within a Resource, and properties may include additional annotations as properties with the name in the format:

```
[PropertyName]@Namespace.TermName
```

where

Variable	Description
PropertyName	The name of the property to annotate. If absent, the annotation applies to the entire JSON object, which may be an entire Resource.
Namespace	The name of the namespace that defines the annotation term.
TermName	The name of the annotation term to apply to the resource or property of the resource.

Services shall limit the annotation usage to the `odata`, `Redfish`, and `Message` namespaces. The OData JSON Format specification defines the `odata` namespace. The `Redfish` namespace is an alias for the `RedfishExtensions.v1_0_0` namespace.

The client can get the definition of the annotation from the OData $metadata document, the HTTP Link header, or may ignore the annotation entirely, but should not fail reading the resource due to

unrecognized annotations, including new annotations that the `Redfish` namespace defines.

9.8.1. Allowable values

To specify the list of allowable values for a parameter, clients can use the `@Redfish.AllowableValues` annotation for properties or action parameters.

To specify the set of allowable values, include a property with the name of the property or parameter, followed by `@Redfish.AllowableValues`. The property value is a JSON array of strings that define the allowable values for the parameter.

9.8.2. Extended information

The following clause describes the two methods of providing extended information.

9.8.2.1. Extended object information

To specify object-level status information, services may annotate a JSON object with the `@Message.ExtendedInfo` annotation.

```
{
    "@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.

9.8.2.2. Extended property information

Services may use `@Message.ExtendedInfo`, prepended with the name of the property to annotate an individual property in a JSON object with extended information:

```
{
    "@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."
        }
    ]
}
```

9.8.3. Action Info annotation

The Action Info annotation is used to convey the parameter requirements and allowable values on parameters for actions. This is done using `@Redfish.ActionInfo` term within the action representation. This term contains a URI to the `ActionInfo` Resource.

Example `#ComputerSystem.Reset` action with the `@Redfish.ActionInfo` annotation and Resource:

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

The `ResetActionInfo` Resource contains a more detailed description of the parameters and the supported values. This Resource follows the `ActionInfo` schema definition.

```
{
    "@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"
            ]
        }
    ]
}
```

9.9. Settings Resource

A Settings Resource is used to represent the future intended state of a Resource. Some Resources have properties that can be updated and the updates take place immediately; however, some properties need to be updated at a certain point in time, such as a system reset. While the Resource represents the current state, the Settings Resource represents the future intended state. The service represents properties of a Resource that can only be updated at a certain point in time using a `@Redfish.Settings` payload annotation. The Settings annotation contains a link to a subordinate Resource with the same schema definition. The properties within the Settings Resource contain the properties that are updated at a certain point in time.

For Resources that support a future state and configuration, the response shall contain a property with the `@Redfish.Settings` annotation. When a Settings annotation is used, the following conditions shall apply:

- The Settings Resource linked to current resource with the `@Redfish.Settings` annotation shall be of the same schema definition.
- The Settings Resource should be a subset of properties that can be updated.
- The Settings Resource shall not contain the `@Redfish.Settings` annotation.
- The Settings Resource may contain the `@Redfish.SettingsApplyTime` annotation.

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 indicates 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"
                ]
            }
        ]
    },
    ...
}
```

A client may indicate its preference on when to apply the future configuration by including the `@Redfish.SettingsApplyTime` annotation in the request body when updating the Settings Resource.

- If a service supports configuring when to apply the future settings, the Settings Resource shall contain a property with the `@Redfish.SettingsApplyTime` annotation.
- Only Settings Resources shall contain the `@Redfish.SettingsApplyTime` annotation.

Below is an example request body that shows a client configuring when the values in the Settings Resource are to be applied. In this case it is either on reset or during the specified maintenance window:

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

9.10. Special Resource situations

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

9.10.1. Absent resources

Resources may be 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 return a value of `Absent` for the `State` property in the `Status` object. 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 an 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 an single DIMM socket and the installed DIMM. When that DIMM is removed, the `Memory` resource remains as an absent resource to indicate the empty DIMM socket. Later, a new DIMM is installed in that socket, and the `Memory` resource represents data about this new DIMM, which could have completely different characteristics.

9.11. Registries

Registries are used in Redfish to optimize data being transferred from a Redfish Service.

Registry Resources are those Resources that assist the client in interpreting Redfish Resources beyond the Redfish Schema definitions. 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.

Redfish defines the following registry types:

- Message Registries. These are the most common and are used to take a MessageId and other message information to construct a message that can be presented to an end user. These are used in both eventing and in error responses to operations. More information on how a registry is used to construct messages can be found in the Error responses and Eventing clauses.
- BIOS Registries. A BIOS registry is used to determine the semantics about each of the properties contained in a BIOS Resource or the BIOS Settings Resource. Since BIOS information can vary from platform to platform, Redfish cannot define a fixed schema for these values. The registry contains not only a description of the properties, but other information such as data type, allowable values, and user menu information.
- Privilege Registries. These registries contain a mapping of the resources within the Redfish Service and which privileges are allowed to perform the specified operations against those

resource. This information allows a client to determine which roles should have specific privileges and thus map accounts to those roles to perform the desired operations on Redfish Resources. For more information on how privileges relate to roles, see the Privilege model/ Authorization clause.

9.12. Schema annotations

Schema annotations are used throughout the schema definitions of the data model to provide additional documentation for developers. This clause describes the different types of schema annotations used by the Redfish data model. See the Schema definition languages clause for information about how each of the annotations are implemented in their respective schema languages.

9.12.1. Description annotation

The Description annotation can be applied to any type, property, action, or parameter to provide a description of Redfish Schema elements suitable for end users or user interface help text.

A Description annotation shall be included on the following schema definitions:

- Redfish types
- Properties
- Reference properties
- Enumeration values
- Resources and Resource Collections
- Structured types

9.12.2. Long Description annotation

The Long Description annotation can be applied to any type, property, action, or parameter to provide a formal, normative specification of the schema element. Where the Long Descriptions in the Redfish Schema contains normative language, the service shall be required to conform with the statement.

A Long Description annotation shall be included on the following schema definitions:

- Redfish types
- Properties
- Reference properties
- Resources and Resource Collections
- Structured types

9.12.3. Resource Capabilities annotation

The Resource Capabilities annotation can be applied to Resources and Resource Collections to express the different type of HTTP operations a client is able to invoke on the given Resource or Resource

Collection.

- Insert Capabilities is used to indicate whether a client is able to perform a POST on the Resource.
- Update Capabilities is used to indicate whether a client is able to perform a PATCH or PUT on the Resource.
- Delete Capabilities is used to indicate whether a client is able to perform a DELETE on the Resource.
- A service may implement a subset of the capabilities that are allowed on the Resource or Resource Collection.

All schema definitions for Redfish Resources and Resource Collections shall include Resource Capabilities annotations.

9.12.4. Resource URI Patterns annotation

The Resource URI Patterns annotation is used to express the valid URI patterns for a given Resource or Resource Collection.

The strings for the URI patterns may use `{` and `}` characters to express parameters within a given URI pattern. 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 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 Resource, or the `MemberId` string property for the corresponding object within a Resource. The process for forming the strings that are concatonated to form the URI pattern are in the Resource, schema, property, and URI naming conventions clause.

The following string is an example URI pattern that describes a Manager Account Resource: `/redfish/v1/AccountService/Accounts/{ManagerAccountId}`

Using the example above, `{ManagerAccountId}` would be replaced by the `Id` property of the corresponding `ManagerAccount` resource. If the `Id` property for a given Manager Account resource is `John`, then the full URI for that resource would be `/redfish/v1/AccountService/Accounts/John`.

