Document Identifier: DSP0266

Date: 2021-04-08

Version: 1.13.0

Redfish Specification

Supersedes: 1.12.1

Document Class: Normative

Document Status: Published

Document Language: en-US

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

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patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a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

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

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

CONTENTS

1 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 DMTF.

This version supersedes version 1.12.1. For a list of the changes, see ANNEX A (informative) Change log.

1.1 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
- David Black - Dell Inc.
- Jeff Bobzin - Insyde Software Corp.
- Patrick Boyd - Dell Inc.
- David Brockhaus - Vertiv
- Richard Brunner - VMware Inc.
- Sean Byland - Cray Inc.
- Lee Calcote - Seagate Technology
- Keith Campbell - Lenovo
- 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 - Arm Limited, 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
- Magnus Lundmark - Ericsson AB
- Steve Lyle - Hewlett Packard Enterprise
- Gunnar Mills - IBM
- 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.
- Joseph Reynolds - IBM
- Irina Salvan - Microsoft Corporation
- Bill Scherer - Hewlett Packard Enterprise
- Hemal Shah - Broadcom Inc.
- Jim Shelton - Vertiv
- Tom Slaight - Intel Corporation
- Josiah Smith - Eaton
- Donnie Sturgeon - Vertiv
- Pawel Szymanski - Intel Corporation
- Paul Vancil - Dell Inc.
- Joseph White - Dell Inc.
- Linda Wu - NVIDIA Corporation, Super Micro Computer, Inc.

2 Introduction {-}

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.

This document defines the RESTful interface protocol and the various concepts and services necessary to implement a Redfish interface. The definition of the schema based data model and standard messages for the Redfish interface are covered separately in the following documents:

- DMTF DSP8010, *Redfish Schema Bundle*, https://www.dmtf.org/dsp/DSP8010 contains the individual schema definition files in multiple schema description languages.
- DMTF DSP0268, *Redfish Schema Supplement*, https://www.dmtf.org/dsp/DSP0268 contains the normative descriptions and example payloads for all standard Redfish schema in a single reference guide.
- DMTF DSP8011, *Redfish Standard Registries Bundle*, https://www.dmtf.org/dsp/DSP8011 contains the message registries used for error reporting and event messages.

3 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 specification defines the required elements for all Redfish implementations, and the optional elements that system vendors and manufacturers can choose. This specification also defines at which points an implementation can provide OEM-specific extensions.

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 how a client can successfully and effectively access and use a Redfish service.

The specification does not require that implementations of the Redfish interfaces and functions require particular hardware or firmware.

4 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- DMTF DSP0270, *Redfish Host Interface Specification*, https://www.dmtf.org/sites/default/files/standards/documents/DSP0270_1.0.0.pdf
- Redfish Schema: RedfishExtensions v1.0.0, https://redfish.dmtf.org/schemas/v1/RedfishExtensions_v1.xml
- *Transport Layer Security (TLS) Parameters*, https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml
- *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
- IETF RFC1738, T. Berners-Lee et al, *Uniform Resource Locators (URL)*, https://tools.ietf.org/html/rfc1738
- IETF RFC3986, T. Berners-Lee et al, *Uniform Resource Identifier (URI): Generic Syntax*, https://tools.ietf.org/html/rfc3986
- IETF RFC5280, D. Cooper et al, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*,
- IETF RFC6585, M. Nottingham et al, *Additional HTTP Status Codes*, https://tools.ietf.org/html/rfc6585
- IETF RFC6901, P. Bryan, Ed. et al, *JavaScript Object Notation (JSON) Pointer*, https://tools.ietf.org/html/rfc6901
- IETF RFC7230, R. Fielding et al, *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, https://tools.ietf.org/html/rfc7230
- IETF RFC7231, R. Fielding et al, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, https://tools.ietf.org/html/rfc7231
- IETF RFC7232, R. Fielding et al, *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*, https://tools.ietf.org/html/rfc7232
- IETF RFC7234, R. Fielding et al, *Hypertext Transfer Protocol (HTTP/1.1): Caching*, https://tools.ietf.org/html/rfc7234
- IETF RFC7525, Y. Sheffer et al, *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, https://tools.ietf.org/html/rfc7525
- IETF RFC7578, L. Masinter et al, *Returning Values from Forms: multipart/form-data*, https://tools.ietf.org/html/rfc7578
- IETF RFC7617, J. Reschke et al, *The 'Basic' HTTP Authentication Scheme*, https://tools.ietf.org/html/rfc7617
- IETF RFC8259, T. Bray, Ed., *The JavaScript Object Notation (JSON) Data Interchange Format*, https://tools.ietf.org/html/rfc7617
- IETF RFC8288, M. Nottingham, *Web Linking*, https://tools.ietf.org/html/rfc8288

- ISO 639-1:2002, *Codes for the representation of names of languages - Part 1: Alpha-2 code*, https://www.iso.org/standard/22109.html
- 24 February 2014, *OData Version 4.0 Part 1: Protocol*, https://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html
- 24 February 2014, *OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*, https://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html
- 10 March 2016, *OData Version 4.0 Plus Errata 03 OASIS Standard incorporating Draft 01 of Errata 03*, https://docs.oasis-open.org/odata/odata/v4.0/errata03/csd01/complete/vocabularies/Org.OData.Measures.V1.xml
- 20 November 2014, SNIA *TLS Specification for Storage Systems*, https://www.snia.org/tech_activities/standards/curr_standards/tls
- *The OpenAPI Specification*, https://swagger.io/specification/
- *The Unified Code for Units of Measure*, https://ucum.org/ucum.html
- 24 December 2020, *Fetch Living Standard*, https://fetch.spec.whatwg.org/
- *9.2 Server-sent events* in the *HTML Living Standard*, https://html.spec.whatwg.org/multipage/server-sent-events.html

5 Terms, definitions, symbols, and abbreviated terms

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.

The term "deprecated" in this document is to be interpreted as material that is not recommended for use in new development efforts. Existing and new implementations may use this material, but they should move to the favored approach. Deprecated material may be implemented in order to achieve backwards compatibility. Deprecated material should contain references to the last published version that included the deprecated material as normative material and to a description of the favored approach. Deprecated material may be removed from the next major version of the specification.

This document defines these additional terms:

5.1 Hardware terms

5.1.1 baseboard management controller (BMC)

embedded device or service

Note 1 to entry: 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.

5.1.2 IPMI

Intelligent Platform Management Interface

5.1.3 KVM-IP

keyboard, video, mouse redirection over IP

5.1.4 NIC

network interface controller

5.1.5 PCI

Peripheral Component Interconnect

5.1.6 PCIe

Peripheral Component Interconnect Express

5.2 Web development terms

5.2.1 CORS

cross-origin resource sharing

5.2.2 CRUD

basic **C**reate, **R**ead, **U**pdate, and **D**elete operations that any interface can support

5.2.3 CSRF

cross-site request forgery

5.2.4 event

data structure that corresponds to one or more alerts

5.2.5 excerpt

subset of data that is copied from one resource and presented in another resource

Note 1 to entry: An excerpt provides data in convenient locations without duplication of entire resources.

5.2.6 HTTP

Hypertext Transfer Protocol

5.2.7 HTTPS

Hypertext Transfer Protocol Secure

Note 1 to entry: TLS secures HTTP.

5.2.8 hypermedia API

API that enables you to navigate through URIs that a service returns

5.2.9 IP

Internet Protocol

5.2.10 JSON

JavaScript Object Notation

5.2.11 member

single resource instance in a resource collection

5.2.12 message

complete HTTP-formatted or HTTPS-formatted request or response

Note 1 to entry: In the REST-based Redfish protocol, every request results in a response.

5.2.13 OData

Open Data Protocol (OData), as defined in OData Version 4.0 Part 1: Protocol

5.2.14 OData service document

resource that provides information about the service root for generic OData clients

5.2.15 operation

HTTP `POST`, `GET`, `PUT`, `PATCH`, `HEAD`, and `DELETE` request methods that map to generic CRUD operations

5.2.16 parent resource

parent to another resource if the initial segment of the resource URI is the same as the URI of the other resource, but is at least one level higher

Note 1 to entry: For example, `/redfish/v1/Chassis/A88` is a parent resource of `/redfish/v1/Chassis/A88/Assembly`.

5.2.17 property

name-value pair in a Redfish-defined request or response

Note 1 to entry: A property can be any valid JSON data type.

5.2.18 request

message from a client to a service

5.2.19 response

message from a service to a client in response to a request message

5.2.20 subscription

registration of a destination to receive events

5.2.21 task

representation of a long-running operation

5.2.22 task monitor

opaque service-generated URI that the client who initiates the request can use to monitor an asynchronous operation

5.2.23 TCP

Transmission Control Protocol

5.2.24 TLS

Transport Layer Security

5.2.25 XSS

cross-site scripting

5.3 Redfish terms

5.3.1 collection

see resource collection

5.3.2 Redfish client

communicates with a Redfish service and accesses one or more of the service's resources or functions

5.3.3 Redfish protocol

discovers, connects to, and inter-communicates with a Redfish service

5.3.4 Redfish schema

a set of human and machine-readable documents that define Redfish resources using one or more of the supported schema definition languages

5.3.5 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

Note 1 to entry: Also known as the *service*.

5.3.6 resource

URI-addressable Redfish data structure

5.3.7 resource collection

set of similar resources where the number of instances can shrink or grow

5.3.8 resource tree

tree structure of resources accessible through a well-known starting URI

Note 1 to entry: A client can discover the available resources on a Redfish service by following the resource hyperlinks from the base of the tree.

5.3.9 resource type

set of definitions for properties and actions contained within a resource and documented in the Redfish schema files

5.3.10 service root

starting-point resource for locating and accessing the other resources and associated metadata that make up an instance of a Redfish service

5.3.11 subordinate resource

is subordinate to another resource if the initial segment of the resource URI is the same as the URI of the other resource, but is at least one level deeper

Note 1 to entry: For example, `/redfish/v1/Chassis/A88/Assembly` is a subordinate resource of the `Chassis` resource named `A88`.

6 Typographical conventions

The following typographical convention indicates deprecated material:

DEPRECATED

Deprecated material appears here.

END DEPRECATED

In places where this typographical convention cannot be used, such as tables or figures, the "DEPRECATED" label is used alone.

7 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 Redfish 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.

An interoperable Redfish schema defines this model, which is freely available and published in OpenAPI YAML, OData CSDL, and JSON Schema formats.

7.1 Goals

As an architecture, data model, and set of protocols that enable a client to access Redfish services, Redfish has these goals.

Table 1 describes these goals:

Goal	Purpose
Scalable	Can scale on stand-alone machines or racks of equipment.
Flexible	Can implement through existing hardware or entirely as a software service.
Extensible	Can easily add new and vendor-specific capabilities to the data model.
Backward-compatible	Can add capabilities while preserving investments in earlier implementations.
Interoperable	Provides consistent functionality across multiple vendor implementations.
Standards-based	Built on ubiquitous and secure protocols. Leverages other standards where applicable.
Simple	Easy-to-use without the need for highly specialized programming skills or systems knowledge.
Lightweight	Designed to reduce complexity and implementation costs. Minimizes the required footprint for implementations.

: **Table 1 — Redfish goals**

7.2 Design tenets

To deliver these goals, Redfish:

- Provides a RESTful interface by using a JSON payload and a data model.
- Separates the protocol from the data model, which enables the independent revision and use of each.
- Specifies versioning rules for protocols and schema.
- Leverages strength of ubiquitous standards where it meets architectural requirements, such as JSON, HTTP, OData, OpenAPI, and the RFCs that this document references.
- Organizes the data model so that it provides clearly demarcated and value-add features in the same payload as standardized items.
- Makes data in payloads as obvious in context as possible.
- Maintains implementation flexibility. Does not tie the interface to any particular underlying implementation or architecture.
- Focuses on widely used capabilities. To avoid complexity, does not add functions that only a small percentage of users value.

7.3 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 to manage new system or component types, or update the data model.

Interoperable does not mean identical. Many elements of Redfish are optional. Clients should be prepared to discover the optional elements by 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.

7.4 Additional design background and rationale

7.4.1 REST-based interface

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.

7.4.2 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.

7.4.3 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.

7.4.4 Hypermedia API service root

Redfish has a single service root URI and clients can discover all resources through referenced URIs. The *hypermedia API* enables the discovery of resources through hyperlinks.

7.4.5 OpenAPI v3.0 support

The OpenAPI v3.0 provides a rich ecosystem of tools for using RESTful interfaces that meet the design requirements of that specification. Starting with *Redfish Specification v1.6.0*, 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.

For details, see *OpenAPI Specification v3.0*.

7.4.6 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 follows OData conventions for describing schema, URL conventions, and definitions for typical properties in a JSON payload.

7.5 Service elements

7.5.1 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.

7.5.2 Eventing mechanism

Redfish enables clients to receive messages outside the normal request and response paradigm. 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.

This specification defines two styles of eventing:

- Push-style eventing.

  When the service detects the need to send an event, it calls 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.

- Server-sent events (SSE)-style eventing.

The client opens an SSE connection to the service through a `GET` on the `ServerSentEventUri`-specified URI in the event service.

For information, see the Eventing clause.

7.5.3 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.

7.5.4 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.

7.5.5 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, or 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 that enable client discovery of these capabilities and access mechanisms to enable interoperability.

7.6 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.

8 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 transports 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, which provides network and interaction efficiency for both user interface (UI) and automation usage. Specifically, the protocol can leverage existing tool chains.

Table 2 describes the items that the Redfish protocol uses:

Item	Description
HTTP methods	Maps to common CRUD operations.
Actions	Expands operations beyond CRUD-type operations.
Media types	Negotiates the type of data sent in the message body.
HTTP status codes	Indicates the success or failure of the server's request.
Error responses	Returns more information than HTTP status codes.
TLS	Secures messages. See Security details.
Asynchronous semantics	Manages long-running operations.

: **Table 2 — Redfish protocol**

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

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

8.1 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.

A `GET` operation on a URI returns a representation of the resource with properties and hyperlinks to associated resources. The service root URI is well known and is based on the protocol version.

To discover the URIs to additional resources, extract the associated resource hyperlinks from earlier responses. The *hypermedia API* enables the discovery of resources through hyperlinks.

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

For example, this URI:

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

Contains these component parts:

- `https:` is the scheme.
- `mgmt.vendor.com` is the authority to which to delegate the URI.
- `redfish/v1` is the service root and version.
- `Systems/1` is the unique resource path.

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 root 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.

URIs that conform to RFC3986 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 references another property within a resource, use the RFC6901-defined URI fragment identifier representation format. If the property is 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.

8.2 HTTP methods

Table 3 describes the mapping of HTTP methods to the Redfish-supported operations. If the **Required** column contains **Yes**, a Redfish interface shall support the HTTP method. If the **Required** column contains **No**, a Redfish interface may support the HTTP method.

HTTP method	Interface semantic	Required
`POST`	Create resource Resource action Eventing	Yes
`GET`	Retrieve resource	Yes
`PUT`	Replace resource	No
`PATCH`	Update resource	Yes
`DELETE`	Delete resource	Yes
`HEAD`	Retrieve resource header	No
`OPTIONS`	Retrieve header Cross-origin resource sharing (CORS) preflight	No

: **Table 3 — Mapping of HTTP methods to Redfish-supported operations**

For HTTP methods that the Redfish service does not support or that Table 3 omits, the Redfish service shall return the HTTP `405 Method Not Allowed` status code or the HTTP `501 Not Implemented` status code.

8.3 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.

The service for the redirected resource shall enforce the authentication and authorization requirements for the redirected resource.

8.4 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 return 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 a JSON object, as specified in RFC8259. 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 in the response body, clients specify an `Accept-Encoding` request header.

8.5 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.

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, it may respond with the HTTP `428 Precondition Required` 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>
```

8.6 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

- `<MajorVersion>` is an integer that represents the major version. Indicates a backward-incompatible change.
- `<MinorVersion>` is an integer that represents the minor version. Indicates a minor update. Redfish introduces functionality but does not remove any functionality. The minor version preserves compatibility with earlier minor versions.
- `<ErrataVersion>` is an integer that represents the errata version. Indicates a fix to the earlier version.

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

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

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

8.7 Redfish-defined URIs and relative reference rules

Table 4 describes the Redfish-defined URIs that a Redfish service shall support:

URI	Returns	Note
`/redfish`	Version. A major update that does not preserve compatibility with earlier minor versions.	Services shall support this URI.
`/redfish/v1/`	Redfish service root.	Services shall support this URI.
`/redfish/v1/odata`	Redfish OData service document.	Services shall support this URI.
`/redfish/v1/$metadata`	Redfish metadata document.	Services shall support this URI.
`/redfish/v1/openapi.yaml`	Redfish OpenAPI YAML document.	Services should support this URI.

: **Table 4 — Redfish-defined URIs**

In addition, Table 5 describes the URIs that services shall process 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/`
`/redfish/`	`/redfish`

: **Table 5 — Redfish-defined URIs without trailing slashes**

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-defined URIs and URIs starting with `/redfish` are reserved for future standardization by DMTF and DMTF alliance partners, except OEM extension URIs, which shall conform to the requirements of the OEM URIs clause.

All relative references 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`.

For details, see RFC3986.

9 Service requests

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

9.1 Request headers

The *HTTP Specification* defines headers for request messages. Table 6 defines those headers and their requirements for Redfish services and clients.

For Redfish services:

- Redfish services shall process the headers in Table 6 as defined by the *HTTP 1.1 Specification* if the **Service requirement** column contains **Yes** or **Conditional**.
- Redfish services should process the headers in Table 6 and Table 7 as defined by the *HTTP 1.1 Specification* if the **Service requirement** column contains **No**.

For Redfish clients (sending the HTTP requests):

- Redfish clients shall include the headers in Table 6 as defined by the *HTTP 1.1 Specification* if the **Client requirement** column contains **Yes** or **Conditional**.
- Redfish clients should transmit the headers in Table 6 and Table 7 as defined by the *HTTP 1.1 Specification* if the **Client requirement** column contains **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`. Services shall support XML metadata requests with `Accept` header values of `application/xml` or `application/xml;charset=utf-8`. Services shall support OpenAPI YAML schema requests with `Accept` header values of `application/yaml` or `application/yaml;charset=utf-8` or `application/vnd.oai.openapi` or `application/vnd.oai.openapi;charset=utf-8`. Services shall support SSE requests with `Accept` header values of `text/event-stream` or `text/event-stream;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 return an acceptable response to a request with this header, it shall respond with the HTTP `406 Not Acceptable` status code. If the request omits this header, the service should 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	RFC7617	Required for HTTP basic authentication. A client can access unsecured resources without this header on 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 needs to use `Content-Length` and does not support `Transfer-Encoding`, it responds with the HTTP `406 Not Acceptable` 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.

Header	Service requirement	Client requirement	Supported values	Description
`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 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 Not Modified` 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. 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 Precondition Failed` status code.
`Origin`	Yes	No	Fetch Living Standard, 3.1. `Origin` header	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.

: **Table 6 — Request headers**

Redfish services shall understand and be able to process the headers in Table 7 as defined by this specification if the **Service requirement** column contains **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, a client can access unsecured resources without establishing a session.

: **Table 7 — Request headers part 2**

9.2 GET (read requests)

9.2.1 GET (read requests) overview

The `GET` operation retrieves 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 requirements of the media types. 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.

If supported by the service, clients can perform a conditional `GET` operation by specifying an `If-None-Match` request header that contains the ETag of the resource.

9.2.2 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. These calls can result in missed or duplicate elements if multiple `GET` requests use paging to retrieve the `Members` array instances.

- Clients shall not make assumptions about the URIs for the members of a resource collection.
- Retrieved resource collections shall always include the count (`Members@odata.count`) property to specify the total number of entries in its `Members` array.
- Regardless of the next link (`Members@odata.nextLink`) property or paging, the count (`Members@odata.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 might 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 can contain a subset of the full resource collection's members.
- Individual members shall not be split across response bodies.
- A next link (`Members@odata.nextLink`) property annotation shall be supplied in the response body with the URI to the next set of members in the collection.
- The next link (`Members@odata.nextLink`) property shall adhere to the rules in the Next link property clause.
- `GET` operations on the next link (`Members@odata.nextLink`) property shall return the subsequent section of the resource collection response.

9.2.3 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.

9.2.4 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.

9.3 Query parameters

9.3.1 Query parameter overview

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 `$select` query parameters.
- Shall include the `ProtocolFeaturesSupported` object in the service root, if the service supports query parameters.
  - This object indicates which parameters and options have been implemented.
- Shall ignore unknown or unsupported query parameters that do not begin with `$`.
- 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, such as `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 on operations other than `GET`.

Services shall process query parameters in this order:

- `$filter`
- `$skip`
- `$top`

- Apply server-side pagination
- `$expand`
- `excerpt`
- `$select`

Table 8 describes the query parameters:

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:** `https://resource?excerpt`

`$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 The $expand query parameter.
	
	Example:
	
	`https://resource?$expand=*($levels=3)`
	
	`https://resourcecollection?$expand=.($levels=1)`

`$filter=<string>`	Applies to resource collections. Returns a subset of collection members that match the `$filter` expression. See The $filter query parameter. **Example:** `https://resourcecollection?$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 resource collection, 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:** `https://resourcecollection?only`

Query parameter	Description and example
`$select=<string>`	Returns a subset of the resource's properties that match the `$select` expression. See The $select query parameter. **Example:** `https://resource?$select=SystemType,Status`
`$skip=<integer>`	Applies to resource collections. Returns a subset of the members in a resource collection, or an empty set of members if the `$skip` value is greater than or equal to the member count. This paging query parameter defines the number of members in the resource collection to skip. **Example:** `https://resourcecollection?$skip=5`
`$top=<integer>`	Applies to resource collections. Defines the number of members to show in the response. Minimum value is `0`, though a value of `0` returns an empty set of members. **Example:** `https://resourcecollection?$top=30`

: **Table 8 — Query parameters**

9.3.2 The $expand query parameter

The `$expand` query parameter enables 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.

Table 9 describes the Redfish-supported options for the `$expand` query parameter. The service may implement some of these options but not others. Any other supported syntax for `$expand` is outside the scope of this specification.

Option	Description	Example
asterisk (`*`)	Shall expand all hyperlinks, including those in payload annotations, such as `@Redfish.Settings`, `@Redfish.ActionInfo`, and `@Redfish.CollectionCapabilities`.	`https://resource?$expand=*`

Option	Description	Example
`$levels`	Number of levels the service should cascade the `$expand` operation. The default level shall be 1. For example, `$levels=2` expands both the hyperlinks in the current resource (level 1), and the hyperlinks in the resulting expanded resources (level 2).	`https://resourcecollection?$expand=.($levels=2)`
period (`.`)	Shall expand all hyperlinks **not** in any links property instances of the resource, including those in payload annotations, such as `@Redfish.Settings` , `@Redfish.ActionInfo` , and `@Redfish.CollectionCapabilities` .	`https://resourcecollection?$expand=.`
tilde (`~`)	Shall expand all hyperlinks found in all links property instances of the resource.	`https://resourcecollection?$expand=~`

: **Table 9 — The $expand query parameter options**

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 the HTTP `507 Insufficient Storage` status code.

If a service cannot return the payload corresponding to an individual member of a resource collection, it should

return the `@odata.id` property for that member and should return extended information indicating the reason that member was not returned, such as when a provider internal to the service returns an error or times out.

The following example expands the `RoleCollection` resource 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"]
   }]
}
```