The URI patterns are constructed based on the formation of the Resource Tree. When constructing the URI pattern for a subordinate resource, the URI pattern for the current resource is used, and appended. For example, the `RoleCollection` Resource is subordinate to `AccountService`. Because the URI pattern for `AccountService` is `/redfish/v1/AccountService`, the URI pattern for the `RoleCollection` Resource will be `/redfish/v1/AccountService/Roles`.

In some cases, the subordinate resource is found inside of a structured property of a Resource. In these cases, the name of the structured property is used in the URI pattern for the subordinate resource. For example, the `CertificateCollection` Resource is subordinate to the `ManagerNetworkProtocol`

Resource from the `HTTPS` property. Because the URI pattern for `ManagerNetworkProtocol` is `/redfish/v1/Managers/{ManagerId}/NetworkProtocol`, the URI pattern for the `CertificateCollection` Resource will be `/redfish/v1/Managers/{ManagerId}/NetworkProtocol/HTTPS/Certificates`.

All schema definitions for Redfish Resources and Redfish Resource Collections shall be annotated with the Resource URI Patterns annotation.

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

9.12.5. Additional Properties annotation

The Additional Properties annotation is used to specify whether a type can contain additional properties outside of those defined in the schema. Types that do not support additional properties shall not contain properties beyond those described in the schema.

9.12.6. Permissions annotation

The Permissions annotation is used to specify if a client is allowed to modify the value of a property, or if the property is read-only.

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

9.12.7. Required annotation

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.

9.12.8. Required on Create annotation

The Required on Create annotation is used to specify that a property is required to be provided by the client on creation of the resource. Properties not annotated with the Required on Create annotation are not required to be provided by the client on a create operation.

9.12.9. Units of Measure annotation

In addition to following naming conventions, properties representing units of measure shall be annotated with the Units of Measure annotation to specify the units of measurement for the property.

The value of the annotation shall 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, the case-sensitive "(c/s)"

symbol for the prefix as listed in UCUM should be prepended to the unit symbol. For example, Mebibyte (1024^2 bytes), which has the UCUM prefix "Mi" and symbol "By", would use `MiBy` as the value for the annotation. 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`).

9.12.10. Expanded Resource annotation

The Expanded Resource annotation can be applied to a reference property to specify that the default behavior for the service is to include the contents of the related Resource or Resource Collection in responses. This behavior follows the same semantics of the expand query parameter with a level of 1.

Reference properties annotated with this term shall be expanded by the service, even if not requested by the client. A service may page Resource Collections.

9.12.11. Owning Entity annotation

The Owning Entity annotation can be applied to a schema in order to specify the name of the entity responsible for development, publication, and maintenance of a given schema.

9.13. Versioning

As stated previously, a Resource can be an individual entity or a Resource Collection, which acts as a container for a set of Resources.

A Resource Collection does not contain any version information because it defines a single `Members` property, and the overall collection definition never grows over time.

A Resource has both unversioned and versioned definitions.

The unversioned definition of a Resource is used in references from other Resources to ensure there are no version dependencies between the definitions. The unversioned definition of a Resource contains no property information about the Resource.

The versioned definition of a Resource contains a set of properties, actions, and other definitions associated with the given Resource. The version of a Resource follows the following format:

```
vX.Y.Z
```

where

Variable	Type	Version	Description
X	Integer	The major version	A backward-incompatible change.
Y	Integer	The minor version	A minor update. Redfish introduces new functionality but does not remove any functionality. The minor version preserves compatibility with earlier minor versions. For example, a new property introduces a new minor version of the resource.
Z	Integer	The errata version	A fix in an earlier version. For example, a fix to a schema annotation on a property introduces an errata version of the resource.

9.14. Localization

The creation of separate localized copies of Redfish schemas and registries is allowed and encouraged. Localized schema and registry files may be submitted to the DMTF for republication in the Redfish Schema Repository.

Property names, parameter names, and enumeration values in the JSON response payload are never localized, but translated copies of those names may be provided as additional annotations in the localized schema for use by client applications. A separate file for each localized schema or registry shall be provided for each supported language. The English-language versions of Redfish schemas and registries shall be the normative versions, and alterations of meaning due to translation in localized versions of schemas and registries shall be forbidden.

Schemas and registries in languages other than English shall identify their language using the appropriate schema annotations. Descriptive property, parameter, and enumeration text not translated into the language specified shall be removed from localized versions. This removal allows for software and tools to combine normative and localized copies, especially when minor schema version differences exist.

10. File naming and publication

For consistency in publication and to enable programmatic access, all Redfish-related files shall follow a set of rules to construct the name of each file. The file name construction rules for Redfish Schema files are contained in the Schema definition languages clause, while construction rules for other file types are shown below.

10.1. 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".

10.2. Profile file naming

The 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.

10.3. Dictionary file naming

The binary file describing a Redfish Device Enablement Dictionary follows the Redfish Schema file naming conventions for the schema definition language that the dictionary is converted from. As a single Dictionary file contains all minor revisions of the schema, only the major version is used in the file name. The file names for Dictionaries shall be formatted as:

DictionaryName_vMajorVersion.dict

For example, version 1.2.0 of the Chassis dictionary would be named "Chassis_v1.dict".

10.4. Localized file naming

Localized schemas and registries shall follow the same file naming conventions as the English language versions. When multiple localized copies are present in a repository (which will have the same file name), files in languages other than English shall be organized into subfolders named to match the ISO 639-1 language code for those files. English language files may be duplicated in an `en` subfolder for consistency.

10.5. DMTF Redfish file repository

All Redfish schemas, registries, dictionaries, and profiles published or republished by the DMTF's Redfish Forum are available from the DMTF website http://redfish.dmtf.org/ for download. Programs may access

the repository using the durable URLs listed below. 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.

Organizations creating Redfish-related files such as OEM schemas, Redfish Interoperability Profiles, or Message Registries are encouraged to submit those files to the DMTF using the form at https://redfish.dmtf.org/redfish/portal for republication in the DMTF Redfish file repository.

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/v1/{code}`	Durable URL for programmatic access to localized v1.xx schema files. Localized schemas are organized in subfolders using the 2-character ISO 639-1 language code as the {code} segment.
`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/v1/{code}`	Durable URL for programmatic access to localized v1.xx registry files. Localized schemas are organized in subfolders using the 2-character ISO 639-1 language code as the {code} segment.
`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.

URL	Folder contents
`redfish.dmtf.org/profiles/archive`	Subfolders contain profile files specific to a particular profile version or release.
`redfish.dmtf.org/dictionaries`	Durable URL for programmatic access to all v1.xx Redfish Device Enablement Dictionary files.
`redfish.dmtf.org/dictionaries/v1`	Durable URL for programmatic access to all v1.xx Redfish Device Enablement Dictionary files.
`redfish.dmtf.org/dictionaries/archive`	Subfolders contain dictionary files specific to a particular version release.

11. Schema definition languages

Individual resources and their dependent types and actions are defined within a Redfish schema document. This clause describes how these documents are constructed in the following formats:

- OData Common Schema Definition Language
- JSON Schema
- OpenAPI

11.1. OData Common Schema Definition Language

OData Common Schema Definition Language (CSDL) is an XML schema format defined by the OData CSDL specification. The following clause describes how Redfish uses CSDL to describe Resources and Resource Collections.

11.1.1. File naming conventions for CSDL

Redfish CSDL schema files shall be named using the TypeName 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:

TypeName_vMajorVersion.xml

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

11.1.2. Core CSDL files

The file `Resource_v1.xml` contains all base definitions for Resources, Resource Collections, and common properties such as `Status`.

The file `RedfishExtensions_v1.xml` contains the definitions for all Redfish types and annotations.

11.1.3. CSDL format

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>
```

The Referencing other CSDL files and CSDL Data Services clauses describe the items that are found within the `Edmx` element.

11.1.3.1. Referencing other CSDL files

CSDL files may reference types defined in other CSDL documents. This is done by including `Reference` tags.

The Reference element uses the `Uri` attribute to specify a CSDL file. The Reference element also contains one or more `Include` tags 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. Redfish CSDL files always use the `Alias` attribute on the following namespaces:

- `Org.OData.Core.V1` is aliased as `OData`.
- `Org.OData.Measures.V1` is aliased as `Measures`.
- `RedfishExtensions.v1_0_0` is aliased as `Redfish`.
- `Validation.v1_0_0` is aliased as `Validation`.

```
  <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:Include Namespace="Validation.v1_0_0" Alias="Validation"/>
  </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>
```

11.1.3.2. CSDL Data Services

Structures, enums, and other definitions are defined within a namespace in CSDL. The namespace is defined through a `Schema` tag and using the `Namespace` attribute to declare the name of the namespace. Redfish uses namespaces to differentiate different versions of the schema. CSDL allows for structures to inherit from other structures, which allows for newer namespaces to only define the additional definitions. This behavior is described further in the Elements of CSDL namespaces clause.

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

```
  <edmx:DataServices>
    <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="MyTypes.v1_0_0">
      <!-- Type definitions for version 1.0.0 of MyTypes go here -->
    </Schema>

    <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="MyTypes.v1_1_0">
      <!-- Type definitions for version 1.1.0 of MyTypes go here -->
    </Schema>
  </edmx:DataServices>
```

11.1.4. Elements of CSDL namespaces

The following clauses describe the different definitions that can be found within each namespace.

11.1.4.1. Qualified names

Many definitions in CSDL use references to qualified names. CSDL defines this as a string in the form of "*Namespace*.*TypeName*", where:

- *Namespace* is the name of the namespace.
- *TypeName* is the name of the element contained within the namespace.

For example, if a reference is being made to `MyType.v1_0_0.MyDefinition`, this means the definition can be found in the `MyType.v1_0_0` namespace with an element of the name `MyDefinition`.

11.1.4.2. Entity Type and Complex Type elements

The Entity Type and Complex Type elements are defined using the `EntityType` and `ComplexType` tags respectively. These elements are used to define a JSON structure and their set of properties. This is done by defining Property elements and Navigation Property element within the `EntityType` or `ComplexType` tags.

All Entity Types and Complex Types contain a `Name` attribute, which specifies the name of the definition.

Entity Types and Complex Types may have a `BaseType` attribute, which specifies a qualified name. When the `BaseType` attribute is used, all of the definitions of the referenced `BaseType` are made available to the Entity Type or Complex Type being defined.

All Resources and Resource Collections are defined with the Entity Type element. Resources inherit from `Resource.v1_0_0.Resource`, and Resource Collections inherit from `Resource.v1_0_0.ResourceCollection`.

Most structured properties are defined with the Complex Type element. Some are defined using the Entity Type element that inherits from `Resource.v1_0_0.ReferenceableMember`. The Entity Type element allows for references to be made using the Navigation Property element, whereas Complex Type element does not allow for this usage.

Example Entity Type and Complex Type element:

```
  <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 Navigation Property definitions go here -->

  </EntityType>

  <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 Navigation Property definitions go here -->

  </ComplexType>
```

11.1.4.3. Action element

The Action element is defined using the `Action` tag. This element is used to define an action that can be performed on a Resource.

All Action elements contain a `Name` attribute, which specifies the name of the action. The action shall be represented in payloads as the qualified name of the action, preceeded by a "#".

In Redfish, all Action elements contain the `IsBound` attribute and is always set to `true`. This is used to indicate that the action appears as a member of a given structured type.

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

Because all Action elements in Redfish use the term `IsBound="true"`, the first parameter is called the "binding parameter" and specifies the structured type to which the action belongs. In Redfish, this is always going to be one of the following Complex Type elements:

- For standard actions, the `Actions` Complex Type for the Resource.
- For OEM actions, the `OemActions` Complex Type for 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.

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

  <ComplexType Name="Actions">
    ...
  </ComplexType>
  ...
</Schema>
```

11.1.4.3.1. Action element for OEM actions

OEM-specific actions shall be defined by using the Action element with the binding parameter set to the `OemActions` Complex Type for the Resource. For example, the following definition defines the OEM `#Contoso.Ping` action for a `ComputerSystem`.

```
<Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Contoso">
  <Action Name="Ping" IsBound="true">
    <Parameter Name="ComputerSystem" Type="ComputerSystem.v1_0_0.OemActions"/>
  </Action>
</Schema>
```

11.1.4.4. Property element

Properties of Resources, Resource Collections, and structured properties are defined using the Property element. The `Property` tag is used to define a Property element inside of Entity Type and Complex Type elements.

All Property elements contain a `Name` attribute, which specifies the name of the property.

All Property elements contain a `Type` attribute specifies the data type. The `Type` attribute shall be one of the following:

- A qualified name that references an Enum Type element.
- A qualified name that references a Complex Type element.
- A primitive data type.
- An array of any of the above using the `Collection` term.

Primitive data types shall be one of the following:

Type	Meaning
`Edm.Boolean`	True or False.
`Edm.DateTimeOffset`	Date-time string.
`Edm.Decimal`	Numeric values with fixed precision and scale.
`Edm.Double`	IEEE 754 binary64 floating-point number (15-17 decimal digits).
`Edm.Duration`	Duration string.
`Edm.Guid`	GUID/UUID string.
`Edm.Int64`	Signed 64-bit integer.
`Edm.String`	UTF-8 string.

Property elements may specify a `Nullable` attribute. If the value is `false`, value of `null` is not allowed as a value for the property. If the attribute is `true` or not specified, a value of `null` is allowed.

Example Property element:

```
  <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>
```

11.1.4.5. Navigation Property element

Reference properties of Resources, Resource Collections, and structured properties are defined using the Navigation Property element. The `NavigationProperty` tag is used to define a Navigation Property element inside of Entity Type and Complex Type elements.

All Navigation Property elements contain a `Name` attribute, which specifies the name of the property.

All Navigation Property elements contain a `Type` attribute specifies the data type. The `Type` attribute is a qualified name that references an Entity Type element. This can also be made into an array using the `Collection` term.

Navigation Property elements may specify a `Nullable` attribute. If the value is `false`, `null` is not allowed as a value for the property. If the attribute is `true` or not specified, a value of `null` is allowed.

Unless the reference property is to be expanded, all Navigation Properties in Redfish use the `OData.AutoExpandReferences` Annotation element to show that the reference is always available.

Example Navigation Property element:

```
  <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>
```

11.1.4.6. Enum Type element

The Enum Type element is defined using the `EnumType` tag. This element is used to define a set of enumeration values, which may be applied to one or more properties.

All Enum Types contain a `Name` attribute, which specifies the name of the set of enumeration values.