9.3.3 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 objects or properties in arrays of objects 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 `$select`, they shall return all requested properties of the referenced resource. If a requested property is an object, the service shall return the entire object. The `@odata.id` and `@odata.type` properties shall be in the response payload and contain the same values as if `$select` was omitted. 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 omitted.

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

9.3.4 The $filter query parameter

The `$filter parameter` enables a client to request a subset of the resource collection's members based on the `$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 Bad Request` 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.

Table 10 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'`

Value	Description	Example
`and`	Logical and operator.	`ProcessorSummary/Count eq 2 and MemorySummary/TotalSystemMemoryGiB gt 64`
`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`

: **Table 10 — The $filter query parameter options**

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 Not Implemented` status code.

9.4 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.

Services shall reject `HEAD` requests that contain query parameters. Services should return the HTTP `400 Bad Request` status code if provided with a query parameter in a `HEAD` request.

9.5 Data modification requests

9.5.1 Data modification requests overview

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.

9.5.2 Modification success responses

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

- The HTTP `201 Created` status code with a body that contains the JSON representation of the newly created resource after the request has been applied.
- The HTTP `202 Accepted` 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 Created` status code.
- The HTTP `204 No Content` 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:

- The HTTP `200 OK` 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.
- The HTTP `202 Accepted` 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 OK` status code.
- The HTTP `204 No Content` status code with an empty payload in the event that service cannot return a representation of the modified or deleted resource.

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

9.5.3 Modification error responses

If the resource exists but does not support the requested operation, services shall return the HTTP `405 Method Not Allowed` 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.

9.6 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` method with an empty JSON object, which indicates that the service should make no changes to the resource.

For resources that allow for properties to not be updated immediately, clients can perform `PATCH` requests to a designated settings resource. For more information, see the Settings resource clause.

See the Modification success responses clause for behavior when the `PATCH` operation is successful.

If supported by the service, clients can perform a conditional `PATCH` operation by specifying an `If-Match` or `If-None-Match` request header that contains the ETag of the resource.

The implementation may reject the update on certain properties based on its own policies and, in this case, not make the requested update.

A partial success of a `PATCH` operation occurs when a modification request for multiple properties results in at least one property updated successfully, but one or more properties could not be updated. In these cases, the service shall return the HTTP `200 OK` status code and a resource representation with extended information that lists the properties that could not be updated. Examples include:

- A property is read-only, unknown, or unsupported.
- A service-side error occured, such as a write failure for an EEPROM.

If all properties in the update request are read-only, unknown, or unsupported, but the resource can be updated, the service shall return the HTTP `400 Bad Request` status code and an error response with messages that show the non-updatable properties.

If the update request only contains OData annotations, such as resource identifier, type, and ETag properties, the service should return the HTTP `400 Bad Request` status code with the `NoOperation` message from the Base Message Registry.

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

9.7 PATCH on array properties

The Array properties clause describes the three styles of array properties in a resource.

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 that use the fixed or variable length style remove those elements, while array properties that use the rigid style 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 in the current array shall remove the remaining elements of the array.

For example, a fixed length-style `Flavors` array indicates that 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"]
}
```

After the `PATCH` operation, the resulting array is:

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

9.8 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.

See the Modification success responses clause for behavior when the `PUT` operation is successful.

If supported by the service, clients can perform a conditional `PUT` operation by specifying an `If-Match` or `If-None-Match` request header that contains the ETag of the resource.

Services may reject requests that do not include properties that the resource definition (schema) requires.

9.9 POST (create)

To create a 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. See the Modification success responses clause for behavior when the `POST` operation is successful.

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 member to the resource collection at `/redfish/v1/EventService/Subscriptions`, it can perform a `POST` request to either `/redfish/v1/EventService/Subscriptions` or `/redfish/v1/EventService/Subscriptions/Members`.
- The `POST` operation shall not be idempotent.
- Services may allow the inclusion of `@Redfish.OperationApplyTime` property in the request body. See Operation apply time.
- Services should return the HTTP `400 Bad Request` status code for requests containing properties with the value `null`.

9.10 DELETE (delete)

To remove a resource, the service shall support the `DELETE` method. Resources subordinate to the resource removed by a `DELETE` method are typically removed, as the contents of subordinate resources are dependent on the parent resource. In some cases, related resources may also be relocated in the resource tree based on their definition and usage. Other resources in the resource tree may also be removed or incur side effects of a resource removal.

See the Modification success responses clause for behavior when the `DELETE` operation is successful.

- If the resource was already deleted, the service may return the HTTP `404 Not Found` status code or a success code.
- The service may allow the inclusion of the `@Redfish.OperationApplyTime` property in the request body. See Operation apply time.

9.11 POST (action)

Services shall support the `POST` method as a way for clients 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

- `<ResourceUri>` is the URI of the resource that supports the action.
- `Actions` is the name of the property that contains the actions for a resource, as defined by this specification.
- `<QualifiedActionName>` is the qualified name of the action. Includes the resource type.

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. For information about the structure of the request and required parameters, see the Actions property clause. Some parameter information may require that the client examine the Redfish schema that corresponds to the resource.

If the action request does not contain all required parameters, the service shall return the HTTP `400 Bad Request` status code. If the action request contains unsupported parameters, the service shall ignore the unsupported parameters or return the HTTP `400 Bad Request` status code. 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.

Table 11 describes the HTTP status codes and additional information that the service shall return a response to a successful `POST` (action) request:

To indicate	HTTP status code	Additional information
Success, and the schema does not contain a response definition.	`200 OK`	An error response, 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 Message Registry in the `code` property in the response body.
Success, and the schema contains a response definition for the action.	`200 OK`	The response body conforms to the action response defined in the schema.
A new resource was created, and the schema does not contain a response definition.	`201 Created`	A `Location` response header set to the URI of the created resource. An error response, 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 Message Registry in the `code` property in the response body.
A new resource was created, and the schema contains a response definition for the action.	`201 Created`	A `Location` response header set to the URI of the created resource. The response body conforms to the action response defined in the schema.
Additional time is required to process.	`202 Accepted`	A `Location` response header set to the URI of a *task monitor*.

To indicate	HTTP status code	Additional information
Success, and the schema does not contain a response definition.	`204 No Content`	No response body.

: **Table 11 — POST (action) status codes**

If an action requested by the client has no effect, such as a reset of a `ComputerSystem` where the `ResetType` parameter is set to `On` and the `ComputerSystem` is already `On`, the service should respond with the HTTP `200 OK` status code and return the `NoOperation` message from the Base Message Registry.

If an error was detected and the action request was not processed, the service shall return an HTTP `4XX` or HTTP `5XX` status code. The response body, if provided, shall contain an error response that describes the error or errors.

Example successful action response:

```
{
   "error": {
      "code": "Base.1.8.Success",
      "message": "Successfully Completed Request",
      "@Message.ExtendedInfo": [{
         "@odata.type": "#Message.v1_1_1.Message",
         "MessageId": "Base.1.8.Success",
         "Message": "Successfully Completed Request",
         "Severity": "OK",
         "MessageSeverity": "OK",
         "Resolution": "None"
      }]
   }
}
```

9.12 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 operations on a resource collection and delete operations on members of 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 requests to create new members and delete existing members:

```
{
   "@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 Accepted` status code with a `Location` header set to the URI of a *task monitor*, 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.

9.13 Deep operations

Implementations may support operations that modify the current resource as well as subordinate resources. These operations are known as deep operations. They give the client the ability to modify more than one resource with a single operation.

Table 12 describes the types of deep operations that this specification defines:

Operation	Description	Example
Deep `PATCH` (update)	Modify a resource and one or more subordinate resources.	Modify a `ComputerSystem` resource as well as subordinate `Storage` and `NetworkInterface` resources.
Deep `POST` (create)	Create multiple resources in a resource collection.	Create `ManagerAccount` resources.

: **Table 12 — Deep operations**

- Services that support deep `PATCH` for updating resources shall set the value of the `DeepPATCH` property in the `DeepOperations` property in the `ProtocolFeaturesSupported` property within the service root to `true`.
- Services that support deep `POST` for creating resources shall set the value of the `DeepPOST` property in the `DeepOperations` property in the `ProtocolFeaturesSupported` property within the service root to `true`.
- The `Members` property in resource collections shall not be removed when using a deep `PATCH`.
- Action URIs shall not support deep `POST` operations.
- If the service supports deep operations, the `MaxLevels` property in the `DeepOperations` property in the `ProtocolFeaturesSupported` property in the service root shall indicate the maximum number of levels that the service supports for deep operations.
- To request deep operations on a resource, send the HTTP method to the deep operation URI of the resource. The URI for deep operations on any resource shall be in the format: `<ResourceUri>.Deep`.
- The schema used for validating the root level of the request body shall be the schema of the resource in the resource URI.
  - The subordinate resources included in the request body shall be validated against their corresponding schema.

The body of deep operations contains the resource being modified as well as the subordinate resources being modified. This resource can be a collection or a single instance. These resources could be subordinate resources, subordinate resource collections, or subordinate members of resource collections. The client can omit properties from the request such as those it does not want to modify or that the service controls. Requests that include references to multiple instances, such as members of a collection, shall include the `Members` property as part of the request body.

To determine which members of subordinate resource collections are to be modified by a deep `PATCH`, services shall use the `@odata.id` property provided by the client to identify the member of the resource collection to be modified.

Clients may provide the `@odata.etag` property in subordinate resources being modified by a deep `PATCH`. If the request contains the `If-Match` or `If-None-Match` header, the service shall compare the ETag in the request header with the ETag of the resource specified by the URI. If this check passes, the operation can proceed using the `@odata.etag` values contained in the body of the subordinate resources. In this case, the operation on each subordinate resource shall be completed independently, where some subordinate values that pass the condition check proceed and the resources that fail do not proceed. In this case, annotated extended information shall be included in the subordinate resource representation of the response.

Failure semantics for deep operations are similar to that of other operations of similar type. If any properties in a deep `PATCH` operation succeeded, the result is a `200 OK` with the results returned in the response, and the service should include extended information indicating warnings or errors. For a deep `POST` operation, if any member of the collection was created then a `201 Created` shall be returned, and any members that were not created should have extended information in their place holders with sufficient identifying information, such as returning all of the properties provided in the `POST` request body for that member, as well as extended information indicating why the creation was not successful. When sending a deep `POST` request, the value of the `Location` header shall be that of one of the URIs created and should be that of one of the least subordinate URIs, such as that of a `ComputerSystem` resource and not one of the devices subordinate to the `ComputerSystem` resource.

Deep `POST` shall not be allowed on the `SessionCollection` resource.

The following deep `PATCH` example modifies two members of the `RoleCollection` resource:

```
PATCH /redfish/v1/AccountService/Roles.Deep HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
   "Members": [{
      "@odata.id": "/redfish/v1/AccountService/Roles/OperatorRestricted",
      "AssignedPrivileges": ["Login", "ConfigureComponents"]
   }, {
      "@odata.id": "/redfish/v1/AccountService/Roles/ReadOnlyRestricted",
      "AssignedPrivileges": ["Login"]
   }]
}
```

The following deep `POST` example creates two members in the `RoleCollection` resource:

```
POST /redfish/v1/AccountService/Roles.Deep HTTP/1.1
```

```
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
   "Members": [{
      "RoleId": "OperatorRestricted",
      "AssignedPrivileges": ["Login", "ConfigureComponents"]
   }, {
      "RoleId": "ReadOnlyRestricted",
      "AssignedPrivileges": ["Login"]
   }]
}
```

The following deep `PATCH` example modifies the asset tag and BIOS settings of a `ComputerSystem` resource:

```
PATCH /redfish/v1/Systems/47832.Deep HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
OData-Version: 4.0

{
   "AssetTag": "Inventory Tag 12394783431",
   "Bios": {
      "@odata.id": "/redfish/v1/Systems/47832/Bios",
      "@Redfish.Settings": {
         "SettingsObject": {
            "@odata.id": "/redfish/v1/Systems/47832/Bios/SD",
            "Attributes": {
               "AdminPhone": "(123) 456-789",
               "BootMode": "Uefi"
            }
         }
      }
   }
}
```

The following example shows a deep `PATCH` with ETags in the request:

```
PATCH /redfish/v1/AccountService/Roles.Deep HTTP/1.1
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
If-Match: <Collection ETag>
OData-Version: 4.0

{
```

```
   "Members": [{
      "@odata.id": "/redfish/v1/AccountService/Roles/OperatorRestricted",
      "@odata.etag": "W/\"ABCDEFG\"",
      "AssignedPrivileges": ["Login", "ConfigureComponents"]
   }, {
      "@odata.id": "/redfish/v1/AccountService/Roles/ReadOnlyRestricted",
      "@odata.etag": "W/\"ABCDEFG\"",
      "AssignedPrivileges": ["Login"]
   }]
}
```

The following example response shows a partial failure of a deep `PATCH` where the ETag provided in the request for the `Role` resource named `ReadOnlyRestricted` was incorrect:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
Content-Length: <computed length>
ETag: <Resource collection ETag>
OData-Version: 4.0

{
   "Members": [{
      "@odata.id": "/redfish/v1/AccountService/Roles/OperatorRestricted",
      "@odata.etag": "W/\"ABCDEFG\"",
      "AssignedPrivileges": ["Login", "ConfigureComponents"]
   }, {
      "@odata.id": "/redfish/v1/AccountService/Roles/ReadOnlyRestricted",
      "@Message.ExtendedInfo": [{
         "@odata.type": "#Message.v1_1_1.Message",
         "MessageId": "Base.1.8.PreconditionFailed",
         "RelatedProperties": ["#/AssignedPrivileges"]
      }]
   }]
}
```

10 Service responses

This clause describes the responses that Redfish services can return to clients.

10.1 Response headers

HTTP defines headers for use in response messages. Table 13 defines those headers and their requirements for Redfish services:

- Redfish services shall return the *HTTP 1.1 Specification*-defined headers if the **Required** column contains **Yes**.
- Redfish services should return the *HTTP 1.1 Specification*-defined headers if the **Required** column contains **No**.
- Redfish clients shall be able to both understand and process all the *HTTP 1.1 Specification*-defined headers.

Header	Required	Supported values	Description
`Access-Control-Allow-Origin`	Yes	Fetch Living Standard, 3.2.3. HTTP responses	Prevents or allows requests based on originating domain. Prevents CSRF attacks.
`Allow`	Yes	`POST`, `PUT`, `PATCH`, `DELETE`, `GET`, `HEAD`	Shall be returned with the HTTP 405 (Method Not Allowed) status code 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.
`Cache-Control`	Yes	RFC7234	Shall be supported and indicates whether a response can or cannot be cached.
`Content-Encoding`	No	RFC7231	Encoding used to compress the message body.
`Content-Length`	No	RFC7231	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 the HTTP `411 Length Required` status code.

Header	Required	Supported values	Description
`Content-Type`	Yes	RFC7231	The message body's representation type. Services shall specify a `Content-Type` of `application/json` when returning resources as JSON. Services shall specify a `Content-Type` of `application/xml` when returning metadata as XML. Services shall specify a `Content-Type` of `application/yaml` or `application/vnd.oai.openapi` when returning OpenAPI schema as YAML. Services shall specify a `Content-Type` of `text/event-stream` when returning an SSE stream. `;charset=utf-8` shall be appended to the `Content-Type` if specified in the chosen media-type in the `Accept` header for the request.
`ETag`	Conditional	RFC7232	An identifier for a specific version of a resource, often a message digest. The `ETag` header shall be included on responses to `GET`s of `ManagerAccount` resources.
`Link`	Yes	RFC8288	`Link` headers shall be returned, as described in the Link headers clause.
`Location`	Conditional	RFC7231	URI of a newly created resource. Shall be returned upon creation of a resource. `Location` and `X-Auth-Token` shall be included on responses that create user sessions.
`Max-Forwards`	No	RFC7231	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
`OData-Version`	Yes	4.0	OData version of the payload to which the response conforms.
`Retry-After`	No	RFC7231, Section 7.1.3	Informs a client how long to wait before requesting the *task* information again.
`Server`	No	RFC7231	A product token and its version. Multiple product tokens may be listed. **Note:** Previous versions of the Specification marked this header as required. This has been changed as no use cases for requiring it have been identified.
`Via`	No	RFC7230	Defines the network hierarchy and recognizes message loops. Each pass inserts its own `Via` header.
`WWW-Authenticate`	Yes	RFC7617	Required for Basic and other optional authentication mechanisms. For details, see the Security details clause.
`X-Auth-Token`	Yes	Opaque encoded octet strings	Contains the authentication token for user sessions. The token value shall be indistinguishable from random.

: **Table 13 — Response headers**

10.2 Link header

The `Link` header provides metadata information on the accessed resource in response to a `HEAD` or `GET` request. The metadata information can include hyperlinks from the resource and JSON Schemas that describe the resource.

The following example shows the `Link` headers for a `ManagerAccount` with an `Administrator` role, in addition to 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 because the `@odata.type` in the resource indicates the appropriate version, or reference the versioned JSON Schema, which according to previous normative statements need to match the version 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.

10.3 Status codes

HTTP defines status codes that appear 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 processing 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 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:** For security implications of extended errors, See Security details.

Table 14 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. For usage and additional requirements imposed by this specification, see the **Description** column.

- Clients shall understand and be able to process the status codes in Table 14 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 Table 14 as defined in **Description** column.
- Redfish services should not return the HTTP `100` status code. Using the HTTP protocol for a multipass data transfer should be avoided, except for the upload of extremely large data.
- If no other status code in the `4XX` range is appropriate for client-side errors, the default status code should be the HTTP `400 Bad Request` status code.
- If no other status code in the `5XX` range is appropriate for service-side errors, the default status code should be the HTTP 500 Internal Server Error status code.

HTTP status code	Description
`200 OK`	Request completed successfully and includes a representation in its body.
`201 Created`	Request to create a resource completed successfully. The `Location` header shall be set to the canonical URI for the newly created resource. For `POST` (create) requests, the response body may include a representation of the newly created resource. For `POST` (action) requests, the response body shall include the action response.
`202 Accepted`	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. The response body may include a representation of the `Task` resource.
`204 No Content`	Request succeeded, but no content is being returned in the body of the response.
`301 Moved Permanently`	Requested resource resides under a different URI.
`302 Found`	Requested resource resides temporarily under a different URI.
`304 Not Modified`	Service has made a conditional `GET` request where access is allowed but the resource content has not changed. Either or both the `If-Modified-Since` and `If-None-Match` headers initiate conditional requests to save network bandwidth if no change has occurred. See HTTP 1.1, sections 14.25 and 14.26.
`400 Bad Request`	Request could not be processed because it contains invalid information, such as an invalid input field, or is missing a required value. The response body shall return an extended error as defined in the Error responses clause.
`401 Unauthorized`	Authentication credentials included with this request are missing or invalid.

HTTP status code	Description
`403 Forbidden`	Service recognized the credentials in the request but those credentials do not possess authorization to complete this request. This code is also returned when the user credentials provided need to be changed before access to the service can be granted. For details, see the Security details clause.
`404 Not Found`	Request specified a URI of a resource that does not exist.
`405 Method Not Allowed`	HTTP verb in the request, such as `DELETE`, `GET`, `HEAD`, `POST`, `PUT`, or `PATCH`, is not supported for this request URI. The response shall include an `Allow` header that provides a list of methods that the resource identified by the URI in the client request supports.
`406 Not Acceptable`	`Accept` header was specified in the request and the resource identified by this request cannot generate a representation that corresponds to one of the media types in the `Accept` header.
`409 Conflict`	Creation or update request could not be completed because it would cause a conflict in the current state of the resources that the platform supports. For example, a conflict occurred due to an attempt to set multiple properties that work in a linked manner by using incompatible values.
`410 Gone`	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 cannot determine whether the condition is permanent, client should use the HTTP `404 Not Found` status code. This response is cacheable unless otherwise indicated.
`411 Length Required`	Request did not use the `Content-Length` header to specify the length of its content but perhaps used the `Transfer-Encoding: chunked` header instead. The addressed resource requires the `Content-Length` header.
`412 Precondition Failed`	Precondition check, such as check of the `OData-Version`, `If-Match`, or `If-Not-Modified` header, failed.
`415 Unsupported Media Type`	Request specifies a `Content-Type` for the body that is not supported.
`428 Precondition Required`	Request did not provide the required precondition, such as an `If-Match` or `If-None-Match` header.
`431 Request Header Field Too Large`	Service is unwilling to process the request because either an individual header field or the collection of all header fields are too large.
`500 Internal Server Error`	Service encountered an unexpected condition that prevented it from fulfilling the request. The response body shall return an extended error as defined in the Error responses clause.
`501 Not Implemented`	Service does not currently support the functionality required to fulfill the request. This response is appropriate when the service does not recognize the request method and cannot support the method for any resource.

HTTP status code	Description
`503 Service Unavailable`	Service currently cannot 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. A service may also use this response to indicate that the service itself is undergoing maintenance, such as finishing initialization steps after reboot of the service.
`507 Insufficient Storage`	Service cannot build the response for the client due to the size of the response.

: **Table 14 — HTTP status codes**

10.4 OData metadata responses

10.4.1 OData metadata responses overview

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 can access the OData metadata at the `/redfish/v1/$metadata` URI.

A client can access the OData service document at the `/redfish/v1/odata` URI.

10.4.2 OData $metadata

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

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

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

10.4.2.1 Referencing other schemas

The OData metadata should include the namespaces for each of the Redfish resource types, along with the `RedfishExtensions.v1_0_0` namespace. Dynamic clients that reference the OData metadata document leverage schema definitions that are referenced to understand the definitions of the resources in the service. However, there are cases where it might not be practical to maintain an accurate document, such as when resources are dynamically discovered by the service through devices that support Redfish Device Enablement.

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.

10.4.2.2 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>
```