Enum Types 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>
```

11.1.4.7. Annotation element

Annotations in CSDL are expressed using the `Annotation` element. The `Annotation` element can be applied to any schema element in CSDL. The following examples show how each of the different schema annotations used by Redfish are expressed in CSDL.

Terms with the prefix `OData` are defined in the OData Core Schema.

Terms with the prefix `Measures` are defined in OData Measures Schema.

Terms with the prefix `Redfish` are defined in RedfishExtensions Schema.

Example Description annotation:

```
  <Annotation Term="OData.Description" String="This property contains the user name
for the account."/>
```

Example Long Description annotation:

```
  <Annotation Term="OData.LongDescription" String="This property shall contain the
user name for the account."/>
```

Example Additional Properties annotation:

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

Example Permissions annotation (Read Only):

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

Example Permissions annotation (Read/Write):

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

Example Required annotation:

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

Example Required on Create annotation:

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

Example Units of Measure annotation:

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

Example Expanded Resource annotation:

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

Example Insert Capabilities Annotation (showing POST is not allowed):

```
<Annotation Term="Capabilities.InsertRestrictions">
  <Record>
    <PropertyValue Property="Insertable" Bool="false"/>
  </Record>
</Annotation>
```

Example Update Capabilities Annotation (showing PATCH and PUT are allowed):

```
  <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>
```

Example Delete Capabilities Annotation (showing DELETE is allowed):

```
  <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>
```

Example Resource URI Patterns Annotation:

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

Example Owning Entity Annotation:

```
  <Annotation Term="Redfish.OwningEntity" String="DMTF"/>
```

11.2. JSON Schema

The JSON Schema specification defines a JSON format for describing JSON payloads. The following clause describes how Redfish uses JSON Schema to describe Resources and Resource Collections.

11.2.1. File naming conventions for JSON Schema

Versioned Redfish JSON Schema files shall be named using the TypeName, following the format:

ResourceTypeName.vMajorVersion_MinorVersion_Errata.json

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

Unversioned Redfish JSON Schema files shall be named using the TypeName, following the format:

ResourceTypeName.json

For example, the unversioned definition of the Chassis schema would be named "Chassis.json".

11.2.2. Core JSON Schema files

The file `odata-v4.json` contains the definitions for common OData properties.

The file `redfish-error.v1_0_0.json` contains the payload definition of the Redfish error response.

The file `redfish-schema-v1.json` contains extensions to the JSON Schema used to define Redfish JSON Schema files.

The file `Resource.json` and its subsequent versioned definitions contain all base definitions for Resources, Resource Collections, and common properties such as `Status`.

11.2.3. JSON Schema format

Each JSON Schema file contains a JSON object to describe Resources, Resource Collections, or other definitions for the data model. The following terms can be found in the JSON object:

- `$id`: A reference to the URI where the schema file is published.
- `$ref`: If the schema file describes a Resource or Resource Collection, this is a reference to the structural definition of the Resource or Resource Collection.
- `$schema`: A URI to the Redfish schema extensions for JSON Schema. The value should be `http://redfish.dmtf.org/schemas/v1/redfish-schema-v1.json`.
- `copyright`: The copyright statement for the organization producing the JSON Schema.
- `definitions`: Contains structures, enumerations, and other definitions defined by the schema.
- `title`: If the schema file describes a Resource or Resource Collection, this shall be the matching type identifier for the Resource or Resource Collection.

11.2.4. JSON Schema definitions body

This clause describes the types of definitions that can be found in the `definitions` term of a Redfish JSON Schema file.

11.2.4.1. Resource definitions in JSON Schema

To satisfy versioning requirements, the JSON Schema representation of each Resource has one unversioned schema file, and a set of versioned schema files.

The unversioned definition of a Resource contains an `anyOf` statement. This statement consists of an array of `$ref` terms, which point to the following:

- The JSON Schema definition for a reference property.
- The versioned definitions of the Resource.

The unversioned definition of a Resource also uses the `uris` term to express the allowable URIs for the resource, and the `deletable`, `insertable`, and `updatable` terms to express the capabilities of the resource.

Example unversioned Resource definition in JSON Schema:

```
{
    "ComputerSystem": {
        "anyOf": [
            {
                "$ref": "http://redfish.dmtf.org/schemas/v1/
odata.v4_0_3.json#/definitions/idRef"
            },
            {
                "$ref": "http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_0_0.json#/definitions/ComputerSystem"
            },
            {
                "$ref": "http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_0_1.json#/definitions/ComputerSystem"
            },
            {
                "$ref": "http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_6_0.json#/definitions/ComputerSystem"
            }
        ],
        "deletable": true,
        "description": "The ComputerSystem schema represents a general purpose machine
or system.",
        "insertable": false,
        "longDescription": "This resource shall be used to represent resources that
represent a computing system.",
        "updatable": true,
        "uris": [
            "/redfish/v1/Systems/{ComputerSystemId}"
```

```
        ]
    },
    ...
}
```

The versioned definition of a Resource contains the property definitions for the given version of the Resource.

11.2.4.2. Enumerations in JSON Schema

Definitions for enumerations can consist of the following terms:

- `enum`: A string array that contains the possible enumeration values.
- `enumDescriptions`: An object that contains the descriptions for each of the enumerations as name-value pairs.
- `enumLongDescriptions`: An object that contains the long descriptions for each of the enumerations as name-value pairs.
- `type`: Because all enumerations in Redfish are strings, the `type` term always has the value `string`.

Example enumeration definition in JSON Schema:

```
{
    "IndicatorLED": {
        "enum": [
            "Lit",
            "Blinking",
            "Off"
        ],
        "enumDescriptions": {
            "Blinking": "The Indicator LED is blinking.",
            "Lit": "The Indicator LED is lit.",
            "Off": "The Indicator LED is off."
        },
        "enumLongDescriptions": {
            "Blinking": "This value shall represent the Indicator LED is in a blinking
state where the LED is being turned on and off in repetition.",
            "Lit": "This value shall represent the Indicator LED is in a solid on
state.",
            "Off": "This value shall represent the Indicator LED is in a solid off
state."
        },
        "type": "string"
    },
```

```
    ...
}
```

11.2.4.3. Actions in JSON Schema

Versioned definitions of Resources contain a definition called `Actions`. This definition is a container with a set of terms that point to the different actions supported by the resource. The names of standard actions shall be in the form of "*#ResourceType.ActionName*".

Example `Actions` definition:

```
{
    "Actions": {
        "additionalProperties": false,
        "description": "The available actions for this resource.",
        "longDescription": "This type shall contain the available actions for this
resource.",
        "properties": {
            "#ComputerSystem.Reset": {
                "$ref": "#/definitions/Reset"
            }
        },
        "type": "object"
    },
    ...
}
```

Another definition within the same schema file is used to describe the action itself. This definition contains a term called `parameters` to describe the client request body. It also contains property definitions for the `target` and `title` properties shown in service response payloads for the Resource.

Example definition of an action:

```
{
    "Reset": {
        "additionalProperties": false,
        "description": "This action is used to reset the system.",
        "longDescription": "This action shall perform a reset of the ComputerSystem.",
        "parameters": {
            "ResetType": {
                "$ref": "http://redfish.dmtf.org/schemas/v1/Resource.json#/definitions/
ResetType",
```

```
                "description": "The type of reset to be performed.",
                "longDescription": "This parameter shall define the type of reset to
be performed."
            }
        },
        "properties": {
            "target": {
                "description": "Link to invoke action",
                "format": "uri",
                "type": "string"
            },
            "title": {
                "description": "Friendly action name",
                "type": "string"
            }
        },
        "type": "object"
    },
    ...
}
```

11.2.4.3.1. OEM actions in JSON Schema

OEM-specific actions shall be defined by using an action definition in an appropriately named JSON Schema file. For example, the following definition defines the OEM `#Contoso.Ping` action, assuming it's found in the versioned Contoso JSON Schema file with a name such as "Contoso.v1_0_0.json".