10.4.3 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 use the `application/json` MIME type to return the OData service document as a JSON object.

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.

Table 15 describes the properties that each JSON object entry includes:

Property	Description
`name`	User-friendly resource name of the resource.
`kind`	Type of resource. Value is `Singleton` for all cases defined by Redfish.
`url`	Relative URL for the top-level resource.

: **Table 15 — JSON object properties**

10.5 Resource responses

Services use the `application/json` MIME type to return resources and resource collections as JSON payloads. A service shall not break responses for a single resource into multiple results.

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

10.6 Error responses

HTTP 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 Bad Request` status code does not tell the client which values are in error. Error responses 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.

Table 16 describes the properties in the extended error response, which is a single JSON object:

Property	Description
`code`	String. Defines a `MessageId` from the message registry. See the MessageId format clause for the format of `MessageId`.
`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.

: **Table 16 — Error properties**

See the Schema definition languages clause for references to the schema definitions of the error response payload.

The `@Message.ExtendedInfo` property should be present in all error responses. If the `@Message.ExtendedInfo` property is present, all information necessary to process the error should be provided in the `@Message.ExtendedInfo` property. Clients should look for the `@Message.ExtendedInfo` property for error processing first, and fallback on the `code` and `message` properties if `@Message.ExtendedInfo` is not present.

The following sample error response contains two messages in the `@Message.ExtendedInfo` property that describe two different errors. The message described by the `code` and `message` properties do not provide actionable information for the client.

```
{
   "error": {
      "code": "Base.1.8.GeneralError",
      "message": "A general error has occurred. See Resolution for information on how to resolve the error.",
```

```
      "@Message.ExtendedInfo": [{
         "@odata.type": "#Message.v1_1_1.Message",
         "MessageId": "Base.1.8.PropertyValueNotInList",
         "RelatedProperties": ["#/IndicatorLED"],
         "Message": "The value Red for the property IndicatorLED is not in the list of acceptable values.",
         "MessageArgs": ["Red", "IndicatorLED"],
         "Severity": "Warning",
         "MessageSeverity": "Warning",
         "Resolution": "Choose a value from the enumeration list that the implementation can support and resubmit the request
      }, {
         "@odata.type": "#Message.v1_1_1.Message",
         "MessageId": "Base.1.8.PropertyNotWritable",
         "RelatedProperties": ["#/SKU"],
         "Message": "The property SKU is a read only property and cannot be assigned a value.",
         "MessageArgs": ["SKU"],
         "Severity": "Warning",
         "MessageSeverity": "Warning",
         "Resolution": "Remove the property from the request body and resubmit the request if the operation failed."
      }]
   }
}
```

11 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 can also define the payload in any encoding and the data model is intended to be schema-language agnostic. While Redfish uses the JSON data-interchange format, Redfish provides a common encoding type that ensures property naming conventions that make development easier in JavaScript, Python, and other languages. This encoding type helps the Redfish data model be more easily accessible in modern tools and programming environments.

The data model allows an OEM to extend the model by adding an *OEM resource* or *extending a resource*.

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

11.1 Resources

A *resource* is a single entity accessed at a specific URI. Services use the `application/json` MIME type to return resources as JSON payloads.

Each resource shall be strongly typed, defined by a resource type in a Redfish schema document, and identified in the response payload by the value of the 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 that resource type. Responses shall not include any properties not defined by that resource type.

11.2 Resource types

A *resource type* defines the set of properties that may be returned in the response payload of a Redfish resource request. Each resource type is documented in a Redfish schema document, and those documents are known collectively as the *Redfish schema*. The resource type may also include definitions for actions available for that resource.

Resource types are named to match the contents and purpose of the resource that they define. For example the `Circuit` resource type defines the properties and actions related to a single electrical circuit. Resource types provide global uniqueness for definitions across multiple schema files and allow for schema files to reference each other. Resource types may be defined by OEMs to extend the Redfish schema, and should follow the naming rules specified by the OEM resource types clause.

11.3 Resource collections

A *resource collection* is a set of resources that share the same schema definition. Services use the `application/json` MIME type to return resource collections as JSON payloads.

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.

11.4 OEM resources

OEMs and other third parties can extend the Redfish data model by creating additional resource types. Extending the data model is accomplished by defining an OEM resource type, and schema file, for each resource type, and creating hyperlinks to connect instances of new resources to the *resource tree*.

Companies, OEMs, and other organizations may also use the `Oem` property in resources, the links property, and the actions property to define additional properties, hyperlinks, and actions for standard Redfish resource types.

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.

11.5 Common data types

11.5.1 Primitive types

Table 17 describes the primitive data types for properties and action parameters in the data model:

Type	Description
Boolean	A variable with a value of `true` or `false`.
Number	A number with optional decimal point or exponent. Number properties may restrict the representation to an integer or a number with decimal point.
String	A sequence of characters enclosed with double quotes (`"`).
Array	A comma-separated set of the previous types enclosed with square braces (`[` and `]`). See the Array properties clause.
Object	A set of properties enclosed with curly braces (`{` and `}`). See the Structured properties clause.
Null	`null` value, which the service uses when it is unable to determine the property's value due to an error or other temporary condition, or if the schema has requirements for using `null` for other special conditions.

: **Table 17 — Primitive data types**

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.

11.5.2 Empty string values

String properties should return an empty string ("") for properties configured by a user or external service that have not been set to an initial value. This allows client software to identify the property as supported by the service, and avoids the use of `null`, which indicates an error condition. For example, the `AssetTag` property must be set by the end user, and therefore would return an empty string ("") until assigned a value by the user, while a failure to read the stored `AssetTag` value due to a non-volatile memory error would return `null`. To improve interoperability, implementations should avoid the use of filler strings, such as `N/A` or `<Empty>`, to represent a value not set by a user.

11.5.3 GUID and UUID values

Globally Unique Identifier (GUID) and 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})
```

11.5.4 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>-<DD>T<hh>:<mm>:<ss>[.<SSS>](Z|((+|-)<HH>:<MM>))
```

where

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

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

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

11.5.5 Duration values

Duration values are strings according to the ISO 8601 duration format, with the exception of not expressing a representation for years, months, or weeks. Duration values shall use the format:

```
P[<d>D][T[<h>H][<m>M][<s>[.<f>]S]]
```

where

- `<d>` is the number of days.
- `<h>` is the number of hours.
- `<m>` is the number of minutes.
- `<s>` is the number of seconds.
- `<f>` is the fractional seconds.

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

For example, Table 18 describes the following durations:

Value	Duration
`P90D`	Ninety days.
`P3D`	Three days.
`PT6H`	Six hours.
`PT10S`	Ten seconds.
`PT0.001S`	0.001 seconds.
`PT1H30M`	One hour and 30 minutes.

: **Table 18 — Durations**

> **DEPRECATED:** Duration values shall use the format: `P[<y>Y][<m>M][<w>W][<d>D][T[<h>H][<m>M][<s>[.<f>]S]]`. This definition allows for specifying years, months, and weeks. ISO 8601 does not specify an exact value for the duration of a year or of a month, which introduces interoperability challenges.

11.5.6 Reference properties

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

11.5.7 Non-resource reference properties

Non-resource reference properties provide a reference to services or documents that are not Redfish-defined resources. These properties shall include the `Uri` term in their property name. For example, `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.

11.5.8 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.

Table 19 describes the array types, regardless of the data type of the elements:

Array type	Description
Fixed length	Contains a static number of elements. The property definition sets or the implementation chooses the size of the array.
Variable length	Contains a variable number of elements. The array size is not specified and the size varies among instances. The array size may change. This array style is the most common style.
Rigid	The array index is meaningful. When elements are added to or removed from the array, the elements do not change their position, or index, in the array. An element that is removed from a rigid array shall be replaced by a `null` element and all other elements shall remain at their current index. Empty elements in a rigid array property shall be represented by `null` elements. Any array property that uses this style shall indicate the rigid style in the long description of its schema definition.

: **Table 19 — Array types**

Services may pad an array property with `null` elements at the end of the sequence to indicate the array size to clients. This practice 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.

11.5.9 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` property definition in the `Resource` schema has gone through several iterations since the `Resource.v1_1_0` type was introduced, and each iteration inherits from the earlier version so that existing references in other schemas can leverage the 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 the `Location` property 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.

11.5.10 Message object

11.5.10.1 Overview

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

Table 20 describes the properties of the message object, which is a JSON object:

Property	Type	Required	Defines
`MessageId`	String	Yes	Error or message. Do not confuse this value with the HTTP status code. Clients can use this code to access a detailed message from a message registry.
`Message`	String	No	Human-readable error message that indicates the semantics associated with the error. This shall be the complete message, and not rely on substitution variables.
`RelatedProperties`	An array of JSON pointers	No	Properties in a JSON payload that the message describes.
`MessageArgs`	An array of strings	No	Substitution parameter values for the message. If the parameterized message defines a `MessageId`, the service shall include the `MessageArgs` in the response.
`MessageSeverity`	String (enumeration)	No	Severity of the error. Services can replace the value of the `MessageSeverity` property defined in the message registry with a value more applicable to the implementation.

Property	Type	Required	Defines
`Severity`	String	No	Severity of the error. Services can replace the value of the `Severity` property defined in the message registry with a value more applicable to the implementation. **DEPRECATED:** This property has been deprecated in favor of `MessageSeverity`.
`Resolution`	String	No	Recommended actions to take to resolve the error. Services can replace the value of the `Resolution` property defined in the message registry with a service-defined resolution.

: **Table 20 — Message object properties**

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.

A `MessageId` identifies a specific message that a message registry defines.

11.5.10.2 MessageId format

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

`<RegistryName>.<MajorVersion>.<MinorVersion>.<MessageKey>`

where

- `<RegistryName>` is the name of the registry. The registry name shall be Pascal-cased.
- `<MajorVersion>` is a non-negative integer that represents the major version of the registry.
- `<MinorVersion>` is a non-negative integer that represents 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.

To search the message registry for a message, the client can use the `MessageId`.

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

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

When an implementation uses `GeneralError` from the Base Message Registry in `ExtendedInfo`, the implementation should include a service-defined value for the `Resolution` property with this error to indicate how to resolve the problem.

11.6 Properties

11.6.1 Properties overview

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, or by that particular resource instance. Differences in underlying product support or configuration varies among resource instances, and therefore the properties returned by each instance vary accordingly.
- If an implementation supports a property, it shall always provide a value for that property. If a value is unknown at the time of the operation due to an internal error, or inaccessibility of the data, the value of `null` is an acceptable value if supported by the schema definition.
- Resource instances should omit properties if the underlying product, service, or current configuration does not provide the function described by the property. For example, a chassis resource instance might not provide a serial number, and therefore should omit the `SerialNumber` property, while other chassis resource instances that have a serial number provide this property. See the Special resource situations clause for handling special resource situations.
- 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.

11.6.2 Resource identifier (@odata.id) property

Registry resources in a response may include an `@odata.id` property. All other resources and resource collections in a response shall include an `@odata.id` property. The value of the identifier property shall be the resource URI.

11.6.3 Resource type (@odata.type) property

All resources and resource collections 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 of the type property for resources and structured properties shall be in the format:

```
#<ResourceType>.<Version>.<TermName>
```

where

- `<ResourceType>` is the resource type in the Redfish schema that defines the resource.
- `<Version>` is the resource type version, in the format: `v<MajorVersion>_<MinorVersion>_<ErrataVersion>`.
- `<TermName>` is the specific type defined within the resource type definition. For most Redfish resources, the specific type name is the same as the resource type name.

An example of a resource type value is `#ComputerSystem.v1_0_0.ComputerSystem`, where `ComputerSystem.v1_0_0` denotes the version 1.0.0 of the `ComputerSystem` resource type, and the specific type is `ComputerSystem`.

The value of the type property for resource collections shall be in the format:

`#<ResourceType>.<ResourceType>`

where

- `<ResourceType>` is the resource type in the Redfish schema that defines the resource collection.

An example of a resource collection type value is `#ComputerSystemCollection.ComputerSystemCollection` for the `ComputerSystemCollection` resource collection.

11.6.4 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.

11.6.5 Resource context (@odata.context) property

Responses for resources and resource collections 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>.<ResourceType>`

where

- `<ResourceType>` is the resource type of the resource or resource collection.

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 that OData Protocol specifies.

11.6.6 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.

11.6.7 Name

The `Name` property conveys 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.

11.6.8 Description

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

11.6.9 MemberId

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

11.6.10 Count (Members@odata.count) property

The count property defines the total number of resource, 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. If the number of members available in the resource collection is reduced due to filtering, such as in response to the `$filter` query parameter, the count should be the total number of members available in the resource collection after the filter is applied.

11.6.11 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.

11.6.12 Next link (Members@odata.nextLink) property

The next link (`Members@odata.nextLink`) property value 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.

11.6.13 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.

11.6.13.1 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"
      }
   }
}
```

11.6.13.2 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"
      }]
   }
}
```

11.6.14 Actions property

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

11.6.14.1 Action representation

Each supported action is represented as a property nested under `Actions`. The unique name that identifies the action is used to construct the property name.

This property name shall be in the format:

`#<ResourceType>.<ActionName>`

where

- `<ResourceType>` is the resource where the action is defined.
- `<ActionName>` is 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. For details, see the Action info annotation clause.

11.6.14.2 Action responses

Response payloads for actions may contain a JSON body that is described by the schema definition for the action. See the Schema definition languages clause for the representation of these definitions. Actions that do not define a response body may provide an error response in the response payload.

11.6.15 Oem

The `Oem` property is used for extending standard resources with OEM extensions.

11.6.16 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.

11.7 Naming conventions

The Redfish interface is intended to be easily readable and intuitive. Thus, consistency helps the consumer understand the use of a newly discovered property. While consistency is no substitute for the normative information in the *Redfish Specification* and Redfish schema, the following naming rules help with readability and client usage. In general, names in Redfish are designed and intended to be human-readable and convey the meaning of the name, in context, without the need to consult schema definitions or other documentation.

11.7.1 Naming rules

Standard Redfish schema and registries defined and published by the DMTF, and those created by others and republished by the DMTF, shall follow a set of naming conventions. These conventions are intended to ensure consistent naming and eliminate naming collisions. For schema files, the resource type is used to construct the type property and the schema file name.

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 enables code re-use across resources and increases interoperability. New resource definitions should leverage existing property definitions whenever possible.

The general Redfish naming rules for resource types, registries, properties, action parameters, and enumerations are as follows:

- Names shall be Pascal-cased. The first letter of each word in a name shall be uppercase and spaces between words shall be removed. For example, `ComputerSystem`, `PowerState`, and `SerialNumber`.
- Names of array properties or reference properties for resource collections should use a plural form of the name. All other names should use the singular form of the name.
- Reference properties for resource collections should omit the term "collection" in the name.
- Names shall not contain spaces or underscore characters. Names should not contain any special characters that violate naming rules for supported schema description languages or programming languages.
- Both characters should be capitalized for two-character acronyms. For example, `IPAddress` or `RemoteIP`.
- Names constructed from a single acronym or mixed-case name, such as `LDAP`, `PCIe`, or `SNMP`, should use the typical capitalization for that name.

- Names incorporating acronyms with three or more characters should follow the capitalization used in related names for consistency. For example, `EnableSNMPv1` and `EnableSNMPv2` follow the pattern used for `SNMP`.
- Pascal-casing may be used for acronyms longer than two characters to improve readability, especially when two or more acronyms appear together in a name, which should be avoided.
- Enumeration names should start with a letter and be followed by letters or numbers to conform to schema description language requirements. Underscore characters may be used to replace other special characters, or to significantly improve readability, but this usage is discouraged.
- Enumeration names should prioritize readability as they may appear unmodified on user interfaces, whereas property or schema names should follow conventions and strive for consistency.
- The names `Settings` and `SD` are reserved for use for settings resources and shall not be used for schema names.

Exceptions are allowed for the following cases:

- Well-known technology abbreviations, acronyms, or product names should follow their defined capitalization. Examples include `iSCSI`, `iSCSITarget`, and `iLO`.
- OEM appears as `Oem` in schema and property names either alone or as a portion of a name, but should be `OEM` when used alone as an enumeration value.
- Underscore characters are allowed in the construction of OEM-specified object property names when required, and in OEM-defined resource types or OEM-defined registry names.

For properties that have units or other special meaning, append a unit identifier to the name. Examples include:

- Bandwidth (Mbps). For example, `PortSpeedMbps`.
- CPU speed (Mhz). For example, `ProcessorSpeedMhz`.
- Memory size (MB). For example, `MemoryMB`.
- Counts of items (Count). For example, `ProcessorCount` or `FanCount`.
- The state of a resource (State). For example, `PowerState`.
- State values where work is in process. For example, `Applying` or `ClearingLogic`.

11.7.2 URI naming rules

The following rules apply to Redfish schema-defined URIs:

- URI segments should generally follow the naming rules, and for each segment, follow the name of the property that provides the hyperlink.
- URI segments for resource collections should use the plural form of the resource collection schema name, with the `Collection` term omitted. For example, `Processors` for a `ProcessorCollection`.
- If a hyperlink to a subordinate 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.

11.8 Extending standard resources

11.8.1 Extending standard resources overview

In the context of this clause, the OEM term refers to any company, manufacturer, or organization that provides or defines an extension to the DMTF-published schema and functionality for Redfish. All Redfish-specified resources include an empty structured `Oem` property. The value of this predefined placeholder can encapsulate one or more OEM-specified object properties, which can contain OEM-specific property definitions.

11.8.2 OEM property format and content

Each property contained within the `Oem` property shall be an OEM-specified JSON object. The name of each object property shall uniquely identify the OEM or organization that defines the properties contained by that object. The OEM-specified object naming clause describes this naming convention.

The OEM-specified object shall include a type property if the object:

- Is not contained in an array of objects.
- Is contained in the first object within an array of objects.
- In subsequent array members containing an OEM-specified object, whose type is different than the first array member.

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-specified object, 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 elements within an OEM-specified object, it is intended it is typically used for only 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 creating a subordinate resource for their extensions.

11.8.3 OEM-specified object naming

The OEM-specified object properties within the `Oem` property are named by using a unique OEM identifier. 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 prefixed 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 prefixed with `EID:`.

Organizations that use `.com` domain names may omit the `.com` suffix. For example, Contoso.com would use `Contoso` instead of `Contoso_com`, but Contoso.org would use `Contoso_org`. 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.

The OEM identifier portion of the object name may be followed by an underscore (`_`) and any additional string to enable further subdivisions of OEM-specified objects as desired. For example, `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 try to parse nor interpret the content.

This suffix could be used in many ways, depending on OEM need. For example, the Contoso company may have a *Research* suborganization, in which case the OEM-specified property name might be extended to *Contoso_Research*. Alternatively, it can identify a unique resource type for a functional area, geography, subsidiary, and so on.

The OEM identifier portion of the name typically identifies the company or organization that created and maintains the schema for the property. However, this practice is not a requirement. The identifier is only required to uniquely identify the party that is the top-level manager of a resource type to prevent collisions between OEM property definitions from different vendors or organizations. Consequently, the organization for the top of the resource type may be different than the organization that provides the definition of the OEM-specified property. For example, Contoso may allow one of their customers, such as `CustomerA`, to extend a Contoso product with certain CustomerA proprietary properties. In this case, although Contoso allocated the name `Contoso_CustomerA`, it could be CustomerA that defines the content and functionality within that resource type. In all cases, OEM identifiers should not be used except with permission or as specified by the identified company or organization.

11.8.4 OEM resource types

Companies, OEMs, and other organizations can define additional resources and link to them from an `Oem` property in a standard Redfish resource, preferably from the `Oem` property within the `Links` property. To avoid naming collisions with current or future standard Redfish schema files, the defining organization's name should be prepended to the resource type name, which is also used to construct the file name of the schema files. For example, `ContosoDrive` would not conflict with a `Drive` resource type or schema, or conflict with another OEM's drive-related definition.

11.8.5 OEM registries

Companies, OEMs, and other organizations can define additional registries and utilize them in message objects or for BIOS attributes. To avoid naming collisions with current or future standard Redfish message registries, the defining organization's name should be prepended to the registry name, which is also used to construct the name of the registry file. For example, `ContosoDriveEvents` would not conflict with a `DriveEvent` message registry, or conflict with another OEM's drive-related registry.

11.8.6 OEM URIs

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:

`<BaseUri>/Oem/<OemName>/<ResourcePath>`

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-specified object naming clause.
- `<ResourcePath>` is the path to the OEM-defined resource. This path might contain multiple segments for cases where OEM-defined resources are subordinate to an OEM-defined resource. Each segment in the path contains the name of an OEM-defined resource.

For example, if Contoso defined a new `ContosoAccountServiceMetrics` resource to be linked through the `Oem` property at the `/redfish/v1/AccountService` URI, the OEM resource has the `/redfish/v1/AccountService/Oem/Contoso/AccountServiceMetrics` URI.

11.8.7 OEM property examples

The following fragment shows examples of naming and 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": "#ContosoAnvil.v1_2_1.AnvilTypes1",
         "Slogan": "Contoso anvils never fail",
         "Disclaimer": "* Most of the time"
      },
      "Contoso_biz": {
         "@odata.type": "#ContosoBizEngine.v1_1_0.RelatedSpeed",
         "Speed": "Ludicrous"
      },
      "EID_412": {
         "@odata.type": "#AdatumPowerExtensions.v1_0_1.PowerInfoExt",
         "ReadingInfo": {
            "Accuracy": "5",
            "IntervalSeconds": "20"
```

```
            }
         },
         "Contoso_CustomerA": {
            "@odata.type": "#ContosoCustomerASling.v1_0_0.SlingPower",
            "AvailableTargets": ["Rabbit", "Duck", "Runner"],
            "LaunchPowerOptions": ["Low", "Medium", "Eliminate"],
            "LaunchPower": "Eliminate",
            "Target": "Rabbit"
         }
      },
      ...
}
```

11.8.8 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, which shall follow the form:

`#<ResourceType>.<Action>`

where

- `<ResourceType>` is the resource type.
- `<Action>` is the action.

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

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

`<ResourceUri>/Actions/Oem/<ResourceType>.<Action>`

where

- `<ResourceUri>` is the URI of the resource that supports invoking the action. For example, `/redfish/v1/Systems/1/`.
- `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.
- `<ResourceType>.<Action>` is the resource type followed by the action. For example, `Contoso_ABC_ComputerSystem.Ping`.

11.9 Payload annotations

11.9.1 Payload annotations overview

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

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

where

- `<PropertyName>` is the name of the property to annotate. If absent, the annotation applies to the entire JSON object, which may be an entire resource.
- `<Namespace>` is the namespace that defines the annotation term.
- `<TermName>` is 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.

11.9.2 Allowable values

To specify the list of allowable values for a property or action parameter, services may 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 action parameter, followed by `@Redfish.AllowableValues`. The property value is a JSON array of strings that define the allowable values for the property or action parameter.

11.9.3 Extended information

The following clauses describe the methods of providing extended information:

- Extended object information
- Extended property information

11.9.3.1 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.1.8.PropertyDuplicate",
      "Message": "Indicates that a duplicate property was included in the request body.",
      "RelatedProperties": ["#/InterfaceEnabled"],
      "Severity": "Warning",
      "MessageSeverity": "Warning",
      "Resolution": "Remove the duplicate property from the request body and resubmit the request if the operation failed."
   }]
}
```

The property contains an array of message objects.

11.9.3.2 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.1.8.PropertyValueNotInList",
      "Message": "The value Contoso for the property PinOut is not in the list of acceptable values.",
      "Severity": "Warning",
      "MessageSeverity": "Warning",
      "Resolution": "Choose a value from the enumeration list that the implementation can support and resubmit the request if
   }]
}
```


11.9.4 Action info annotation

The `@Redfish.ActionInfo` term within the action representation conveys the parameter requirements and allowable values on parameters for actions. 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"]
   }]
}
```

11.9.5 Settings and settings apply time annotations

See the Settings resource clause.

11.9.6 Operation apply time and operation apply time support annotations

See the Operation apply time clause.

11.9.7 Maintenance window annotation

The settings apply time and operation apply time annotations enable an operation to be performed during a maintenance window. The `@Redfish.MaintenanceWindow` term at the root of a resource configures the start time and duration of a maintenance window for a resource.

The following example body for the `/redfish/v1/Systems/1` resource configures the maintenance window to start at `2017-05-03T23:12:37-05:00` and last for 600 seconds.

```
{
   "@odata.id": "/redfish/v1/Systems/1",
   "@odata.type": "#ComputerSystem.v1_5_0.ComputerSystem",
```

```
   "@Redfish.MaintenanceWindow": {
      "MaintenanceWindowStartTime": "2017-05-03T23:12:37-05:00",
      "MaintenanceWindowDurationInSeconds": 600
   },
   ...
}
```

11.9.8 Collection capabilities annotation

Resource collections may contain a collection capabilities annotation. The `@Redfish.CollectionCapabilities` term at the root of a resource collection shows what properties a client is allowed to use in a POST request for creating a resource.

The following `ComputerSystemCollection` example body contains the collection capabilities annotation. The `UseCase` property contains the `ComputerSystemComposition` value, and the `CapabilitiesObject` property contains the `/redfish/v1/Systems/Capabilities` value. The resource at `/redfish/v1/Systems/Capabilities` describes the `POST` request format for creating a `ComputerSystem` resource for compositions.

```
{
   "@odata.id": "/redfish/v1/Systems",
   "@odata.type": "#ComputerSystemCollection.ComputerSystemCollection",
   "Name": "Computer System Collection",
   "Members@odata.count": 0,
   "Members": [],
   "@Redfish.CollectionCapabilities": {
      "@odata.type": "#CollectionCapabilities.v1_1_0.CollectionCapabilities",
      "Capabilities": [{
         "CapabilitiesObject": {
            "@odata.id": "/redfish/v1/Systems/Capabilities"
         },
         "UseCase": "ComputerSystemComposition",
         "Links": {
            "TargetCollection": {
               "@odata.id": "/redfish/v1/Systems"
            }
         }
      }]
   }
}
```

The `CapabilitiesObject` resource follows the same schema for the resource that the resource collection contains. It contains annotations to show which properties the client can use in the `POST` request body. Table 21 describes the `CapabilitiesObject` resource annotations. These annotations describe which properties are required, optional, or if other rules are associated with the properties.

Annotation	Description
`<PropertyName>@Redfish.RequiredOnCreate`	Required in the `POST` request body.
`<PropertyName>@Redfish.OptionalOnCreate`	Not required in the `POST` request body.
`<PropertyName>@Redfish.SetOnlyOnCreate`	Cannot be modified after the resource is created.
`<PropertyName>@Redfish.UpdatableAfterCreate`	Can be modified after the resource is created.
`<PropertyName>@Redfish.AllowableValues`	Can be set to any of the listed values.
`@Redfish.RequestedCountRequired`	Required in the `POST` request body for the corresponding object to indicate the number of requested object instances. Used for composition requests.
`@Redfish.ResourceBlockLimits`	Indicates restrictions regarding quantities of `ResourceBlock` resources of a given type in the `POST` request body. Used for composition requests.

: **Table 21 — CapabilitiesObject resource annotations**

Example `CapabilitiesObject` resource:

```
{
   "@odata.id": "/redfish/v1/Systems/Capabilities",
   "@odata.type": "#ComputerSystem.v1_8_0.ComputerSystem",
   "Id": "Capabilities",
   "Name": "Capabilities for the system collection",
   "Name@Redfish.RequiredOnCreate": true,
   "Name@Redfish.SetOnlyOnCreate": true,
   "Description@Redfish.OptionalOnCreate": true,
   "Description@Redfish.SetOnlyOnCreate": true,
   "HostName@Redfish.OptionalOnCreate": true,
   "HostName@Redfish.UpdatableAfterCreate": true,
   "Links@Redfish.RequiredOnCreate": true,
   "Links": {
      "ResourceBlocks@Redfish.RequiredOnCreate": true,
      "ResourceBlocks@Redfish.UpdatableAfterCreate": true
   },
   "@Redfish.ResourceBlockLimits": {
      "MinCompute": 1,
      "MaxCompute": 1,
      "MaxStorage": 8
   }
}
```

11.9.9 Requested count and allow over-provisioning annotations

Table 22 describes the `@Redfish.RequestedCount` and `@Redfish.AllowOverprovisioning` annotations.

Clients use these annotations in composition requests to define the number of resource to allocate and to indicate whether the Redfish service can provision more resources than the client requests:

Annotation	Description
`@Redfish.RequestedCount`	Number of requested resources.
`@Redfish.AllowOverprovisioning`	Boolean. If `true`, the service may provision more resources than the `@Redfish.RequestedCount` annotation requests. Default is `false`.

: **Table 22 — RequestCount and AllowOverprovisioning annotations**

Example client request for at least four and possibly more `Processor` resources:

```
{
   "Processors": {
      "Members": [{
         "@Redfish.RequestedCount": 4,
         "@Redfish.AllowOverprovisioning": true
      }]
   },
   ...
}
```

11.9.10 Zone affinity annotation

The zone affinity annotation is used by clients in composition requests to indicate the components for the composition come from the specified resource zone. The `@Redfish.ZoneAffinity` term in the request body contains the value of the `Id` property of the requested resource zone.

Example client request for components to be allocated from the resource zone with the `Id` property containing `1`:

```
{
   "@Redfish.ZoneAffinity": "1",
   ...
}
```

11.9.11 Supported certificates annotation

Resource collections of type `CertificateCollection` should contain a supported certificates annotation. The `@Redfish.SupportedCertificates` term at the root of a resource collection shows the different certificate formats allowed in the resource collection.

Example `CertificateCollection` that only supports PEM style certificates:

```
{
   "@odata.id": "/redfish/v1/Managers/BMC/NetworkProtocol/HTTPS/Certificates",
   "@odata.type": "#CertificateCollection.CertificateCollection",
   "Name": "Certificate collection",
   "Members@odata.count": 1,
   "Members": [{
      "@odata.id": "/redfish/v1/Managers/BMC/NetworkProtocol/HTTPS/Certificates/1"
   }],
   "@Redfish.SupportedCertificates": ["PEM"]
}
```

11.9.12 Deprecated annotation

Services may annotate properties with `@Redfish.Deprecated` if the schema definition has the property marked as deprecated.

Example deprecated property:

```
{
   "VendorID": "0xABCD",
   "VendorID@Redfish.Deprecated": "This property has been deprecated in favor of ModuleManufacturerID.",
   ...
}
```

11.10 Settings resource

A settings resource represents 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 future point in time, such as after a system reset. While the *active resource* represents the current state, the settings resource represents the future intended state.

For resources that support a future intended state, the response shall contain a property with the `@Redfish.Settings` payload annotation. When a settings annotation is used, the following conditions shall apply:

- The settings resource shall be of the same schema definition as the active resource.
- The settings resource should contain a subset of updatable properties from the active resource. Additionally, it shall contain required properties, which are always mandatory.
- The settings resource shall not contain the `@Redfish.Settings` annotation.
- The settings resource may contain the `@Redfish.SettingsApplyTime` annotation.
- The URI for the settings resource shall reflect that it is subordinate to the active resource. The URI should be in the form `<BaseUri>/Settings` or `<BaseUri>/SD` where `<BaseUri>` is the URI of the active resource.

The settings resource shall contain the properties that are updated at a future point in time. For resources that support a future intended state, Table 23 describes the behavior of supported properties in the resource and settings resource that a service should support.

Property	Active resource behavior	Settings resource behavior
Read-only, required.	Returned in the resource response to a `GET` request.	Returned in the settings resource response to a `GET` request.
Read-only, not required.	Returned in the resource response to a `GET` request.	Not returned in the settings resource response to a `GET` request.
Writable, updates immediately, but not at a future point in time.	Active value returned in the resource response to a `GET` request. Modification requests change the active value immediately.	Not returned in the settings resource response to a `GET` request. Modification requests are rejected.
Writable, updates immediately or at a future point in time.	Active value returned in the resource response to a `GET` request. Modification requests change the active value immediately.	Future value returned in the settings resource response to a `GET` request if a future value is pending, otherwise not returned. Modification requests change the future value.
Writable, updates at a future point in time, but not immediately.	Active value returned in the resource response to a `GET` request. Modification requests are rejected.	Future value returned in the settings resource response to a `GET` request. Modification requests change the future value.

: **Table 23 — Active resource and settings resource property behavior**

The `@Redfish.Settings` annotation includes several properties that help clients monitor when the service has consumed the active resource and determine the success or failure of applying the values.

- 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. Immediate updates made directly to the active resource are not reflected in it.

- The `Time` property indicates the time when the settings resource was last applied. Immediate updates made directly to the active resource are not reflected in it.

The following active resource example body supports a settings resource. A client can use the `SettingsObject` property to locate the URI of the settings resource.

```
{
   "@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.1.8.PropertyNotWritable",
         "RelatedProperties": ["#/Attributes/ProcTurboMode"]
      }]
   },
   ...
}
```

If a service enables a client to indicate when to apply settings:

- The settings resource shall contain a property with the `@Redfish.SettingsApplyTime` annotation.
  - Only settings resources shall contain the `@Redfish.SettingsApplyTime` annotation.
- The `@Redfish.Settings` annotation in the active resource shall contain the `SupportedApplyTimes` property for showing the allowable values for `ApplyTime` within `@Redfish.SettingsApplyTime`.
- Clients can modify the `@Redfish.SettingsApplyTime` annotation to indicate when to apply the settings.

In the following example request, the client indicates that the settings resource values are applied on reset during the specified maintenance window:

```
{
   "@Redfish.SettingsApplyTime": {
      "ApplyTime": "InMaintenanceWindowOnReset",
      "MaintenanceWindowStartTime": "2017-05-03T23:12:37-05:00",
      "MaintenanceWindowDurationInSeconds": 600
   },
   ...
}
```

11.11 Special resource situations

11.11.1 Overview

Resources need to exhibit common semantic behavior whenever possible. This can be difficult in some situations discussed in this clause.

11.11.2 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 because it indicates a capability, such as 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. For example, 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 the `Absent` value for the `State` property in the `Status` object.

In this circumstance, any required properties that have 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. For example, 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.

11.12 Registries

Registry resources assist the client in interpreting Redfish resources beyond the Redfish schema definitions. To get

more information about a resource, event, message, or other item, use an identifier to search registries. This information can include other properties, property restrictions, and the like. Registries are themselves resources.

Table 24 describes the types of registries that Redfish supports:

Registry	Description	See
BIOS	Determines the semantics of each property in a BIOS or BIOS settings resource. Because BIOS information can vary from platform to platform, Redfish cannot define a fixed schema for these values. BIOS registries should be assigned unique identifiers to allow users to match a given registry with compatible products. This registry contains both property descriptions and other information, such as data type, allowable values, and user menu information.	
Message	Constructs a message from a `MessageId` and other message information to present to an end user. The messages in these registries appear in both eventing and error responses to operations. This registry is the most common type of registry.	• Error responses • Eventing
Privilege	Maps the resources in a Redfish service to the privileges that can complete specified operations against those resources. A client can use this information to: • Determine which roles should have specific privileges. • Map accounts to those roles so that the accounts can complete operations on Redfish resources.	Privilege model

: **Table 24 — Registries**

11.13 Schema annotations

11.13.1 Schema annotations overview

The schema definitions of the data model use schema annotations to provide additional documentation for developers. This clause describes the different types of schema annotations that the Redfish data model uses. For information about how each of the annotations are implemented in their respective schema languages, see the Schema definition languages clause.

11.13.2 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.

All schemas that are published or republished by the DMTF's Redfish Forum shall include a description annotation on the following schema definitions:

- Redfish types
- Properties
- Reference properties
- Enumeration values
- Resources and resource collections
- Structured types

11.13.3 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.

When the long descriptions in the Redfish schema contain normative language, the service shall be required to conform with the statement.

All schemas that are published or republished by the DMTF's Redfish Forum shall include a long description annotation on the following schema definitions:

- Redfish types
- Properties
- Reference properties
- Resources and resource collections
- Structured types

11.13.4 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 can invoke on the given resource or resource collection.

- Insert capabilities indicate whether a client can perform a `POST` request on the resource to create a resource.
- Update capabilities indicate whether a client can perform a `PATCH` or `PUT` request on the resource.
- Delete capabilities indicate whether a client can perform a `DELETE` request on the resource.
- A service may implement a subset of the capabilities that are allowed on the resource or resource collection.

All schemas that are published or republished by the DMTF's Redfish Forum for resources and resource collections shall include resource capabilities annotations.

11.13.5 Resource URI patterns annotation

The resource URI patterns annotation expresses the valid URI patterns for a 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 concatenated to form the URI pattern are in the URI naming rules clause.

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

Using the previous example, `{ManagerAccountId}` is replaced by the `Id` property of the corresponding `ManagerAccount` resource. If the `Id` property for a `ManagerAccount` resource is `John`, the full URI for that resource is `/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 is `/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 appears 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 is `/redfish/v1/Managers/{ManagerId}/NetworkProtocol/HTTPS/Certificates`.

All schemas that are published or republished by the DMTF's Redfish Forum for resources and 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.