```
{
    "Ping": {
        "additionalProperties": false,
        "parameters": {},
        "properties": {
            "target": {
                "description": "Link to invoke action",
                "format": "uri",
                "type": "string"
            },
            "title": {
                "description": "Friendly action name",
                "type": "string"
            }
        },
        "type": "object"
    },
    ...
```

```
}
```

11.2.5. JSON Schema terms used by Redfish

The following JSON Schema terms are used to provide schema annotations for Redfish JSON Schema:

- `description` and `enumDescriptions` are used for the Description annotation.
- `longDescription` and `enumLongDescriptions` are used for the Long Description annotation.
- `additionalProperties` is used for the Additional Properties annotation.
- `readonly` is used for the Permissions annotation.
- `required` is used for the Required annotation.
- `requiredOnCreate` is used for the Required on Create annotation.
- `units` is used for the Units of Measure annotation.
- `autoExpand` is used for the Expanded Resource annotation.
- `deletable`, `insertable`, and `updatable` are used for the Resource Capabilities Annotation.
- `uris` is used for the Resource URI Patterns Annotation.
- `owningEntity` is used for the Owning Entity Annotation.

11.3. OpenAPI

The OpenAPI specification defines a format for describing JSON payloads, as well as the set of URIs a client is allowed to access on a service. The following clause describes how Redfish uses OpenAPI to describe Resources and Resource Collections.

11.3.1. File naming conventions for OpenAPI Schema

Versioned Redfish OpenAPI files shall be named using the TypeName, following the format:

ResourceTypeName.vMajorVersion_MinorVersion_Errata.yaml

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

Unversioned Redfish OpenAPI files shall be named using the TypeName, following the format:

ResourceTypeName.yaml

For example, the unversioned definition of the Chassis schema would be named "Chassis.yaml".

11.3.2. Core OpenAPI Schema files

The file `odata-v4.yaml` contains the definitions for common OData properties.

The file `openapi.yaml` contains the URI paths and their respective payload structures.

The file `Resource.yaml` and its subsequent versioned definitions contain all base definitions for Resources, Resource Collections, and common properties such as `Status`.

11.3.3. openapi.yaml

The YAML file `openapi.yaml` is the starting point for clients to understand the construct of the service. It contains the following terms:

- `components`: Contains global definitions. For Redfish, this is used to contain the format of the Redfish error response.
- `info`: A structure consisting of information about what the `openapi.yaml` is describing, such as the author of the file and any contact information.
- `openapi`: The version of OpenAPI the document follows.
- `paths`: The URIs supported by the document, along with possible methods, response bodies, and request bodies.

The service shall return the `openapi.yaml`, if present in the Redfish Service, as a YAML document by using either the `application/yaml` or `application/vnd.oai.openapi` MIME types. The Service may append `;charset=utf-8` to the MIME type. Note that while the `application/yaml` type is in common use today, the `application/vnd.oai.openapi` type was recently defined and approved specifically to support OpenAPI. Implementations should use caution when selecting the MIME type as this specification may change in the future to reflect adoption of the OpenAPI-specific MIME type.

The `paths` term contains an array of the possible URIs. For each URI, it also lists the possible methods. For each method, it lists the possible response bodies and request bodies.

Example `paths` entry for a Resource:

```
/redfish/v1/Systems/{ComputerSystemId}:
  get:
    parameters:
    - description: The value of the Id property of the ComputerSystem resource
      in: path
      name: ComputerSystemId
      required: true
      schema:
        type: string
    responses:
      '200':
        content:
          application/json:
            schema:
```

```
                $ref: http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_6_0.yaml#/components/schemas/ComputerSystem
          description: The response contains a representation of the ComputerSystem
            resource
        default:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/RedfishError'
          description: Error condition
```

Example `paths` entry for an action:

```
  /redfish/v1/Systems/{ComputerSystemId}/Actions/ComputerSystem.Reset:
    post:
      parameters:
      - description: The value of the Id property of the ComputerSystem resource
        in: path
        name: ComputerSystemId
        required: true
        schema:
          type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_6_0.yaml#/components/schemas/ResetRequestBody
        required: true
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/RedfishError'
          description: The response contains the results of the Reset action
        '202':
          content:
            application/json:
              schema:
                $ref: http://redfish.dmtf.org/schemas/v1/Task.v1_4_0.yaml#/components/
schemas/Task
          description: Accepted; a Task has been generated
        '204':
          description: Success, but no response data
```

```
        default:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/RedfishError'
          description: Error condition
```

11.3.4. OpenAPI file format

With the exception of `openapi.yaml`, each OpenAPI file contains a YAML object to describe Resources, Resource Collections, or other definitions for the data model. The following terms can be found in the YAML object:

- `components`: Contains structures, enumerations, and other definitions defined by the schema.
- `x-copyright`: The copyright statement for the organization producing the OpenAPI file.
- `title`: If the schema file describes a Resource or Resource Collection, this shall be the matching type identifier for the Resource or Resource Collection.

11.3.5. OpenAPI components body

This clause describes the types of definitions that can be found in the `components` term of a Redfish OpenAPI file.

11.3.5.1. Resource definitions in OpenAPI

To satisfy versioning requirements, the OpenAPI representation of each Resource has one unversioned schema file, and a set of versioned schema files.

The unversioned definition of a Resource contains an `anyOf` statement. This statement consists of an array of `$ref` terms, which point to the following:

- The OpenAPI definition for a reference property.
- The versioned definitions of the Resource.

Example unversioned Resource definition in OpenAPI:

```
    ComputerSystem:
      anyOf:
      - $ref: http://redfish.dmtf.org/schemas/v1/odata.v4_0_3.yaml#/components/schemas/
idRef
      - $ref: http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_0_0.yaml#/components/schemas/ComputerSystem
```

```
      - $ref: http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_0_1.yaml#/components/schemas/ComputerSystem
      - $ref: http://redfish.dmtf.org/schemas/v1/
ComputerSystem.v1_6_0.yaml#/components/schemas/ComputerSystem
      description: The ComputerSystem schema represents a general purpose machine
        or system.
      x-longDescription: This resource shall be used to represent resources that
represent
        a computing system.
```

The versioned definition of a Resource contains the property definitions for the given version of the Resource.

11.3.5.2. Enumerations in OpenAPI

Definitions for enumerations can consist of the following terms:

- `enum`: A string array that contains the possible enumeration values.
- `type`: Since all enumerations in Redfish are strings, the `type` term always has the value `string`.
- `x-enumDescriptions`: An object that contains the descriptions for each of the enumerations as name-value pairs.
- `x-enumLongDescriptions`: An object that contains the long descriptions for each of the enumerations as name-value pairs.

Example enumeration definition in OpenAPI:

```
    IndicatorLED:
      enum:
      - Lit
      - Blinking
      - 'Off'
      type: string
      x-enumDescriptions:
        Blinking: The Indicator LED is blinking.
        Lit: The Indicator LED is lit.
        'Off': The Indicator LED is off.
      x-enumLongDescriptions:
        Blinking: This value shall represent the Indicator LED is in a blinking state
          where the LED is being turned on and off in repetition.
        Lit: This value shall represent the Indicator LED is in a solid on state.
        'Off': This value shall represent the Indicator LED is in a solid off state.
```

11.3.5.3. Actions in OpenAPI

Versioned definitions of Resources contain a definition called `Actions`. This definition is a container with a set of terms that point to the different actions supported by the resource. The names of standard actions shall be in the form of "*#ResourceType.ActionName*".

Example `Actions` definition:

```
    Actions:
      additionalProperties: false
      description: The available actions for this resource.
      properties:
        '#ComputerSystem.Reset':
          $ref: '#/components/schemas/Reset'
      type: object
      x-longDescription: This type shall contain the available actions for this
resource.
```

Another definition within the same schema file is used to describe the action itself. This definition contains property definitions for the `target` and `title` properties shown in service response payloads for the Resource.

Example definition of an action:

```
    Reset:
      additionalProperties: false
      description: This action is used to reset the system.
      properties:
        target:
          description: Link to invoke action
          format: uri
          type: string
        title:
          description: Friendly action name
          type: string
      type: object
      x-longDescription: This action shall perform a reset of the ComputerSystem.
```

The parameters for the action are shown in another definition with `RequestBody` appended to the name of the action. This gets mapped from the `openapi.yaml` file for expressing the POST method for the URI of the action.

Example definition of parameters of an action:

```
    ResetRequestBody:
      additionalProperties: false
      description: This action is used to reset the system.
      properties:
        ResetType:
          $ref: http://redfish.dmtf.org/schemas/v1/Resource.yaml#/components/schemas/
ResetType
          description: The type of reset to be performed.
          x-longDescription: This parameter shall define the type of reset to be
performed.
      type: object
      x-longDescription: This action shall perform a reset of the ComputerSystem.
```

11.3.5.3.1. OEM actions in OpenAPI

OEM-specific actions shall be defined by using an action definition in an appropriately named OpenAPI file. For example, the following definition defines the OEM `#Contoso.Ping` action, assuming it's found in the versioned Contoso OpenAPI file with a name such as "Contoso.v1_0_0.yaml".