11.13.6 Additional properties annotation

The additional properties annotation specifies 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.

11.13.7 Permissions annotation

The permissions annotation specifies whether a client can modify the value of a property, or if the property is read-only.

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

11.13.8 Required annotation

The required annotation specifies whether a service needs to support a property. Required properties shall be annotated with the required annotation. All other properties are optional.

11.13.9 Required on create annotation

The required on create annotation specifies 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.

11.13.10 Units of measure annotation

In addition to following the naming rules, 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. For example, `RPM` commonly reports 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 `Mi` prefix and `By` symbol, would use `MiBy` as the value for the annotation. For values that also include rate information, such as megabits per second, the rate unit's symbol should be appended and use a slash (`/`) character as a separator. For example, `Mbit/s` .

11.13.11 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.

11.13.12 Owning entity annotation

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

11.13.13 Deprecated annotation

The deprecated annotation specifies if a property, enumeration, or other schema element has been deprecated. Schema elements marked as deprecated contain a schema version that shows when the element was deprecated, as well as text that specifies the favored approach.

Existing and new implementations may use deprecated schema elements, but they should move to the favored approach. Deprecated schema elements may be implemented to achieve backwards compatibility. Deprecated schema elements may be removed from the next major version of the schema.

11.14 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.

References from other resources use the unversioned definition of a resource to ensure no version dependencies exist 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 resource. The version of a resource follows the format:

`v<X>.<Y>.<Z>`

where

- `<X>` is an integer that represents the major version. Indicates a backward-incompatible change.
- `<Y>` is an integer that represents the minor version. Indicates 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>` is an integer that represents the errata version. Indicates a fix in an earlier version. For example, a fix to a schema annotation on a property introduces an errata version of the resource.

11.15 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 non-English languages shall use the appropriate schema annotations to identify their language. Descriptive property, parameter, and enumeration text not translated into the specified language shall be removed from localized versions. This removal enables software and tools to combine normative and localized copies, especially for minor schema version differences.

12 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 Schema definition languages clause describes the file name construction rules, while the following clauses describe the construction rules for other file types.

12.1 Registry file naming

Redfish message or privilege registry files shall use the registry name to construct the file name, in this format:

`<RegistryName>.<MajorVersion>.<MinorVersion>.<Errata>.json`

For example, the file name of the Base Message Registry v1.0.2 is `Base.1.0.2.json`.

12.2 Profile file naming

The document that describes a profile follows the Redfish schema file naming conventions. The file name format for profiles shall be:

`<ProfileName>.v<MajorVersion>_<MinorVersion>_<Errata>.json`

For example, the file name of the BasicServer profile v1.2.0 is `BasicServer.v1_2_0.json`. The file name shall include the profile name and version, which matches those property values within the document.

12.3 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. Because a single dictionary file contains all minor revisions of the schema, only the major version appears in the file name. The file names for Dictionaries shall be formatted as:

`<DictionaryName>_v<MajorVersion>.dict`

For example, the file name of the Chassis dictionary v1.2.0 is `Chassis_v1.dict`.

12.4 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 and which 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.

12.5 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 for download. Programs may use the following durable URLs to access the repository. 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 use the form at https://redfish.dmtf.org/redfish/portal to submit those files to the DMTF for republication in the DMTF Redfish file repository.

Table 25 describes how files are organized on the site:

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 two-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 two-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.

: **Table 25 — Redfish file repository**

13 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

13.1 OData Common Schema Definition Language

13.1.1 OData Common Schema Definition Language overview

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.

13.1.2 File naming conventions for CSDL

Redfish CSDL schema files shall be named using the resource type name for the schema, followed by `_v` and the major version of the schema. Because a single CSDL schema file contains all minor revisions of the schema, only the major version appears in the file name. The file name shall be formatted as:

`<ResourceType>_v<MajorVersion>.xml`

For example, version 1.3.0 of the Chassis schema is `Chassis_v1.xml`.

13.1.3 Core CSDL files

Table 26 describes the core CSDL files:

File	Description
`RedfishError_v1.xml`	Payload definition of the Redfish error response.
`RedfishExtensions_v1.xml`	All definitions for Redfish types and annotations.
`Resource_v1.xml`	All base definitions for resources, resource collections, and common properties, such as `Status`.

: **Table 26 — Core CSDL files**

13.1.4 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.

13.1.4.1 Referencing other CSDL files

CSDL files may use `Reference` tags to reference types defined in other CSDL documents.

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>
```

13.1.4.2 CSDL data services

Define structures, enumerations, and other definitions in CSDL within a namespace. Use a `Schema` tag to define the schema and use the `Namespace` attribute to declare the name of the namespace.

Redfish uses namespaces to differentiate different versions of the schema. CSDL enables structures to inherit from other structures, which enables newer namespaces to define only the changes. The Elements of CSDL namespaces clause describes this behavior.

Namespaces containing unversioned resource and resource collection definitions shall use the resource type to name the namespace, in this format:

`<ResourceType>`

For example, the unversioned namespace of the Chassis resource is `Chassis`.

Namespaces containing versioned resource definitions shall use the resource type to name the namespace, in this format:

`<ResourceType>.v<MajorVersion>_<MinorVersion>_<Errata>`

For example, the version 1.3.0 namespace of the Chassis resource is `Chassis.v1_3_0`.

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>
```

13.1.5 Elements of CSDL namespaces

The following clauses describe the definitions within each namespace:

- Qualified names

- Entity type and complex type elements

13.1.5.1 Qualified names

Many definitions in CSDL use references to qualified names. CSDL defines this as a string in the form:

`<Namespace>.<TypeName>`

where

- `<Namespace>` is the namespace name.
- `<TypeName>` is the name of the element in the namespace.

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

13.1.5.2 Entity type and complex type elements

Use the `EntityType` and `ComplexType` tags to define the entity type and complex type elements, respectively. These elements define a JSON structure and their set of properties by defining property elements and navigation property elements 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 definitions of the referenced `BaseType` are 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 use the entity type element that inherits from `Resource.v1_0_0.ReferenceableMember`. The entity type element enables references to be made by using the Navigation Property element, whereas the 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="The TypeA entity type description."/>
  <Annotation Term="OData.LongDescription" String="The TypeA entity type normative description."/>
  <!-- Property and navigation property definitions go here -->
</EntityType>
<ComplexType Name="PropertyTypeA">
```

```
  <Annotation Term="OData.Description" String="The TypeA structured property description."/>
  <Annotation Term="OData.LongDescription" String="The TypeA structured property normative description."/>
  <!-- Property and navigation property definitions go here -->
</ComplexType>
```

13.1.5.3 Action element

Use the `Action` tag to define the action element. This element defines 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, preceded by `#`.

In Redfish, all action elements contain the `IsBound` attribute that is always set to `true`, which indicates that the action appears as a member of a structured type.

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

Because all action elements in Redfish use the `IsBound="true"` term, the first parameter is called the *binding parameter* and specifies the structured type to which the action belongs. In Redfish, this parameter is always 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>
```

Some action parameters may specify a type that is defined by an entity type element. In these cases, the parameter in the request is a reference object to a resource within the service.

13.1.5.4 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>
```

13.1.5.5 Action with a response body

A response body for an action shall be defined using the `ReturnType` tag within an Action element. For example, the following definition defines the `GenerateCSR` action with a response that contains the definition specified by `GenerateCSRResponse`.

```
<Action Name="GenerateCSR" IsBound="true">
  <Parameter Name="CertificateService" Type="CertificateService.v1_0_0.Actions"/>
    ...
  <ReturnType Type="CertificateService.v1_0_0.GenerateCSRResponse" Nullable="false"/>
</Action>

<ComplexType Name="GenerateCSRResponse">
  <Annotation Term="OData.AdditionalProperties" Bool="false"/>
  <Annotation Term="OData.Description" String="The response body for the GenerateCSR action."/>
  <NavigationProperty Name="CertificateCollection"
    Type="CertificateCollection.CertificateCollection" Nullable="false">
    <Annotation Term="OData.Permissions" EnumMember="OData.Permission/Read"/>
    <Annotation Term="OData.Description"
      String="The link to the certificate resource collection where the certificate is installed."/>
    <Annotation Term="Redfish.Required"/>
  </NavigationProperty>
  <Property Name="CSRString" Type="Edm.String" Nullable="false">
    <Annotation Term="OData.Permissions" EnumMember="OData.Permission/Read"/>
    <Annotation Term="OData.Description" String="The string for the certificate signing request."/>
    <Annotation Term="Redfish.Required"/>
  </Property>
</ComplexType>
```

Using the above example, the following payload is an example response for the `GenerateCSR` action.

```
{
   "CSRString": "-----BEGIN CERTIFICATE REQUEST-----...-----END CERTIFICATE REQUEST-----",
   "CertificateCollection": {
      "@odata.id": "/redfish/v1/Managers/BMC/NetworkProtocol/HTTPS/Certificates"
   }
}
```

13.1.5.6 Property element

Properties of resources, resource collections, and structured properties are defined using the property element. The `Property` tag defines a property element inside 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 names or types:

- 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 the previous names or types by using the `Collection` term.

Table 27 describes the primitive data types:

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.

: **Table 27 — Primitive data types**

Property elements may specify a `Nullable` attribute. If the attribute is `false`, the property value cannot be `null`. If the attribute is `true` or absent, the property value can be `null`.

Example property element:

```
<Property Name="Property1" Type="Edm.String" Nullable="false">
  <Annotation Term="OData.Description" String="The Property1 property description."/>
  <Annotation Term="OData.LongDescription" String="The Property1 property normative description."/>
  <Annotation Term="OData.Permissions" EnumMember="OData.Permission/Read"/>
  <Annotation Term="Redfish.Required"/>
  <Annotation Term="Measures.Unit" String="Watts"/>
</Property>
```

13.1.5.7 Navigation property element

Reference properties of resources, resource collections, and structured properties are defined using the navigation property element. The `NavigationProperty` tag defines a navigation property element inside 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 attribute is `false`, the property value cannot be `null`. If the attribute is `true` or absent, the property value can be `null`.

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="The RelatedType navigation property description."/>
  <Annotation Term="OData.LongDescription"
    String="The RelatedType navigation property normative description."/>
  <Annotation Term="OData.AutoExpandReferences"/>
</NavigationProperty>
```

13.1.5.8 Enum type element

Use the `EnumType` tag to define the enum type element. This element defines a set of enumeration values, which may be applied to one or more properties.

All enum type elements contain a `Name` attribute, which specifies the name of the set of enumeration values.

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

```
<EnumType Name="EnumTypeA">
  <Annotation Term="OData.Description" String="The EnumTypeA enum type description."/>
  <Annotation Term="OData.LongDescription" String="The EnumTypeA enum type normative description."/>
  <Member Name="MemberA">
    <Annotation Term="OData.Description" String="The description of MemberA"/>
  </Member>
  <Member Name="MemberB">
    <Annotation Term="OData.Description" String="The description of MemberB"/>
  </Member>
</EnumType>
```

13.1.5.9 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 Redfish schema annotation is expressed in CSDL.

- The OData Core Schema defines terms with the `OData` prefix.
- The OData Measures Schema defines terms with the `Measures` prefix.
- The RedfishExtensions Schema defines terms with the `Redfish` prefix.

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"/>
```

Example deprecated annotation:

```
<Annotation Term="Redfish.Revisions">
  <Collection>
    <Record>
      <PropertyValue Property="Kind" EnumMember="Redfish.RevisionKind/Deprecated"/>
      <PropertyValue Property="Version" String="v1_3_0"/>
      <PropertyValue Property="Description"
        String="This property has been deprecated in favor of ModuleManufacturerID."/>
    </Record>
```

```
    </Collection>
  </Annotation>
```

13.2 JSON Schema

13.2.1 JSON Schema overview

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.

13.2.2 File naming conventions for JSON Schema

Each Redfish JSON Schema file represents a single resource type.

Versioned Redfish JSON Schema files shall use the resource type to name the file, in this format:

`<ResourceType>.v<MajorVersion>_<MinorVersion>_<Errata>.json`

For example, version 1.3.0 of the Chassis schema is `Chassis.v1_3_0.json`.

Unversioned Redfish JSON Schema files shall use the resource type to name the file, in this format:

`<ResourceType>.json`

For example, the unversioned definition of the Chassis schema is `Chassis.json`.

13.2.3 Core JSON Schema files

Table 28 describes the core JSON Schema files:

File	Description
`odata-v4.json`	Definitions for common OData properties.
`redfish-error.v1_0_0.json` and its subsequent versions	Payload definition of the Redfish error response.
`redfish-schema-v1.json`	Extensions to the JSON Schema that define Redfish JSON Schema files.
`Resource.json` and its subsequent versions	All base definitions for resources, resource collections, and common properties, such as `Status`.

: **Table 28 — Core JSON Schema files**

13.2.4 JSON Schema format

Each JSON Schema file contains a JSON object to describe resources, resource collections, and other definitions for the data model.

Table 29 describes the JSON object, which contains the following terms:

Term	Description
`$id`	Reference to the URI where the schema file is published.
`$ref`	For a schema file that describes a resource or resource collection, the reference to the structural definition of the resource or resource collection.
`$schema`	URI to the Redfish schema extensions for JSON Schema. The value should be `http://redfish.dmtf.org/schemas/v1/redfish-schema-v1.json`.
`copyright`	Copyright statement for the organization producing the JSON Schema.
`definitions`	Structures, enumerations, and other definitions defined by the schema.
`title`	For a schema file that describes a resource or resource collection, the matching type identifier for the resource or resource collection.

: **Table 29 — JSON Schema format**

13.2.5 JSON Schema definitions body

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

13.2.5.1 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 definitions:

- 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.

The following example shows an 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 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.

13.2.5.2 Enumerations in JSON Schema

Table 30 describes the terms that constitute definitions for enumerations:

Term	Description
`enum`	String array that contains the possible enumeration values.
`enumDescriptions`	Object that contains the descriptions for each of the enumerations as name-value pairs.
`enumLongDescriptions`	Object that contains the long descriptions for each of the enumerations as name-value pairs.
`enumDeprecated`	Object that contains the deprecation guidance for each of the enumerations as name-value pairs.
`enumVersionDeprecated`	Object that contains the deprecation version information for each of the enumerations as name-value pairs.
`type`	Because all enumerations in Redfish are strings, the `type` term always has the `string` value.

: **Table 30 — JSON Schema enumerations**

The following example shows an 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
         "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"
   },
   ...
}
```

13.2.5.3 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:

`#<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 describes 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 response payloads for the resource.

The following example shows a definition of an action:

```
{
   "Reset": {
      "additionalProperties": false,
      "description": "This action resets 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"
   },
   ...
}
```

Some action parameters may specify a type that is a resource definition. In these cases, the parameter in the request is a reference object to a resource within the service.

13.2.5.4 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 `#ContosoNetworkDevice.Ping` action, assuming it's found in the versioned ContosoNetworkDevice JSON Schema file, such as `ContosoNetworkDevice.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"
   },
   ...
}
```

13.2.5.5 Action with a response body

A response body for an action shall be defined using the `actionResponse` term within the action definition. For example, the following definition defines the `GenerateCSR` action with a response that contains the definition specified by `#/definitions/GenerateCSRResponse`.

```
{
   "GenerateCSR": {
      "actionResponse": {
         "$ref": "#/definitions/GenerateCSRResponse"
      },
      "parameters": {}
   },
   "GenerateCSRResponse": {
      "additionalProperties": false,
      "description": "The response body for the GenerateCSR action.",
      "properties": {
         "CSRString": {
            "description": "The string for the certificate signing request.",
            "readonly": true,
            "type": "string"
         },
         "CertificateCollection": {
            "$ref": "http://redfish.dmtf.org/schemas/v1/CertificateCollection.json#/definitions/CertificateCollection",
            "description": "The link to the certificate resource collection where the certificate is installed.",
            "readonly": true
         }
      },
      "required": ["CertificateCollection", "CSRString"],
      "type": "object"
   }
```

```
}
```

In the previous example, the following payload is an example response for the `GenerateCSR` action.

```
{
   "CSRString": "-----BEGIN CERTIFICATE REQUEST-----...-----END CERTIFICATE REQUEST-----",
   "CertificateCollection": {
      "@odata.id": "/redfish/v1/Managers/BMC/NetworkProtocol/HTTPS/Certificates"
   }
}
```

13.2.6 JSON Schema terms

Table 31 describes the JSON Schema terms that Redfish uses to provide schema annotations for Redfish JSON Schema:

JSON Schema term	Related Redfish schema annotation
`description` `enumDescriptions`	Description
`longDescription` `enumLongDescriptions`	Long description
`additionalProperties`	Additional properties
`readonly`	Permissions
`required`	Required
`requiredOnCreate`	Required on create
`units`	Units of measure
`autoExpand`	Expanded resource

`deletable`
`insertable`
`updatable`
	Resource capabilities

`uris`	Resource URI patterns
`owningEntity`	Owning entity

JSON Schema term	Related Redfish schema annotation
`deprecated` `versionDeprecated`	Deprecated

: **Table 31 — JSON Schema terms**

13.3 OpenAPI

13.3.1 OpenAPI overview

The OpenAPI Specification defines a format for describing JSON payloads and the set of URIs a client can access on a service. The following clause describes how Redfish uses OpenAPI to describe resources and resource collections.

13.3.2 File naming conventions for OpenAPI schema

Each Redfish OpenAPI file represents a single resource type.

Versioned Redfish OpenAPI files shall be named using the resource type name for the schema, following the format:

`<ResourceType>.v<MajorVersion>_<MinorVersion>_<Errata>.yaml`

For example, version 1.3.0 of the Chassis schema is `Chassis.v1_3_0.yaml`.

Unversioned Redfish OpenAPI files shall use the resource type name to name the file, in this format:

`<ResourceType>.yaml`

For example, the unversioned definition of the Chassis schema is `Chassis.yaml`.

13.3.3 Core OpenAPI schema files

Table 32 describes the core OpenAPI schema files:

File	Description
`odata-v4.yaml`	Definitions for common OData properties.
`openapi.yaml`	URI paths and their respective payload structures.
`Resource.yaml` and its subsequent versions	All base definitions for resources, resource collections, and common properties, such as `Status`.

: **Table 32 — Core OpenAPI schema files**

13.3.4 openapi.yaml

The `openapi.yaml` file is the starting point for clients to understand the construct of the service.

Table 33 describes the terms that the `openapi.yaml` file contains:

Term	Description
`components`	Global definitions. For Redfish, contains the format of the Redfish error response.
`info`	Structure consisting of information about what the `openapi.yaml` is describing, such as the author of the file and any contact information.
`openapi`	Version of OpenAPI the document follows.
`paths`	URIs supported by the document, with possible methods, response bodies, and request bodies.

: **Table 33 — openapi.yaml terms**

The service shall return the `openapi.yaml` file, 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-defined 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
        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
```

13.3.5 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. Table 34 describes the terms that the YAML object contains:

Term	Description
`components`	Structures, enumerations, and other definitions defined by the schema.
`x-copyright`	Copyright statement for the organization producing the OpenAPI file.
`title`	For a schema file that describes a resource or resource collection, the matching type identifier for the resource or resource collection.

: **Table 34 — YAML object terms**

13.3.6 OpenAPI components body

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

13.3.6.1 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 definitions:

- 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.

13.3.6.2 Enumerations in OpenAPI

Table 35 describes the terms in OpenAPI enumerations:

Term	Description
`enum`	String array that contains the possible enumeration values.
`type`	Because all enumerations in Redfish are strings, the `type` term always has the value `string`.
`x-enumDescriptions`	Object that contains the descriptions for each of the enumerations as name-value pairs.
`x-enumLongDescriptions`	Object that contains the long descriptions for each enumeration as a name-value pair.
`x-enumDeprecated`	Object that contains the deprecation guidance for each of the enumerations as name-value pairs.
`x-enumVersionDeprecated`	Object that contains the deprecation version information for each of the enumerations as name-value pairs.

: **Table 35 — OpenAPI enumerations**

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.
```

13.3.6.3 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:

`#<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 describes the action itself. This definition contains property definitions for the `target` and `title` properties shown in response payloads for the resource.

The following example shows a definition of an action:

```
Reset:
  additionalProperties: false
  description: This action resets 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 reset 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.