```
    Ping:
      additionalProperties: false
      properties:
        target:
          description: Link to invoke action
          format: uri
          type: string
        title:
          description: Friendly action name
          type: string
      type: object
    PingRequestBody:
      additionalProperties: false
      properties: {}
      type: object
```

11.3.6. OpenAPI terms used by Redfish

The following OpenAPI terms are used to provide schema annotations for Redfish OpenAPI files:

- `description` and `x-enumDescriptions` are used for the Description annotation.
- `x-longDescription` and `x-enumLongDescriptions` are used for the Long Description annotation.
- `additionalProperties` is used for the Additional Properties annotation.
- `readOnly` is used for the Permissions annotation.

- `required` is used for the Required annotation.
- `x-requiredOnCreate` is used for the Required on Create annotation.
- `x-units` is used for the Units of Measure annotation.
- `x-autoExpand` is used for the Expanded Resource annotation.
- `x-owningEntity` is used for the Owning Entity Annotation.

11.4. Schema modification rules

Schema referenced from the implementation 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 constrain the capabilities of a Resource or Resource Collection to remove support for HTTP operations.
- Modified schema may remove properties.
- Modified schema may change any external references to point to Redfish Schema that adheres to the modification rules.
- Modified schema may change the Owning Entity annotation in order to specify who made the modifications.
- Other modifications to the Schema shall not be allowed.

12. Service details

12.1. Eventing

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

NOTE: Refer to the Security details clause for security implications of Eventing.

12.1.1. 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.

12.1.1.1. 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 after 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 shall 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.
- Services shall not send a "push" event payload of more than 1 Mebibyte (1 MiB). If there is more than 1 MiB worth of data to send the service shall divide the payload on the nearest Event entry such that the total payload transmitted to the client is less than 1 MiB. This restriction shall not apply to Metric Reports.

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.

12.1.1.2. SSE method

A service may support the `ServerSentEventUri` property within the Event Service. 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: Event Service clause for details on this method.

12.1.2. 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.

12.1.3. Ways to register for events

Event subscriptions can be subscribed to by specifying a RegistryPrefixes, ResourceTypes, OriginResources (including SubordinateResources) property 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.

12.1.4. 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.

12.1.4.1. 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 characters.

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.

12.1.4.2. Metric Report message objects

Metric Report message objects sent to the specified client endpoint shall contain the properties as described in the Redfish Metric Report Schema.

12.1.5. 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 want 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 to provide the OEM section within the registry but shall not change the standard values (such as Messages) in such registries.

12.2. 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.
- Services shall persist pending Tasks produced by client requests containing "@Redfish.OperationApplyTime" across service restarts, until the Task begins execution.
- Tasks that are pending execution should include the "@Redfish.OperationApplyTime" property to indicate when the Task will start. If the OperationApplyTime value is AtMaintenanceWindowStart or InMaintenanceWindowOnReset, then the task should also include the "@Redfish.MaintenanceWindow" property.

12.3. Resource Tree stability

The Resource Tree, which is defined as the set of URIs and array elements within the implementation, should be consistent on a single service across device reboot and A/C power cycle, and should 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 should 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.

12.4. Discovery

Automatic discovery of managed devices supporting Redfish 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.

12.4.1. 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.

12.4.2. 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 '::dmtf-org'

12.4.3. M-SEARCH response

The Redfish Service Search Target (ST) is defined as: urn:dmtf-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:
```

12.4.4. 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, shall 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.

12.5. 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.

Successful Resource responses for SSE shall:

- Return the HTTP 200 status code.
- Have a `Content-Type` header set as `text/event-stream` or `text/event-stream;charset=utf-8`.

Unsuccessful Resource responses for SSE shall:

- Return an HTTP status code of 400 or greater.
- Have a `Content-Type` header set as `application/json` or `application/json;charset=utf-8`.
- Contain a JSON object in the response body, as described in Error responses, which details the error or errors.

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.

12.5.1. Event Service

A service's implementation of the Event Service 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 Event Service, the service shall create an Event Destination Resource in the Subscriptions collection for the Event Service to represent the connection. The `Context` property in the Event Destination Resource shall be an opaque string generated by the service. The service shall delete the corresponding Event Destination Resource when the connection is closed. The service shall close the connection if the corresponding Event Destination Resource is deleted.

There are two formats of SSE streams:

- Metric Report SSE stream: This format shall be used when the Telemetry Service 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.

To reduce the amount of data returned to the client, the Service should support the `$filter` query parameter in the URI for the SSE stream.

Note: The `$filter` syntax shall follow the format in the Query parameters for Filter clause.

The Service should support these properties as filter criteria:

- `EventFormatType`

  The Service sends events of the matching `EventFormatType`.

  Example:

  ```
  https://sseuri?$filter=EventFormatType eq Event
  ```

  Valid values are the `EventFormatType` enumerated string values that the Redfish Event Service Schema defines.

- `EventType`

  The Service sends events of the matching `EventType`.

  Example:

  ```
  https://sseuri?$filter=EventType eq StatusChange
  ```

  Valid values are the `EventType` enumerated string values that the Redfish Event Schema defines.

- `MessageId`

  The Service sends events with the matching `MessageId`.

  Example:

```
  https://sseuri?$filter=MessageId eq 'Contoso.1.0.TempAssert'
```

- `MetricReportDefinition`

  The Service sends metric reports generated from the `MetricReportDefinition`.

  Example:

```
  https://sseuri?$filter=MetricReportDefinition eq '/redfish/v1/
TelemetryService/MetricReportDefinitions/PowerMetrics'
```

- `OriginResource`

  The Service sends events for the Resource.

  Example:

```
  https://sseuri?$filter=OriginResource eq '/redfish/v1/Chassis/1/Thermal'
```

- `RegistryPrefix`

  The Service sends events with messages that are part of the `RegistryPrefix`.

  Example:

```
  https://sseuri?$filter=(RegistryPrefix eq Resource) or (RegistryPrefix eq
Task)
```

- `ResourceType`

  The Service sends events for Resources that match the `ResourceType`.

  Example:

```
  https://sseuri?$filter=(ResourceType eq 'Power') or (ResourceType eq
'Thermal')
```

- `SubordinateResources`

When `SubordinateResources` is `true` and `OriginResource` is specified, the Service sends events for the Resource and its subordinate Resources.

Example:

```
  https://sseuri?$filter=(OriginResource eq '/redfish/v1/Systems/1') and
(SubordinateResources eq true)
```

12.5.1.1. Event message SSE stream

The service shall use the `id` field in the SSE stream to uniquely indicate an event payload. The value of the `id` field shall be the same as the `Id` property in the event payload. The value of the `Id` property in the event payload should be the same as the `EventId` property of the last event record in the `Events` array. The value of the `EventId` property for an event record should be a positive integer value and should be generated in a sequential manner. Because clients may not subscribe for every event, the SSE stream may contain gaps in the sequence of `EventId` values. A service should accept the `Last-Event-ID` header from the client 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 clause.

The example payload below shows a stream containing a single event payload with the `id` field set to `1`, and the `data` field containing a single event record within the `Events` array.

```
id: 1
data:{
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": "1",
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:            ],
```

```
data:            "OriginOfCondition": {
data:                "@odata.id": "/redfish/v1/Systems/1/EthernetInterfaces/1"
data:            },
data:            "Context": "ABCDEFGH"
data:        }
data:    ]
data:}
```

12.5.1.2. 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 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 clause.

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.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:}
```

12.6. Update Service

This clause covers the mechanism for software update using the Update Service.

12.6.1. Software update types

There are two methods for clients to provide software updates through the Update Service:

- Simple updates: This is a "pull" model, where the client indicates a network location from which the Service pulls the update.
- Multipart HTTP push updates: This model allows the client to "push" a software image to the Service using HTTP or HTTPS with a multipart formatted request body.

12.6.1.1. Simple updates

A Service may support the `SimpleUpdate` action within the Update Service Resource. A client can perform a POST to the action target URI to initiate a pull-based update, as define by the `UpdateService` schema. After a successful POST, the Service should return the HTTP 202 Accepted status code with the `Location` header set to the URI of a Task Monitor. This Task can be used by clients to monitor the progress and results of the update, which includes the progress of image transfer to the Service.

12.6.1.2. Multipart HTTP push updates

A Service may support the `MultipartHttpPushUri` property within the Update Service Resource. A client can perform an HTTP or HTTPS POST on the URI specified by this property to initiate a push-based update.

- Access to this URI shall require the same privilege as access to the Update Service.

- A client POST to this URI shall contain the `Content-Type` HTTP header with the value `multipart/form-data`, with the body formatted as defined by this specification. For more information on `multipart/form-data` HTTP requests, refer to RFC7578.
- The client POST request shall contain the binary image as one of the parts in a `multipart/form-data` request body, as defined by the table below. In addition, the request shall include parameters for the update in a JSON formatted part in the same `multipart/form-data` request body, as defined by the table below. If the request has no parameters, an empty JSON object shall be used.
- A Service may require the `Content-Length` HTTP header for POST requests to this URI. In this case, if a client does not include the required `Content-Length` header in the POST request, the Service shall return the HTTP 411 Length Required status code.
- A Service should return HTTP 412 Precondition Failed status code if the size of the binary image is larger than the maximum image size that the Service supports, as advertised in `MaxImageSizeBytes` property in the Update Service Resource.
- After a successful POST to this URI, the Service shall return the HTTP 202 Accepted status code with a `Location` header set to the URI of a Task Monitor. This Task can be used by clients to monitor the progress and results of the update.

The following table describes the requirements of a `multipart/form-data` request body for HTTP push software update:

Request body part	HTTP headers	Header value and parameters	Required	Description
Update parameters JSON part	`Content-Disposition`	`form-data; name="UpdateParameters"`	Yes	JSON-formatted part for passing the update parameters. The value of the `name` field shall be `"UpdateParameters"`. The format of the JSON shall follow the definition of the `UpdateParameters` object in the `UpdateService` schema.
	`Content-Type`	`application/json;charset=utf-8` or `application/json`	Yes	The media type format and character set of this request part.
Update file binary part	`Content-Disposition`	`form-data; name="UpdateFile";`	Yes	Binary file to use for this software update. The

Request body part	HTTP headers	Header value and parameters	Required	Description
		`filename=string`		value of the `name` field shall be `"UpdateFile"`. The value of the `filename` field should reflect the name of the file as loaded by the client.
	`Content-Type`	`application/octet-stream`	Yes	The media type format of the binary update file.
OEM specific parts	`Content-Disposition`	`form-data; name="OemXXXX"`	No	Optional OEM part. The value of the `name` field shall start with `"Oem`. `Content-Type` is optional, and depends on the OEM part type.

This is an example of a `multipart/form-data` request to push an update image:

```
POST /redfish/v1/UpdateService/upload HTTP/1.1
Host: <host-path>
Content-Type: multipart/form-data; boundary=----------------------------d74496d66958873e
Content-Length: <computed-length>
Connection: keep-alive
X-Auth-Token: <session-auth-token>

----------------------------d74496d66958873e
Content-Disposition: form-data; name="UpdateParameters"
Content-Type: application/json

{
    "Targets": [
        "/redfish/v1/Managers/1"
    ],
    "@Redfish.OperationApplyTime": "OnReset",
    "Oem": {}
}

----------------------------d74496d66958873e
Content-Disposition: form-data; name="UpdateFile"; filename="flash.bin"
```

```
Content-Type: application/octet-stream

<software image binary>
```

13. Security details

13.1. Protocols

13.1.1. 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.

13.1.2. 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
```

13.1.3. 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.

13.2. Operations involving sensitive data

Operations containing sensitive data should only be allowed using HTTPS. For example, a SimpleUpdate action with a user name and password should use HTTPS to ensure the sensitive data is protected.

13.3. 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.

13.3.1. 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.

13.3.1.1. 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.

13.3.2. Extended error handling

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

13.3.3. 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.

13.3.3.1. 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.

13.3.3.2. Request/Message level authentication

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

13.3.4. Session management

13.3.4.1. 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.**

13.3.4.2. 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"
        }
    },
    ...
}
```

13.3.4.3. 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 an 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.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 log out 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.

13.3.4.4. 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.

13.3.4.5. 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.

13.3.4.6. 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.

13.3.5. 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.

13.3.6. Password management

A Redfish Service provides local user accounts via a collection of `ManagerAccount` resources located under the `AccountService`. The `ManagerAccount` resources allow users to manage their own account information, and for administrators to create, delete, and manage other user accounts.

When account properties are changed, the Service may close open Sessions for this account and require re-authentication.

13.3.6.1. Password change required handling

The Service may require that passwords assigned by the manufacturer be changed by the end user prior to accessing the Service. In addition, administrators may require users to change their account's password upon first access.

The `ManagerAccount` resource contains a `PasswordChangeRequired` boolean property to enable this functionality. Resources that have the property set to `True` shall require the user to change the write-only `Password` property in that resource before access is granted. Manufacturers including user credentials for the Service may use this method to force a change to those credentials before access is granted.

When a client accesses the Service using credentials from a `ManagerAccount` resource that has a `PasswordChangeRequired` value of `True`, the Service shall:

- Allow a Session login and include a `@Message.ExtendedInfo` object in the response containing the `PasswordChangeRequired` message from the Base Message Registry. This indicates to the client that their session is restricted to performing only the password change

operation before access is granted.
- Allow a GET operation on the `ManagerAccount` resource associated with the account.
- Allow a PATCH operation on the `ManagerAccount` resource associated with the account to update the `Password` property. If the value of `Password` is changed, the service shall also set the `PasswordChangeRequired` property to `False`.
- For all other operations, the Service shall respond with status code 403 and include a `@Message.ExtendedInfo` object containing the `PasswordChangeRequired` message from the Base Message Registry.

13.3.7. 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.

13.3.8. Event subscriptions

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

13.3.9. 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 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, an activity that requires privileged access to a read data item represented by an ETag requires the same privileged access to read the ETag.

13.3.10. 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.

13.3.10.1. 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 operations-to-privilege 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.

13.3.10.2. 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 to promote interoperability for Redfish clients.

13.3.10.3. 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 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 a 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" ]
            }
        ]
    }
}
```

13.3.10.4. 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 an operation than those defined for the entity schema. The overrides are defined in the context of the operation-to-privilege mapping for an entity.