The following example shows a definition of parameters of an action:

```
ResetRequestBody:
  additionalProperties: false
  description: This action resets the system.
  properties:
```

```
    ResetType:
      $ref: http://redfish.dmtf.org/schemas/v1/Resource.yaml#/components/schemas/ResetType
      description: The reset type.
      x-longDescription: This parameter shall define the type of reset to perform.
  type: object
  x-longDescription: This action shall reset the ComputerSystem.
```

13.3.6.4 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 `#ContosoNetworkDevice.Ping` action, assuming it's found in the versioned ContosoNetworkDevice OpenAPI file with a name, such as `ContosoNetworkDevice.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
```

13.3.7 OpenAPI terms used by Redfish

Table 36 describes the OpenAPI terms that Redfish uses to provide schema annotations for Redfish OpenAPI files:

OpenAPI term	Related Redfish schema annotation
`description` `x-enumDescriptions`	Description
`x-longDescription` `x-enumLongDescriptions`	Long description
`additionalProperties`	Additional properties
`readOnly`	Permissions

OpenAPI term	Related Redfish schema annotation
`required`	Required
`x-requiredOnCreate`	Required on create
`x-units`	Units of measure
`x-autoExpand`	Expanded resource
`x-owningEntity`	Owning entity

`deprecated`
`x-deprecatedReason`
`x-versionDeprecated`
	Deprecated

: **Table 36 — OpenAPI terms used by Redfish**

13.4 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 following rules. 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 a property by adding length annotations to properties that do not have those annotations.
- Modified schema may constrain a property by adding a pattern annotation to properties that do not have that annotation.
- Modified schema may constrain the capabilities of a resource or resource collection to remove support for HTTP operations.
  - Modified schema may change the update capabilities to indicate a client can perform a `PATCH` or `PUT` request on the resource to support writable OEM properties.
- Modified schema may remove properties that are not required.
- Modified schema may remove actions.
- Modified schema may remove action parameters that are not required.
- Modified schema may change description annotations.
- 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 to specify who made the modifications.
- Modified schema may remove URIs from the resource URI patterns annotation.
- Modified schema may add URIs to the resource URI patterns annotation to define OEM URIs for standard resources and shall follow the OEM URI rules specified by the OEM URIs clause.

- Other modifications to the schema shall not be allowed.

14 Service details

14.1 Eventing

14.1.1 Eventing overview

This clause describes how to use the REST-based mechanism to subscribe to and receive event messages.

> **Note:** For security implications of eventing, see the Security details clause.

The Redfish service requires a client or administrator to create *subscriptions* to receive events.

To create a subscription, use one of these methods:

- Directly HTTP `POST` to the subscription collection.
- Indirectly open a server-sent events (SSE) connection for the event service.

14.1.2 POST to subscription collection

To locate the event service, the client traverses the Redfish service interface. The event service is located in the service root, as described in the `ServiceRoot` schema.

After the client discovers the service, they perform an HTTP `POST` on the resource collection URI for `Subscriptions` in the event service to subscribe to events. For the *subscription* body syntax, see the Redfish `EventDestination` schema. This request includes:

- The URI where an event-receiver client expects events to be sent. When an event is triggered within the Redfish service, the service sends an event to that URI.
- The type of events to send.

If the subscription request succeeds, the service shall return:

- An HTTP `201 Created` status code.
- The `Location` header that contains a URI of the newly created subscription resource.

If the subscription request succeeds, the service should return:

- A response body containing a representation of the subscription resource that conforms to the `EventDestination` schema.

After a subscription is registered with the service, clients begin receiving events. Clients do not receive events retroactively. The service does not retain historical events.

Services shall:

- Support *push* style eventing for all resources that can send events.
- 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, the service shall return the HTTP `400 Bad Request` status code.
- 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 the HTTP `400 Bad Request` status code. These properties are considered mutually exclusive.
- Retain subscriptions as persistent across service restarts.

Services shall not:

- *Push* events by using HTTP `POST` unless an event subscription has been created. To terminate the event stream at any time, either the client or the service can delete the subscription.
- Send a *push* event payload larger than 1 Mebibyte (MiB). If more than 1 MiB of data is to be sent, 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.

Services may:

- Terminate a subscription by sending a `SubscriptionTerminated` message from the Base Message Registry as the last event.
- Terminate a subscription if the number of delivery errors exceeds preconfigured thresholds.

To unsubscribe from the events associated with this subscription, the client or administrator shall perform an HTTP `DELETE` request to the subscription's resource URI.

Subsequent requests to subscription resources that have been terminated respond with the HTTP `404 Not Found` status code.

Some configurable properties define the behavior for all event subscriptions. For details, see the Redfish `EventService` schema.

14.1.3 Open an SSE connection

A service may support the `ServerSentEventUri` property in the `EventService` resource. If a client performs a `GET` request on the URI that the `ServerSentEventUri` contains, an SSE connection opens for the client. For details about this method, see the server-sent events Event service clause.

14.1.4 EventType-based eventing

> **DEPRECATED:** `EventType`-based eventing is deprecated in the Redfish schema in favor of using `RegistryPrefix` and `ResourceType`.

DEPRECATED

Table 37 describes the types of events that Redfish generates:

Event	Occurs when	Description
Life cycle	Resources are created, modified, or destroyed. Usually indicates that the resource and, optionally, its properties have changed.	Not every modification of a resource results in an event. This behavior is similar to when ETags are changed and implementations might not send an event for every resource change. For example, if an event is sent for every Ethernet packet that is received or each time that a sensor changes one degree, more events than fit in a scalable interface are generated.
Alert	An event of some significance happens. Depending on the resource, may be generated directly or indirectly.	Usually adopts a message registry approach similar to extended error handling in that a `MessageId` is included. An example of an alert event is, a chassis is opened, a button is pushed, a cable is unplugged, or a threshold exceeded. These events usually do not correspond well to life cycle-type events. Therefore, alerts have their own category.
Metric report	The telemetry service generates or updates a metric report.	Generated as specified by the `MetricReportDefinition` resources found subordinate to the telemetry service. Can occur periodically, on demand, or when changes are detected in the metric properties. For details, see the Redfish `MetricReportDefinition` schema.

: **Table 37 — EventType-based eventing**

END DEPRECATED

14.1.5 Subscribing to events

Table 38 describes the properties that a subscriber provides to subscribe to events and filter received messages:

Property	Description
`RegistryPrefixes`	An array of standard or OEM message registries. An event is sent to the subscriber if one of the message registries that `RegistryPrefixes` lists defines the event message. To receive messages from all registries, pass an empty array. The contents of the array does not include the registry version. For example, if the registry is `Base.1.5.0`, the property value is `Base`.
`ResourceTypes`	An array of standard or OEM resource types. An event is sent to the subscriber if the `OriginOfCondition` resource type matches one of the `ResourceTypes` values. The contents of the array does not include the schema version. For example, if the resource type is `Task.v1_2_0.Task`, the property value is `Task`. To receive messages from any resource, pass an empty array.
`OriginResources`	An array of URIs to resources. An event is sent to the subscriber if the `OriginOfCondition` property matches one of the URIs listed in `OriginResources`. To receive messages from any resource, pass an empty array. To include subordinate resources regardless of depth, set the `SubordinateResources` property to `true`.
`EventFormatType`	The format that can be sent by using the `EventFormatTypes` property in the event service. Represents the format of the payload sent to the event destination. If the subscriber omits this value, the payload corresponds to the `Event` schema.

: **Table 38 — Subscription properties**

14.1.6 Event formats

Table 39 describes the event formats:

Event format	Description
Metric report message objects	Used when the telemetry service generates a new or updates an existing metric report. Metric report message objects sent to the specified client endpoint shall contain the properties, as described in the Redfish `MetricReport` schema.

Event format	Description
Event message objects	Used for all other types of events. Event message objects `POST`ed to the specified client endpoint shall contain the properties as described in the Redfish `Event` schema. Supports a message registry. In a message registry approach, a message registry lists the `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. The registry also contains a message. The message itself can have arguments and default values for severity and recommended actions. The `MessageId` property follows the format defined in the `MessageId` format clause Event messages may also have an `EventGroupId` property, which lets 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 more. For the client to determine whether these have the same root cause, these messages have the same value for the `EventGroupId` property.

: **Table 39 — Event formats**

14.1.7 OEM extensions

OEMs can extend both messages and message registries. Any individual message, per the `MessageRegistry` schema definition, define OEM sections. Thus, if OEMs wish to provide additional information or properties, use 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.

14.2 Asynchronous operations

Services that support asynchronous operations implement the `TaskService` and `Task` resources.

The task service describes the service that handles *task*. It contains a resource collection of zero or more `Task` resources. The `Task` resource describes a long-running operation that is spawned when a request takes longer than a few seconds, such as when a service is instantiated.

The `Task` schema defines task structure, including the start time, end time, task state, task status, and zero or more task-associated messages.

Each task has a number of possible states. The `Task` schema defines the exact states and their semantics.

When a client issues a request for a long-running operation, the service returns the HTTP `202 Accepted` status code

and a `Location` header that contains the URI of the *task monitor* and, optionally, the `Retry-After` header that defines the amount of time that the client should wait before querying the status of the operation.

The task monitor is an opaque service-generated URI that the client who initiates the request can use. To query the status of an operation and determine when the operation has been completed and whether it succeeded, the client performs a `GET` request on the task monitor. The client should not include the `application/http` MIME type in the `Accept` header.

The `202 Accepted` response body should contain an instance of the `Task` resource that describes the state of the task.

As long as the operation is in process, the service shall continue to return the HTTP `202 Accepted` status code when the client queries the task monitor URI.

If a service supports cancellation of a task, the `Allow` header shall contain `DELETE` for the task monitor. To cancel the operation, the client may perform a `DELETE` request on the task monitor URI. The service determines when to delete the associated `Task` resource.

To cancel the operation, the client may also perform a `DELETE` request to the `Task` resource. Deleting the `Task` resource may invalidate the associated task monitor. A subsequent `GET` request on the task monitor URI returns either the HTTP `410 Gone` or `404 Not Found` status code.

In the unlikely event that a `DELETE` of the task monitor or `Task` resource returns the HTTP `202 Accepted` status code, an additional task shall not be started and instead the client may monitor the existing `Task` resource for the status of the cancellation request. When the task finally completes cancellation, operations to the task monitor and `Task` resources shall return the HTTP `404 Not Found` status code.

After the operation has been completed, the service shall update the `TaskState` with the appropriate value. The `Task` schema defines the task completed values.

After the operation has been completed, the task monitor shall return:

- The appropriate HTTP status code, such as but not limited to `200 OK` for most operations or `201 Created` for `POST` to create a resource.
- The headers and response body of the initial operation, as if it had completed synchronously.

If the initial operation fails, the response body shall contain an error response.

If the operation has been completed and the service has already deleted the task, the service may return the HTTP `410 Gone` or `404 Not Found` status code. This situation can occur if the client waits too long to read the task monitor.

To continue to get status information, the client can use the resource identifier from the `202 Accepted` response to directly query the `Task` resource.

- Services that support asynchronous operations shall implement the `Task` resource.
- The response to an asynchronous operation shall return the HTTP `202 Accepted` status code and set the `Location` response header to the URI of a task monitor associated with the task. The response may also include the `Retry-After` header that defines the amount of time that the client should wait before polling for status. The response body should contain a representation of the `Task` resource.
- `GET` requests to either the task monitor or `Task` resource shall return the current status of the operation without blocking.
- HTTP `GET`, `PUT`, and `PATCH` operations should always be synchronous.
- Clients shall be prepared to handle both synchronous and asynchronous responses for HTTP `GET`, `PUT`, `PATCH`, `POST`, and `DELETE` requests.
- 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 `@Redfish.OperationApplyTime` value is `AtMaintenanceWindowStart` or `InMaintenanceWindowOnReset`, the task should also include the `@Redfish.MaintenanceWindow` property.

14.3 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 resets or power cycles, and should withstand a reasonable amount of configuration change, such as adding an adapter to a server.

The resource tree on one service might not be consistent across instances of devices. The client should traverse the data model and discover resources to interact with them.

Some resources might remain very stable from system to system, such as manager network settings. However, the architecture does not guarantee this stability.

- 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 client shall not expect the resource tree to be consistent between instances of services.

14.4 Discovery

14.4.1 Discovery overview

Automatic discovery of managed devices supporting Redfish may be accomplished by using the Simple Service Discovery Protocol (SSDP). This protocol enables 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 enable the user to disable the protocol through the `ManagerNetworkProtocol` resource.

The objective of discovery is for client software to locate managed devices that conform to the *Redfish Specification*. Therefore, the primary SSDP functionality is incorporated in the M-SEARCH query. Redfish also follows the SSDP extensions and naming that UPnP uses, where applicable, so that systems that conform to the *Redfish Specification* can also implement UPnP without conflict.

14.4.2 UPnP compatibility

For compatibility with general-purpose SSDP client software, primarily UPnP, the service should use UDP port 1900 for all SSDP traffic. In addition, the Time-to-Live (TTL) hop count setting for SSDP multicast messages should default to `2`.

14.4.3 USN format

The UUID in the USN field of the service shall equal the UUID property in the service root. If multiple or redundant managers exist, 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`.

14.4.4 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 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 for Search Target (ST) of `upnp:rootdevice`.

The URN provided in the `ST` header in the reply shall use the `redfish-rest:` service name 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, that minor version shall be appended with and preceded by a colon (`:`).

For example, a service that conforms to a *Redfish Specification v1.4* would reply with a `redfish-rest:1:4` service.

The managed device shall provide clients with the `AL` header that points to the Redfish service root URL.

For UPnP compatibility, the managed device should provide clients with the `Location` header that points to the UPnP XML descriptor.

The response to an M-SEARCH multicast or unicast query shall use the following format:

```
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 the service>::urn:dmtf-org:service:redfish-rest:1
AL:<URL of Redfish service root>
EXT:
```

A service may provide additional headers for UPnP compatibility. Fields in brackets are placeholders for device-specific values.

14.4.5 Notify, alive, and shutdown messages

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

14.5 Server-sent events

14.5.1 General

Server-sent events (SSE), defined by the Web Hypertext Application Technology Working Group (WHATWG), enables a client to open a connection with a web service. The web service can continuously push data to the client, as needed.

Successful resource responses for SSE shall:

- Return the HTTP `200 OK` 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 `400` or greater status code.
- 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. Services shall separate events with blank lines. Blank lines should be sent as part of the end of an event, otherwise dispatch may be delayed in conforming consumers.

The following clauses describe how Redfish uses SSE in different Redfish data model contexts. For details about SSE, see the *HTML5 Specification*.

14.5.2 Event service

A service's implementation of the `EventService` resource may contain the `ServerSentEventUri` property. If a client performs a `GET` request on the URI specified by the `ServerSentEventUri` property, the service shall keep the connection open and conform to the HTML5 Specification until the client closes the socket. Service-generated events shall be sent to the client by using the open connection.

When a client opens an SSE stream for the event service, the service shall create an `EventDestination` resource in the `Subscriptions` collection for the event service to represent the connection. The `Context` property in the `EventDestination` resource shall be a service-generated opaque string.

The service shall delete the corresponding `EventDestination` resource when the connection is closed. The service shall close the connection if the corresponding `EventDestination` resource is deleted.

The service shall use the `id` field in the SSE stream to uniquely identify a payload in the SSE stream. The value of the `id` field is determined by the service. 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 based on the payload format. The SSE streams have these formats:

- Metric report SSE stream. Services shall use this format when the telemetry service generates or updates a metric report.
- Event message SSE stream. Services shall use this format 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 $filter query parameter 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 `EventService` 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)
  ```

14.5.2.1 Event message SSE stream

The service shall use the `data` field in the SSE stream to include the JSON representation of the `Event` object.

The following example payload shows a stream that contains a single event with the `id` field set to `1`, and a `data` field that contains a single `Event` object.

```
id: 1
data:{
data:    "@odata.type": "#Event.v1_6_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:            "MessageSeverity": "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:}
```

14.5.2.2 Metric report SSE stream

The service shall use the `data` field in the SSE stream to include the JSON representation of the `MetricReport` object.

The following example payload shows a stream that contains 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_3_0.MetricReport",
data:    "Id": "AvgPlatformPowerUsage",
data:    "Name": "Average Platform Power Usage metric report",
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:}
```

14.6 Update service

14.6.1 Overview

This clause covers the mechanism for software updates by using the update service.

14.6.2 Software update types

Clients can use these methods to update software through the update service:

- Simple updates: The service *pulls* the update from a client-indicated network location.
- Multipart HTTP push updates: The client uses HTTP or HTTPS with a multipart-formatted request body to *push* a software image to the service.

14.6.2.1 Simple updates

A service can support the `SimpleUpdate` action within the `UpdateService` resource. A client can perform a `POST` request on the action target URI to initiate a pull-based update, as defined 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*. Clients can use this *task* to monitor the progress and results of the update, which includes the progress of image transfer to the service.

14.6.2.2 Multipart HTTP push updates

A service may support the `MultipartHttpPushUri` property within the `UpdateService` resource. A client can perform an HTTP or HTTPS `POST` request 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 about `multipart/form-data` HTTP requests, see 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 Table 40. 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 Table 40. 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 the 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 `UpdateService` 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*. Clients can use this *task* to monitor the progress and results of the update.

Table 40 describes the requirements of a `multipart/form-data` request body for an 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	Media type format and character set of this request part.
Update file binary part	`Content-Disposition`	`form-data; name="UpdateFile"; filename=string`	Yes	Binary file to use for this software update. The 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	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.

: **Table 40 — Multipart HTTP push updates**

This example shows 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>
```

15 Security details

15.1 Transport Layer Security (TLS) protocol

15.1.1 Transport Layer Security (TLS) protocol overview

Implementations shall support the Transport Layer Security (TLS) protocol v1.2 with RFC7525 recommendations or later. Implementations may remove support for older versions for TLS in favor of newer versions.

> **DEPRECATED:** Previous versions of this specification allowed for TLS v1.1.

Implementations should support:

- The Storage Networking Industry Association (SNIA) TLS Specification for Storage Systems.
- The latest version of the TLS v1.x specification.

15.1.2 Cipher suites

Implementations shall only support cipher suites listed as "Recommended" in the **TLS Cipher Suites** table defined by the IANA TLS Parameters registry.

Cipher suites that are listed as mandatory in various RFCs, but are not "Recommended" in the **TLS Cipher Suites** table defined by the IANA TLS Parameters registry, shall not be supported.

Implementations should consider the support of pre-shared key ciphers suites listed as "Recommended" in the **TLS Cipher Suites** table defined by the IANA TLS Parameters registry, which enable authentication and identification without trusted certificates.