13.3.10.5. 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" ]
                    }
                ]
            }
        }
    ]
}
```

13.3.10.6. 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" ]
                    }
                ]
            }
        }
    ]
}
```

13.3.10.7. ResourceURI Override syntax

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 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" ]
                    }
                ]
            }
        }
    ]
}
```

13.3.10.8. 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" ]
        }
    ]
}
```

14. 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.

15. 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.

A service that supports Composability 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, the different composition request types allowed, and how the POST request for the collection is formatted, as well as what properties are required.

15.1. 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
- Constrained Composition
- Expandable Resources

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

15.1.1. 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 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.

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" }
        ]
    }
}
```

15.1.2. Constrained Composition

A Constrained Composition is when a client has identified a set of criteria (or constraints) in which to build a logical entity. This includes criteria such as quantities of components, or characteristics of components. A service that supports Constrained Compositions shall support a POST request that contains the set of characteristics to apply to the composed resource. The specific format of the request is defined by the schema for the resource being composed. This type of request may include expanded elements of resources subordinate to the composed resource.

Example Constrained 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",
    "PowerState": "On",
    "BiosVersion": "P79 v1.00 (09/20/2013)",
    "Processors": {
        "Members": [
            {
                "@Redfish.RequestedCount": 4,
                "@Redfish.AllowOverprovisioning": true,
                "ProcessorType": "CPU",
```

```
                "ProcessorArchitecture": "x86",
                "InstructionSet": "x86-64",
                "MaxSpeedMHz": 3700,
                "TotalCores": 8,
                "TotalThreads": 16
            }
        ]
    },
    "Memory": {
        "Members": [
            {
                "@Redfish.RequestedCount": 4,
                "CapacityMiB": 8192,
                "MemoryType": "DRAM",
                "MemoryDeviceType": "DDR4"
            }
        ]
    },
    "SimpleStorage": {
        "Members" : [
            {
                "@Redfish.RequestedCount": 6,
                "Devices": [
                    {
                        "CapacityBytes": 322122547200
                    }
                ]
            }
        ]
    },
    "EthernetInterfaces": {
        "Members": [
            {
                "@Redfish.RequestedCount": 1,
                "SpeedMbps": 1000,
                "FullDuplex": true,
                "NameServers": [
                    "names.redfishspecification.org"
                ],
                "IPv4Addresses": [
                    {
                        "SubnetMask": "255.255.252.0",
                        "AddressOrigin": "Dynamic",
                        "Gateway": "192.168.0.1"
                    }
                ]
            }
```

```
            ]
        }
    }
```

15.1.3. Expandable Resources

An Expandable Resource is when a service has a baseline composition that cannot be removed. Instead of a client making requests to create a new composed resource, a client is only allowed to add or remove resources from the composed resource. A service that supports Expandable Resources shall support one or more of the update methods listed in the Updating a Composed Resource clause.

15.2. Updating a Composed Resource

A service that supports updating a composed resource shall provide one or more of the following methods for updating composed resources:

- The PUT and/or PATCH methods on the composed resource with a modified list of ResourceBlocks.
- Actions on the composed resource for adding and removing ResourceBlocks.
    - If the actions for adding and removing ResourceBlocks are present in the resource, clients should use this method before attempting PUT/PATCH.

16. ANNEX A (informative)

16.1. Change log

Version	Date	Description
1.8.0	2019-08-08	Added clause for using `/redfish/v1/openapi.yaml` as the well-known URI for the OpenAPI document.
		Added clause that specifies non-Resource reference properties with `Uri` in the name are accessed using Redfish protocol semantics.
		Added `SubordinateResources` $filter parameter for SSE.
		Added Update Service clause for describing requirements for the `SimpleUpdate` action and the `MultipartHttpPushUri` property.
1.7.1	2019-08-08	Added missing statements about the "Owning Entity" annotation term

Version	Date	Description
		and its usage in schema modifications.
		Clarified SSE `id` from `Id` in an event payload and `EventId` within an event record.
		Fixed recommended sequencing of the SSE `id` to be related to `EventId` within an event record.
		Clarified that services are allowed to close Sessions for an account when its password has changed.
		Fixed "Password Management" clause so that a user is allowed to GET their respective account resource when a password change is required.
		Clarified that registries are not required to return "@odata.id".
		Clarified that services should use HTTP 400 for invalid query requests.
		Clarified that services should use HTTP 400 when the `only` query is being combined with other query parameters.
		Clarified that services should ust HTTP 400 when query parameters are used on non-GET operations.
		Added missing clause for how enumeration values are constructed.
		Clarified references to specific messages to also reference their Message Registry.
		Added missing language about the construction of action names in payloads.
		Added informative text for how OEM actions can be defined.
		Added guidance for using HTTPS whenever sensitive data is being transmitted.
		Added clause restricting the maximum size of an Event payload to be 1MiB.
		Clarified that auto expanded Resource Collections can use paging.
		Clarified error response format for SSE.
		Clarified that `charset=utf-8` is not required within the `Content-Type` header for SSE.

Version	Date	Description
		Added missing clause for how URI patterns are constructed.
		Added Excerpt term.
1.7.0	2019-05-16	The specification has been significantly rewritten for clarity. Except for the additions listed below, no normative changes were made. Any clarifications that inadvertently altered the normative behavior are considered errata, and will be corrected in future revisions to the specification.
		Added missing normative statements regarding handling of array properties and PATCH operations on arrays.
		Separated data model and schema clauses.
		Added missing clauses that describe how JSON Schema and OpenAPI files are formatted.
		Added missing clause about the schema versioning methodology.
		Added missing clause about how URI patterns are constructed based on the Resource Tree and property hierarchy.
		Added Dictionary file naming rules and repository locations.
		Enhanced localization definitions and defined repository locations.
		Added missing statement about SSE within the "Eventing mechanism" clause.
		Added Constrained Composition and Expandable Resources clauses to Redfish Composability.
		Added clause about requiring Event Subscriptions to be persistent across service restarts.
		Added clause about persistence of Tasks generated as a result of using "@Redfish.OperationApplyTime" across service restarts.
		Added clause about using "@Redfish.OperationApplyTime" and "@Redfish.MaintenanceWindow" within Task responses.
		Removed "@odata.context" property from example payloads.
		Added "Password Management" clause to describe functional behavior for restricting access when an account requires a password change.

Version	Date	Description
		Added clause around the usage of HTTP status code 403 when an account requires a password change.
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 '$filter' examples to use '/' instead of '.' for property paths.
		Clarified the usage of Messages in a successful Action response; provided an example.
		Added clarification about services supporting a subset of HTTP operations on resources specified in schema.
		Added clarification about services implementing writable properties as read only.
		Added clarification about session termination not affecting connections opened by the session.
		Added "Redfish Provider" term definition.
		Updated JSON Schema references to point to Draft 7 of the JSON Schema specification.
		Added clarifications about scenarios for when a request to add an Event Subscription contains conflicting information and how services respond.
		Removed language about ignoring the 'Links' property in PATCH requests.
		Clarified usage of ETags to show that a client is not supposed to PATCH '@odata.etag' when attempting to use ETag protection for a resource.
		Clarified usage of the 'only' query parameter to show it's not to be combined with '$expand' and not to be used with singular resources.
		Clarified the usage of HTTP status codes with Task Monitors.
		Various spelling and grammar fixes.

Version	Date	Description
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 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. 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.

Version	Date	Description
		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 reference 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.
		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.

Version	Date	Description
		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 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

Version	Date	Description
		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.
		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.

Version	Date	Description
		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.
		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

Version	Date	Description
		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.
		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.

Version	Date	Description
		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 matches 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 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

Version	Date	Description
		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 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, 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.

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