DEPRECATED

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

Redfish implementations should consider the support of ciphers, such as the following ciphers, which enable authentication and identification without 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
```

The advantage of these 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.

Additionally, Redfish implementations should support the following cipher:

```
TLS_RSA_WITH_AES_128_CBC_SHA
```

For more information, see RFC5487 and RFC5288.

END DEPRECATED

15.1.3 Certificates

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

Redfish implementations shall use certificates that conform to X.509-v3, as defined in RFC5280.

15.2 Sensitive data

Operations that contain sensitive data should use HTTPS only. For example, a `SimpleUpdate` action with a user name and password should use HTTPS to protect the sensitive data.

Properties in service responses that represent sensitive data, such as passwords, shall be `null`.

Responses from URIs that contain sensitive data may return the HTTP `404 Not Found` status code instead of the HTTP `401 Unauthorized` status code or the HTTP `403 Forbidden` status code to prevent attackers from obtaining the sensitive data in the URI.

15.3 Authentication

15.3.1 Authentication overview

Services:

- Shall support both HTTP Basic authentication and Redfish session login authentication.
- Shall use only connections that conform to TLS to transport the data between any third-party authentication service and clients.

- Shall not require a client that uses HTTP Basic authentication to create a session.
- May implement other authentication mechanisms.

15.3.2 Authentication requirements

15.3.2.1 Resource and operation authentication requirements

Services shall authenticate all write requests to Redfish resources. For example:

- `POST`, except to the `Sessions` resource collection for authentication
- `PUT`
- `PATCH`
- `DELETE`

Redfish resources shall not be available as unauthenticated, except for:

- The service root to identify the device and service locations.
- The Redfish metadata document to get resource types.
- The OData service document for compatibility with OData clients.
- The Redfish OpenAPI YAML document for compatibility with OpenAPI clients.
- The `version` object at `/redfish`.

> **Note:** This specification does not cover external services that are linked through external references. These services may have other security requirements.

15.3.2.2 HTTP header authentication requirements

An authentication header shall accompany every request that establishes a secure channel.

Services:

- Shall process HTTP headers for authentication before other headers that may affect the response. For example, `ETag`, `If-Modified`, and so on.
- Shall not use HTTP cookies to authenticate any activity, such as `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`.

15.3.2.3 Authentication failure requirements

When authentication fails, extended error messages shall not provide privileged information.

15.3.3 HTTP Basic authentication

Services shall support HTTP Basic authentication, as defined by RFC7617, and shall use only connections that conform to TLS to transport the data between any third-party authentication service and clients.

All requests that use HTTP Basic authentication shall require HTTPS.

> **Note:** The IETF has highlighted security concerns with HTTP Basic authentication. While HTTPS is required for the usage of HTTP Basic authentication, there are other concerns implementors need to be aware of that are documented in RFC7617.

15.3.4 Redfish session login authentication

Service shall provide login sessions that conform with this specification.

Session management is determined by the implementation of the Redfish service, which includes orphaned session timeout and the management of the number of simultaneous open sessions.

15.3.4.1 Redfish login sessions

For improved performance and security, a client should use the session management interface to create a Redfish login session. The session service specifies the URI for session management.

To establish a session, find the URI in either:

- The session service's `Sessions` property.
- 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"
      }
   },
   ...
}
```

15.3.4.2 Session login

To create a Redfish session without an authentication header, perform an HTTP `POST` request on the session service's `Sessions` resource collection. The `POST` to create a session shall only be supported with HTTPS. If both HTTP and HTTPS are enabled, a `POST` request to create a session through the HTTP port should redirect to the HTTPS port. Include 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>"
}
```

Fields in brackets are placeholders for client-specific values.

To verify that the request has been initiated from an authorized client domain, services should save the `Origin` header in reference to this session creation and compare it to subsequent requests using this session.

The response to the `POST` request to create a session shall include:

- `X-Auth-Token` header. Contains a session authentication token that the client can use in subsequent requests.
- `Location` header. Contains a hyperlink to the new `Session` resource.
- JSON response body. Contains the full representation of the new `Session` resource.

The following sample response shows a newly created session:

```
HTTP/1.1 201 Created
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>",
```

```
   "Password": null
}
```

The client that sends the session login request should save the session authentication token from the `X-Auth-Token` header and the contents of the `Location` header from the response of the login `POST` request.

To authenticate subsequent requests, the client sets the `X-Auth-Token` header to the session authentication token that the `POST` login request returns.

> **Note:** The session ID differs from the session authentication token, as follows:
>
> - **Session ID**: The session ID uniquely identifies the `Session` resource. The response data with the last segment of the `Location` header URI returns is the session ID. To view active sessions and terminate any session, an administrator with sufficient privileges can use the session ID.
> - **Session authentication token**: Only the client that executes the login has the session authentication token.

15.3.4.3 Session lifetime

Unlike some token-based methods that use token expiration times, Redfish sessions time out. As long as a client continues to send requests more frequently than the session timeout period, the session remains open and the session authentication token remains valid. If the session times out, it is automatically terminated.

15.3.4.4 Session termination or logout

When the client logs out, the Redfish session terminates. The session terminates through a `DELETE` request to the `Session` resource defined in either the `Location` header URI or the session ID in the response data.

This ability to `DELETE` a session through the `Session` resource enables an administrator with sufficient privileges 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.

15.4 Authorization

15.4.1 Authorization overview

The Redfish authorization subsystem controls which users have access to resources and the type of access that users have. It consists of two parts: the privilege model and the operation-to-privilege mapping.

The privilege model maps users to roles and maps roles to privileges. A privilege is a permission to complete an operation, such as read or write, within a defined management domain. For example the `ConfigureUsers` privilege allows adding a user. A user is authorized to access a resource if they have the privileges required for that resource. The operation-to-privilege mapping defines which privileges are required to access any given operation.

Redfish allows vendors to extend the standard privilege model with OEM privileges and custom OEM roles. OEM privileges and custom roles participate in the privilege model the same as Redfish standard privileges and roles. Services may also allow clients to create custom roles. Restricted roles and restricted privileges allow vendors to further refine their authority model.

Services shall enforce the same privilege model for ETag-related activity as is enforced for the data being represented by the ETag. For example, the privilege required to read an ETag shall be the same as the privilege to read the data item that the ETag represents.

15.4.2 Privilege model

Each user shall be assigned exactly one role with the `RoleId` property in the `ManagerAccount` resource. The value of the `RoleId` property identifies a `Role` resource in the `RoleCollection` resource, where a role defines a set of privileges. A role shall be assigned to a user when a manager account is created. The client shall provide the `RoleId` property when creating a manager account to select one of the standard or custom roles.

Services shall provide information about all roles through the `RoleCollection` resource. The `AssignedPrivileges` and `OemPrivileges` arrays in the `Role` resource define a set of assigned privileges for the associated role. Two roles with the same privileges shall behave equivalently.

15.4.2.1 Roles

Redfish defines a set of standard roles, allows a service to define custom OEM roles, and allows client-defined custom roles.

A service shall support all of the standard roles in Table 41. The value of the `Id` and `AssignedPrivileges` properties in the `Role` resource for the standard roles shall contain the **Role name** and **Assigned privileges** column values, respectively. The `AssignedPrivileges` property for standard roles shall not be modifiable. The `IsPredefined` property for standard roles shall contain the value `true`.

Table 41 describes the standard roles:

Role name	Assigned privileges
`Administrator`	`Login`, `ConfigureManager`, `ConfigureUsers`, `ConfigureComponents`, `ConfigureSelf`
`Operator`	`Login`, `ConfigureComponents`, `ConfigureSelf`
`ReadOnly`	`Login`, `ConfigureSelf`

: **Table 41 — Roles**

A service may define custom OEM roles. The `IsPredefined` property for OEM roles shall contain the value `true`. A service shall not allow users to modify predefined OEM roles. OEM role names should begin with a lowercase character or "Oem" followed by a vendor name to avoid conflict with future Redfish predefined role names.

A service may allow custom client-defined roles to be created, modified, and deleted. If allowed, a user can perform a `POST` request on the `RoleCollection` resource to create a role, indicating privileges in the `AssignedPrivileges` and `OemPrivileges` properties in the `Role` resource. A service may restrict which privileges are allowed. The `IsPredefined` property for client-defined roles shall contain the value `false`. A service shall not allow a client-defined role to be deleted while it is in use, for example, when it is assigned to a local user or an LDAP `RemoteRoleMapping` property.

The value of the `RoleId` property shall be unique across all roles within the `RoleCollection` resource.

Non-Redfish services, such as those enabled by the `AccountTypes` property within the `ManagerAccount` resource, should map the Redfish `RoleId` to their permission system. For example, an SSH user with `Administrator` as the value of the `RoleId` property could map to "root" for the SSH service. However, the privileges specified by the `AssignedPrivileges` and `OemPrivileges` do not necessarily map to non-Redfish services.

15.4.2.2 Restricted roles and restricted privileges

Restricted roles and restricted privileges are intended to prevent privilege escalation. Restricted roles and restricted privileges are not less functional, but their usage is restricted to particular users. For example, to have a security administrator have privileges that the administrator does not have, you need to ensure the administrator cannot escalate to the security administrator role. An implementation can help achieve this by restricting the `Administrator` role and providing an alternate administrator role that lacks the security privilege.

A service may restrict any role. The `Restricted` property for restricted roles shall contain the value `true`. When a standard role is restricted, services shall provide the `AlternateRoleId` property to reference a non-restricted custom role intended for clients to use as an alternate. Services may predefine or create accounts that are configured with a restricted role.

Services shall not allow:

- A `RoleId` value for a restricted role to be specified when creating or modifying a `ManagerAccount` resource. This ensures administrators cannot create an account for themselves that has a restricted role.
- Modification of `ManagerAccount` resources with a `RoleId` property containing a value for a restricted role, with the exception of the `Enabled` property. This ensures administrators cannot gain access to another account.
- Deletion of `ManagerAccount` resources with a `RoleId` property containing a value for a restricted role.
- A restricted role to be specified in the `LocalRole` property within the `RemoteRoleMapping` property within the `AccountService` and `ExternalAccountProvider` resources.

A service may restrict any privilege, including standard and OEM privileges. The `RestrictedPrivileges` and `RestrictedOemPrivileges` properties in the `AccountService` resource shall specify the restricted privileges. Services shall not allow custom roles to specify restricted privileges. Services may contain predefined roles that are configured with restricted privileges.

15.4.2.3 OEM privileges

OEM privileges allow a service to extend the privilege model by adding additional privileges to have additional control of what operations are allowed. It can be used when a standard privilege is overly broad.

A service may define OEM privileges and may include OEM privileges in any predefined role, including standard and custom OEM roles. The `OemPrivileges` property within the `Role` resource shall contain the OEM privileges that are assigned to the role. The `OemPrivileges` property in the `Role` resource for the predefined roles shall not be modifiable.

A service may allow OEM privileges to be assigned to client-defined roles.

15.4.3 Redfish service operation-to-privilege mapping

For every request that a client makes to a service, the service shall determine that the authenticated identity of the requester has the authorization to complete the requested operation on the resource 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 typically checks an HTTP request against a mapping of the authenticated requesting identity role and privileges to determine whether the identity privileges are sufficient to complete the operation in the request.

15.4.3.1 Why specify operation-to-privilege mapping?

Initial versions of the *Redfish Specifications* defined several role-to-privilege mappings for standardized roles and normatively identified several privilege labels but did not normatively detail what these privileges or how privilege-to-operations mappings could be specified or represented in a normative fashion.

The lack of a methodology to define which privileges are required to complete a requested operation against the URI in the request puts at risk the interoperability between service implementations that 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 from normatively defining privilege requirements for a service.

15.4.3.2 Representing operation-to-privilege mappings

A service should provide a Privilege Registry in the registry collection. This registry represents the privileges required to complete HTTP operations against resources supported by the service.

The Privilege Registry is a JSON document that contains a `Mappings` array of where an individual entry exists for every resource type that the service supports.

The operation-to-privilege mapping is defined for every resource type and applies to every resource the service implements for the applicable resource type.

In several situations, specific resources or properties may have differing operation-to-privilege mappings than the resource type-level mappings. In these cases, the resource type-level mappings need to be overridden. The `PrivilegeRegistry` schema defines the methodology for resource type-level operation-to-privilege mappings and related overrides.

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

15.4.3.3 Operation map syntax

An operation map defines the set of privileges required to complete an operation on a resource-type.

The mapped operations are `GET`, `PUT`, `PATCH`, `POST`, `DELETE`, and `HEAD`. A privilege mapping is defined for each operation, irrespective of whether the service or data model supports the operation on the resource-type.

The privilege labels may be the Redfish standardized labels that the `PrivilegeType` enumeration in the `Privileges` schema defines and they may be OEM-defined privilege labels. The required privileges for an operation are specified using logical AND and OR behavior. For more information, see the Privilege AND and OR syntax clause.

The following example defines the privileges required for various operations on the `Manager` resource. Unless the implementation defines mapping overrides to the `OperationMap` array, the specified operation-to-privilege mapping represents 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"]
      }]
   }
}
```

15.4.3.4 Mapping overrides syntax

Table 42 describes the operation-to-privilege mapping, which varies from the resource type-level mapping:

Situation	Description
Property override	Property has different privilege requirements than the resource in which it resides. For example, the `Password` property in the `ManagerAccount` resource requires the `ConfigureSelf` or `ConfigureUsers` privilege to change, in contrast to the `ConfigureUsers` privilege required for the other properties in `ManagerAccount` resources. If multiple properties with the same name are present in a resource, the property override applies to all property instances.
Subordinate override	Resource is used in context of another resource and the contextual privileges need to govern. For example, the privileges for `PATCH` operations on `EthernetInterface` resources depend on whether the resource is subordinate to the `Manager` resource, where `ConfigureManager` is required, or the `ComputerSystem` resource, where `ConfigureComponents` is required.
Resource URI override	Resource instance has different privilege requirements for an operation than those defined for the resource type.

: **Table 42 — Mapping overrides syntax**

The overrides are defined in the context of the operation-to-privilege mapping for a resource type.

If multiple overrides are specified for a single resource type, the following precedence should be used for determining the appropriate override to apply:

- Property override
- Resource URI override
- Subordinate override

15.4.3.5 Property override example

In the following example, the `Password` property on the `ManagerAccount` resource requires the `ConfigureSelf` or `ConfigureUsers` privilege to change, in contrast to the `ConfigureUsers` privilege required for the other properties in `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": {
         "PATCH": [{
            "Privilege": ["ConfigureUsers"]
         }, {
            "Privilege": ["ConfigureSelf"]
         }]
      }
   }]
}
```

15.4.3.6 Subordinate override

The `Targets` property in `SubordinateOverrides` lists a hierarchical representation for when to apply the override. In the following example, the override for an `EthernetInterface` resource is applied when it is subordinate to an `EthernetInterfaceCollection` resource, which in turn is subordinate to a `Manager` resource. If a client were to `PATCH` an `EthernetInterface` resource that matches this override condition, it requires the `ConfigureManager` privilege. Otherwise, the client requires 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": {
         "PATCH": [{
            "Privilege": ["ConfigureManager"]
         }],
         "POST": [{
            "Privilege": ["ConfigureManager"]
         }],
         "PUT": [{
            "Privilege": ["ConfigureManager"]
         }],
         "DELETE": [{
            "Privilege": ["ConfigureManager"]
         }]
      }
   }]
}
```

15.4.3.7 Resource URI override

The following example demonstrates the resource URI override syntax to define operation privilege variations for resource URIs.

The example defines both `ConfigureComponents` and `OEMAdminPriv` privileges as required to make 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"]
         }]
      }
   }]
}
```

15.4.3.8 Privilege AND and OR syntax

The array placement of the privilege labels in the `OperationMap` `GET`, `HEAD`, `PATCH`, `POST`, `PUT`, and `DELETE` operation element arrays define the logical combinations of privileges that are required to call an operation on a resource or property.

For OR logical combinations, the privilege label appears in the operation element array as individual elements.

The following example defines either `Login` or `OEMPrivilege1` privileges that are required to perform a `GET` request.

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

For logical AND combinations, the privilege label appears in the `Privilege` property array in the operation element.

The following example defines both `ConfigureComponents` and `OEMSysAdminPriv` that are required to perform a `PATCH` request.

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

15.5 Account service

15.5.1 Account service overview

- Implementations should store user passwords 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 atomic operations. Implementations may reject requests that do not include an ETag.
- When authentication fails, extended error messages shall not provide privileged information.

15.5.2 Password management

A Redfish service provides local user accounts through a collection of `ManagerAccount` resources located under the account service. The `ManagerAccount` resources enable 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.

15.5.3 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 by 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.
- A `GET` operation on the `ManagerAccount` resource associated with the account.
- 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 the HTTP `403 Forbidden` status code and include a `@Message.ExtendedInfo` object that contains the `PasswordChangeRequired` message from the Base Message Registry.

15.6 Asynchronous tasks

Irrespective of which user or privileged context starts a *task*, the information in the task object shall enforce the privileges required to access that object.

15.7 Event subscriptions

Before pushing event data object to the destination, the service may verify the destination for identity purposes.

16 Redfish Host Interface

The *Redfish Host Interface Specification* defines how software that runs on a host computer system can interface with a Redfish service that manages the host. For details, see DSP0270.

17 Redfish composability

A service may implement the `CompositionService` resource off of `ServiceRoot` to bind resources. One example is disaggregated hardware, which allows for independent components, such as processors, memory, I/O controllers, and drives, to be bound to create logical constructs that operate together. This enables a client to dynamically assign resources for an application.

A service that supports composability shall implement resource blocks, defined by the `ResourceBlock` schema, and resource zones, defined in the `Zone` schema, for the composition service. Resource blocks provide an inventory of components available to the client for building compositions. Resource zones describe the binding restrictions of the resource blocks that the service manages.

The resource zones within the composition service shall include the collection capabilities annotation in responses. The collection capabilities annotation allows a client to discover which resource collections in the service support compositions, the different composition request types allowed, how the `POST` request for the resource collection is formatted, and which properties are required.

A service that supports composablity and client multi-tenancy shall:

- Implement the `FreePool` and `ActivePool` properties in the `CompositionService` resource.
- Implement the `CompositionReservations` property in the `CompositionService` resource.
- Filter `GET` requests for the `ResourceBlocks`, `FreePool`, `ActivePool`, `ResourceZones`, and `CompositionReservations` resource collections where the value of the `Client` property in the `ResourceBlock` resource or `CompositionReservation` resource matches the client identity.
- Ensure the resources in composition requests are assigned to the client specified by the `Client` property in the `ResourceBlock` resource or `CompositionReservation` resource.
- Not filter any HTTP operations within the composition service for clients that contain the privilege `ConfigureCompositionInfrastructure` unless specified by query parameters.
- Move resource blocks between the `FreePool` and `ActivePool` resource collections based on the outcome of composition requests.
  - A resource block is moved to the `FreePool` resource collection when it is not contributing to any composed resources.
  - A resource block is moved to the `ActivePool` resource collection when it is contributing to one or more composed resources.

17.1 Composition requests

17.1.1 Composition requests overview

A service that implements the composition service, 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 the removal of a composed resource shall support the `DELETE` method on the composed resource.

A service may implement the `Compose` action in the `CompositionService` resource for the above composition requests.

17.1.2 Specific composition

A specific composition is when a client identifies 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 resource blocks. The schema for the resource being composed defines where the resource blocks are specified in the request.

The following example shows a `ComputerSystem` being composed with a specific composition request:

```
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"
      }]
```

```
   }
}
```

17.1.3 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.

The following constrained composition request composes 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"
         }]
      }]
   }
}
```

17.1.4 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 composed resource, a client can only add or remove resources from the composed resource. A service that supports expandable resources shall support one or more of the update methods that the Updating a composed resource clause describes.

17.2 Updating a composed resource

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

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

18 Aggregation

Aggregation has been a Redfish concept since its inception. Redfish uses collection for services that can represent more than one system. As the scale of Redfish implementations increase, clients want to operate on Redfish resources in bulk.

Aggregation is the representation of Redfish resources from a variety of sources so that they can be managed, in whole or in part, by a Redfish client. Membership can be heterogeneous and arbitrary, but it is expected that most aggregate members are the same resource type, such as an aggregate of `ComputerSystem` resource, which is represented by an `Aggregate` resource where members of its `Elements` array are exclusively of type `ComputerSystem`. The Redfish service proxies on behalf of the aggregated components to provide common operations. The Redfish service is representing resources on behalf of the components and incoming operations must be tracked by the Redfish service before being accomplished by communicating with the individual resources. Thus, aggregation also allows a Redfish client to act on resources as a group using aggregates.

18.1 Classes of aggregators

18.1.1 Implicit and complex aggregators

There are at least two classes of Redfish aggregators:

- Implicit aggregators. An example of an implicit aggregator is an enclosure manager, such as a manager of blades in an enclosure. This implementation has `ComputerSystem` resources representing blades in the `ComputerSystemCollection` resource, and one or more `Manager` resources in the `ManagerCollection` resource. It also would likely have a `Chassis` resource for each blade and a `Chassis` resource for the enclosure, which would use the `Contains` property in `Links` to express the containment relationship to the individual blades. This class of aggregator has tight coupling with system design, and proxies requests to and from the blades to perform management functions.
- Complex aggregators. An example of a complex aggregator is a rack-level manager, fabric manager, or a manager of similar scale, especially if it represents resources that it gathers through the proxy of information from other managers, like BMCs. The sources that this manager aggregates are more complex in nature and potentially varying. This manager probably has an interface to the resources and proxies the Redfish service on behalf of each set of resources. At this scale, a Redfish client would prefer to provide common functions, such as resetting a set of systems, to the Redfish service as a whole rather than invoking actions individually to achieve scalability requirements. This class of service also may need assistance in adding members to the service, such as providing address and account information for the aggregator to contact the components and initiate the proxy of Redfish operations.

18.1.2 Use cases

Several use cases make explicit aggregator representation necessary. What they have in common is the need for common functions for scalability. There are several classes of these common functions.

One use case is service-type functions. An example is a firmware update on a large number of systems. Rather than invoke actions on individual resources, it is more efficient for a client to specify to which resources to apply the image. In this case, a service already exists in the model so an aggregation service is not needed. Instead the existing service must be augmented to enable the application of an image to a list of resources.

Another use case is common actions. Examples are the `Reset` or `SetDefaultBootOrder` actions. These actions are defined in the `ComputerSystem` schema, but the Redfish URI structure requires that the action occur on each `ComputerSystem` resource. Thus, an individual operation applies to each resource. It is more efficient for a client to send one action with the list of the resources to which to apply the action. For example, to reset one thousand systems, sending one thousand individual reset operations requires significant overhead as compared to sending a single operation with a list of one thousand systems to reset.

A final use case is changing an attribute on multiple members of a collection. An example is changing the boot order on a large number of systems. This use case requires one operation per system. However, assuming the resources are in the same collection, the deep PATCH operation meets the requirements of this use case.

18.2 Aggregation service

18.2.1 Aggregation service overview

The `AggregationService` resource represents the Redfish aggregation service, which provides aggregation functions.

The aggregation service contains the group actions that can apply to groups of resources. The `AggregationService` schema defines the common actions that a client can take on groups of resources. These actions take an array of resource URIs as one of the parameters to which the action applies. If all members of the resource array do not support the method, a `4xx` status code shall be returned and the body shall contain an error response. If at least one member of the resource array successfully completed the action but others did not, the status code should be `200 OK` with `@Message.ExtendedInfo` objects for the failed members.

The aggregation service also contains `Aggregate`, `AggregationSource`, and `ConnectionMethod` resources.

18.2.2 Aggregator requirements

By implementing the `AggregationService` resource and including an `AggregationSourceCollection` resource, a complex aggregator shall meet the following requirements:

- Proxy to the aggregated resources on behalf of the service.
- Provide error and state propagation, such as health roll-up, when needed to provide such data to the parent resource.
- Combine resource collections from the aggregated resources.
  - For example, `ComputerSystem` resources that were gathered through proxy shall be in one `ComputerSystemCollection` resource.
  - Services shall complete a *URI fix-up* for all aggregated resources because every system cannot be at `/redfish/v1/Systems/1`.
  - It is advisable for Redfish implementations to use unique values for the `Id` properties. For example, base the `Id` property of a `ComputerSystem` resource on something unique like a UUID or serial number, or the manufacturer MAC address for network adapters, or WWN for Fibre Channel controllers.
- Unify other services.
  - The aggregation implementation hosts only one event service. The implementation shall combine all events into one stream. The implementation also hosts only one sessions service, telemetry service, update service, and other services. Thus the aggregator represents unification of Redfish services with which it communicates and proxies on the client's behalf to the providers of those services and information.

18.2.3 Aggregates

The `Aggregate` resource is the grouping mechanism that clients use to indicate to the service that this group of resources can be treated the same for certain functions, such as the actions. Each aggregate contains the list of individual resources that are to be treated as a single unit for operations. For example, if a client wishes to express that a subset of the `ComputerSystemCollection` resource be treated as a single unit for certain operations like reset, reset boot order, or firmware update, it can express the aggregate as the target URI for the operation.

The `Aggregate` schema defines the common actions that a client can make on an aggregate. The `Aggregate` resource contains an `Elements` array that specifies the members of the aggregate. Actions that are supported on an aggregate but not supported on all `Elements`, such as a `Reset` action that is not supported on an individual member of the `Elements` array, are not silently skipped. If all members of the `Elements` array do not support the method, a `4xx` status code shall be returned and the body shall contain an error response. If at least one member of the `Elements` array successfully completed the action, but others did not, the status code should be `200 OK` with `@Message.ExtendedInfo` objects for the failed members.

18.2.4 Aggregation sources and connection methods

The aggregation service model also includes a definition for the information used to access the resources being represented by the aggregator. Two collections of resources are used to represent this. These are the `AggregationSource` and `ConnectionMethod` resources.

The `AggregationSource` resource represents the source of information for the resources being reflected by the aggregator. It typically represents a lower layer service provided by another manager. It contains information needed

to access that source, such as the address and account information. It also has a reference to the `ConnectionMethod` resource used to access it.

The `ConnectionMethod` resource represents the protocol and other semantics required to communicate with the resources being aggregated. Examples of connection methods are Redfish, IPMI, and proprietary access methods. For methods such as IPMI, it's also possible to specify the variations and nuances from multiple vendors.

19 ANNEX A (informative) Change log

Version	Date	Description
1.13.0	2021-04-08	Added client multi-tenancy behavior to the **Redfish composability** clause. This adds free pool, active pool, and composition reservation constructs to Redfish composability.
		Added `Compose` action as a method of performing composition requests to the **Redfish composability** clause.
1.12.1	2021-04-08	International Organization for Standardization (ISO) updates:
		Added paragraph numbering.
		Added **Foreword** to the table of contents as an unnumbered heading, and placed **Acknowledgments** inside **Foreword**.
		Made **Scope** a level-1 clause.
		Normative references: Removed unused normative references and moved some references into **Bibliography**. The **Bibliography** lists, for information, those documents which are cited informatively in the document, as well as other information resources.
		Changed **Abstract** to **Introduction**.
		Corrected level-1 clauses to remove hanging paragraphs and to correct the occurrence of the single **Use cases** and **Aggregator requirements** sub-clauses.
		Terms, definitions, symbols, and abbreviated terms: • Combined **Symbols and abbreviated terms** clause with **Terms and definitions** clause into **Terms, definitions, symbols, and abbreviated terms** clause. • Formatted the clause correctly. • Added the **Hardware terms**, **Web development terms**, and **Redfish terms** sub-clauses to this clause. • Removed may, shall, and should from definitions. • Removed these terms: *managed system*, *Redfish event receiver*, and *Redfish provider*. • Corrected definitions so none begin with an article.
		Changed *may* to *can* or *might* where appropriate.
		Changed one *must* to *shall*.
		Added numbered captions to tables and changed occurrences of *the following table* to use precise references to the table numbers.
		Fixed broken cross-references.
		Corrected URIs in the deep `PATCH` example.
		Fixed several query parameter examples where string values were not properly wrapped with single quotes.

Version	Date	Description
		Corrected `Accept-Encoding` usage to allow for encoded responses if the client does not provide the header to align with RFC7231.
		Clarified usage of `DELETE` for the `@Redfish.OperationApplyTimeSupport` term.
		Removed duplicative clauses for HTTP `405 Method Not Allowed` usage in **PATCH (update)** in favor of more general clauses.
		Replaced exception table in **PATCH (update)** in favor of text.
		Moved error cases from response table in **POST (action)** to be with other text that describes error cases.
		Added linkage in the description for HTTP `201 Created` to reference response bodies for actions.
		Added informative text regarding the usage of `If-Match` and `If-Match-None` headers in `GET`, `PATCH`, and `PUT` clauses.
		Clarified the behavior of `$select` when an object property is selected.
		Added introductory text to guide readers to other Redfish documents.
		Clarified the ordering of processing query parameters.
		Clarified that update restrictions for a resource can be modified to support writable OEM properties.
		Clarified the **Settings resource** clause to show behavior of properties in the active resource and settings resource based on the service's capabilities.
		Corrected behavior for usage of `null` based on the configuration of a resource and other special situations.
		Clarified OEM naming rules for all OEM definitions to ensure names don't collide.
		Removed the term "namespace" from all non-CSDL related clauses and replaced them with references to a new *resource type* term.
1.12.0	2020-12-01	Added introductory text to the **Authorization** clause.
		Clarified usage of `RoleId` and how there are standard roles, custom OEM roles, and client-defined custom roles.
		Added **Restricted roles and restricted privileges** to describe behavior for when roles and privileges are marked as restricted.
1.11.2	2020-12-01	Clarified that the `Accept-Encoding` header is used to request compression of response bodies.
		Corrected the **PATCH (update)**, **PUT (replace)**, and **DELETE (delete)** clauses to leverage all normative statements for successful operations found in the **Modification success responses** clause.
		Replaced RFC5988 reference with RFC8288.
		Updated IETF links to use the "IETF Tools" site.
		Clarified that insert capabilities is just for resource creation.

Version	Date	Description
		Fixed ETag examples to be RFC7234-conformant.
		Clarified that OEM resources can have subordinate resources.
		Replaced RFC4627 reference with RFC8259.
		Replaced conflicting statements found in "HTTP redirect authentication requirements" with general clause for enforcing authentication and authorization at the target resource.
		Clarified behavior of `@odata.count` when a collection is filtered.
		Created standalone "MessageId format" clause.
		Removed duplicative text found in the event format table and referenced the message object clauses as needed.
		Corrected the response body specified for a `PATCH` operation containing read-only properties.
		Added informative text in the intro to the **Data model** clause describing the methods for OEM extensions.
		Clarified that sensitive data in URIs can be hidden from unauthorized users by returning HTTP `404 Not Found`.
		Added embedded links to the `Location` header entry in the response header table.
		Corrected `$select` example in the **The $select query parameter** clause.
		Corrected several embedded links to direct to the correct clause.
1.11.1	2020-08-04	Added missing clause requiring sensitive data to be returned as `null`.
		Clarified that `Resolution`, `Severity`, and `MessageSeverity` in responses can be service-defined and not come from a message registry.
		Relaxed schema rules to require description, long description, URI, and capabilities annotations only for schemas published or republished by the DMTF.
		Added clauses to **Schema modification rules** to allow for properties, actions, parameters, and URIs to be removed, descriptions to be modified, and pattern and length annotations to be added if not specified.
		Relaxed rule for the OData metadata document to not require, but only recommend that all referenced namespaces are included in the document.
		Added clause to clarify the usage of empty strings.
		Clarified behavior of `$skip` when the value is greater than or equal to the number of members in a resource collection.
		Corrected the minimum value for `$top` to align with OData.
		Clarified behavior of `PATCH` for partial success scenarios.
		Various clarifications and style fixes to the **Aggregation** clause.

Version	Date	Description
		Clarified that `HEAD` requests shall be rejected when a query parameter is provided.
		Removed erroneous requirement for ETags to be strong.
1.11.0	2020-04-30	Added **Aggregation** clause.
		Clarified that services are allowed use HTTP `501 Not Implemented` for unsupported HTTP methods.
		Clarified the normative semantics around the term "deprecated".
		Clarified clauses describing the usage of `null` for properties versus not reporting a property.
1.10.0	2020-03-27	Restructured the **Security details** clause for ease of reading. Other than the changes listed below, no other changes were intended. Any clarifications that inadvertently altered the normative behavior are considered errata, and will be corrected in future revisions to the specification.
		Deprecated TLS v1.1, and set the minimum TLS requirement to be TLS v1.2 with RFC7525 recommendations.
		Deprecated existing cipher suites clause in favor of new clause to leverage IANA recommendations.
		Added requirement for supporting the `/redfish` URI.
		Added support for deep operations.
1.9.1	2020-03-27	Deprecated full ISO8601 duration format in favor of a simplified version that does not contain years, months, and weeks.
		Added missing normative language for how actions with response bodies are defined in schema.
		Added HTTP `201 Created` as valid responses for actions.
		Clarified the `~` operator for the `$expand` query parameter to expand hyperlinks found in all `Links` properties.
		Clarified the `*` and `.` operators for the `$expand` query parameter to expand hyperlinks found in payload annotations, such as `@Redfish.Settings`.
		Clarified usage of action parameters that point to resources; the expectation is a reference object pointing to the resource in question is passed by the client.
		Clarified that `DELETE` on a resource likely deletes subordinate resources.
		Clarified best practices for naming rules, in particular with regards to acronyms.
		Clarified behavior for when individual members of a resource collection cannot be returned as part of a `$expand` request.
		Clarified usage of `@Message.ExtendedInfo` in error responses and provided guidance for clients for handling error responses.
1.9.0	2019-12-06	Made change to no longer require the `Server` response header.
		Added clause to **Schema modification rules** to allow for the addition of OEM URIs to standard resources.

Version	Date	Description
		Loosened requirements on `@odata.type` within `Oem` to not require it in arrays where the type is used repeatedly.
1.8.1	2019-12-06	Made many changes for style consistency, grammar, and general clarity. Except for the following additions, 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.
		Clarified SSE with regards to requiring a blank line after each event.
		Clarified order of precedence for resolving multiple operation overrides within the Privilege Registry.
		Clarified cases for property overrides in the Privilege Registry where multiple objects in the same resource contain the same property name.
		Updated references for HTTP Basic authentication to use RFC7617 instead of RFC7235.
		Added `text/event-stream`, `application/yaml`, and `application/vnd.oai.openapi` usage to the `Accept` and `Content-Type` header table entries.
		Added clause that provides guidance on service behavior when `null` is a property value in `POST` (create) operations.
		Loosened requirements on SSE `id` based on client usage.
		Added documentation for settings, settings apply time, operation apply time, operation apply time support, maintenance window, collection capabilities, requested count, allow over-provisioning, zone affinity, supported certificates, and deprecated terms to the **Payload annotations** clause.
		Added clauses that document responses for actions with a response body defined in schema.
		Clarified the allowable values payload annotation to show it can be used for both properties and action parameters.
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 that describes requirements for the `SimpleUpdate` action and the `MultipartHttpPushUri` property.
1.7.1	2019-08-08	Added statements about the *owning entity* annotation term 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.
		Corrected the **Password management** clause to describe how a user can `GET` their respective account resources when a password change is required.
		Clarified that registries are not required to return `@odata.id`.

Version	Date	Description
		Clarified that services should use HTTP `400 Bad Request` for invalid query requests.
		Clarified that services should use HTTP `400 Bad Request` when the `only` query is being combined with other query parameters.
		Clarified that services should use HTTP `400 Bad Request` when query parameters are used on non-GET operations.
		Added clause about how to construct enumeration values.
		Clarified references to specific messages to also reference their Message Registry.
		Added 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.
		Added clause about how URI patterns are constructed.
		Added Excerpt term.
1.7.0	2019-05-16	Made many changes for style consistency, grammar, and general clarity. Except for the following additions, 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 normative statements about how to handle array properties and `PATCH` operations on arrays.
		Separated data model and schema language clauses.
		Added clauses that describe how JSON Schema and OpenAPI files are formatted.
		Added clause that describes the schema versioning methodology.
		Added 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 statement about SSE to 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.

Version	Date	Description
		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.
		Added clause around the usage of the HTTP `403 Forbidden` status code when an account requires a password change.
1.6.1	2018-12-13	Added clause about percent encoding being allowed for query parameters.
		Changed `$expand` example to use `SoftwareInventory` instead of `LogEntry`.
		Added clause about the use of a separator for multiple query parameters.
		Fixed `$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 `$expand` and not to be used with singular resources.
		Clarified the usage of the HTTP status codes with task monitors.
		Made various spelling and grammar fixes.
1.6.0	2018-08-23	Added methods of using `$filter` on the SSE URI for the event service.
		Added support for the OpenAPI Specification v3.0. This allows OpenAPI-conforming software to access Redfish service implementations.

Version	Date	Description
		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 telemetry service.
		Added registry, resource, origin, or `EventFormatType` -based event subscription methods as detailed in the Specification and schema. Added an `EventFormatType` to enable 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.
		Changed the requiredment for the `@odata.context` annotation to be optional.
		Removed requirement for clients to include the `OData-Version` HTTP header in all requests.
1.5.1	2018-08-10	Added clarifications to required properties in structured properties derived from `ReferenceableMembers`.
		Reorganized **Eventing** clause 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.

Version	Date	Description
		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 entry for the HTTP `431 Request Header Fields Too Large` status code.
		Added statement that the HTTP `503 Service Unavailable` status code can be used during reboot or reset of a service to indicate that the service is temporarily unavailable.
		Clarified usage of the `@odata.type` annotation within embedded objects.
		Added statements about the required `Name`, `Id`, and `MemberId` properties, and the common `Description` property, which have always been shown as required in schema files, but which the Specification did not mention.
		Added guidance for the value of time-date properties when time is unknown.
		Added the `title` property description in actions.
		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 long description annotations.
		Corrected name of `ConfigureComponents` in the **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 `@Redfish.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*.
		Consistently used the term, *hyperlink*.
		Added example to clarify usage of `$select` query parameter with `$expand`, and clarified expected results when using `AutoExpand`. Corrected order of precedence for `$filter` parameter options.
		Corrected terminology for OEM-defined actions removing "custom" in favor of OEM, and clarified that the action `target` property is always required for an action, along with its usage.
		Corrected location header values for responses to data modification requests that create a task (`Task` resource vs. task monitor). Clarified error handling of `DELETE` operations on `Task` resources.

Version	Date	Description
		Removed references to obsolete and unused `Privilege` annotation namespace.
		Clarified usage of the `Base.1.0.GeneralError` message in the Base Message Registry.
		Added durable URIs for registries and profiles, and clarified intended usage for each folder in the repository. Added 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 enable 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 enable 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 to complete actions, and other `POST` , `PATCH` , or `PUT` requests.
		Added entries for the HTTP `428 Precondition Required` and `507 Insufficient Storage` status codes to clarify the proper response to certain error conditions. Added reference links to the HTTP status code table throughout.
		Updated the **Abstract** to reflect the current state of the specification.
		Added reference to RFC6585 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 `@odata.context` property value.
		Added clause to describe how to revise structured properties and 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 `@Redfish.SettingsApplyTime` annotation.
		Added Action example using the `ActionInfo` resource in addition to the simple `@Redfish.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 `428 Precondition Required` status code.
		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.

Version	Date	Description
		Clarified wording of the **Partial resource results** clause.
		Clarified behavior of a service when receiving a `PATCH` with an empty JSON object.
		Added statement about other uses of the HTTP `503 Service Unavailable` status code.
		Clarified format of URI fragments to conform to RFC6901.
		Clarified use of absolute and relative URIs.
		Clarified definition of the `target` property as originating from OData.
		Clarified distinction between *hyperlinks* and the *links property*.
		Corrected the JSON example of the privilege map.
		Clarified format of the `@odata.context` property.
		Added clauses about the schema file naming conventions.
		Clarified behavior of a service when receiving a `PUT` with missing properties.
		Clarified valid values in the `Accept` header to include wildcards per RFC7231.
		Corrected `ConfigureUser` privilege to be spelled `ConfigureUsers`.
		Corrected the **Session login** clause 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 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 enables a service to present a resource or even property-level mapping of HTTP operations to roles and privileges.
		Added references to the Redfish Host Interface Specification (DSP0270).
1.0.5	2016-12-09	Errata release. Various typographical errors.
		Corrected the use of *collection*, *resource collection*, and *members* throughout.
		Added glossary entries for *resource collection* and *members*.
		Corrected certificate requirements to reference definitions and requirements in RFC5280 and added a normative reference to RFC5280.

Version	Date	Description
		Clarified usage of the HTTP `POST` and `PATCH` operations.
		Clarified usage of the HTTP status codes and error responses.
1.0.4	2016-08-28	Errata release. Various typographical errors.
		Added example of an HTTP `Link` Header and clarified usage and content.
		Added the **Schema modification** clause, which describes the 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. Fixed the missing numbering in the table of contents and clauses. Corrected URL references to external specifications. Added missing normative references. Corrected typographical error in ETag example.
		Clarified examples for `@Message.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 and updated examples throughout the document to conform to this format. Specifically, `<namespace>.<n>.<n>.<n>` becomes `<namespace>.v<n>_<n>_<n>`. File naming rules for JSON Schema and CSDL (XML) schemas were also corrected to match this format and to enable future major (v2) versions to coexist.
		Added clause that details the location of the schema repository and lists the durable URLs for the repository.
		Added definition for the value of the Units annotation, using the definitions from the UCUM Specification. Updated examples throughout to use this standardized form.
		Modified the naming requirements for `Oem` property naming to avoid future use of colon `:` and period `.` in property names, which can produce invalid or problematic variable names when used in some programming languages or environments. Both separators have been replaced with underscore (`_`), with colon (`:`) and period (`.`) usage now deprecated (but valid).
		Removed duplicative or out-of-scope subclauses from the **Security** clause, which made unintended requirements on Redfish service implementations.
		Added the 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.

Version	Date	Description
		Clarified ETag header requirements.
		Clarified that no authentication is required for accessing the service root.
		Clarified description of retrieving resource 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 a table entry for the `Retry-After` header usage.
		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 resource collection response clause.
		Corrected name of normative resource `Members` property (`Members` , not `value`).
		Clarified that error responses may include information about multiple error conditions.
		Corrected name of `Measures.Unit` annotation term as used in examples.
		Corrected outdated reference to Core OData Specification in annotation term examples.
		Added the `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 the `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 long description 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.

20 Bibliography

- R. Fielding, 2000, *Architectural Styles and the Design of Network-based Software Architectures*, https://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm
- IETF RFC5288, J. Salowey et al, *AES Galois Counter Mode (GCM) Cipher Suites for TLS*, https://tools.ietf.org/html/rfc5288
- IETF RFC5487, M. Badra et al, *Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode*, https://tools.ietf.org/html/rfc5487
- IETF RFC5789, L. Dusseault et al, *PATCH Method for HTTP*, https://tools.ietf.org/html/rfc5789
- IETF RFC6906, E. Wilde, *The 'profile' Link Relation Type*, https://tools.ietf.org/html/rfc6906
- 28 October 1999, *Simple Service Discovery Protocol/1.0 Operating without an Arbiter*, https://tools.ietf.org/html/draft-cai-ssdp-v1-03
- 10 March 2016, *OData Version 4.0 Plus Errata 03: Core Vocabulary*, https://docs.oasis-open.org/odata/odata/v4.0/errata03/csd01/complete/vocabularies/Org.OData.Core.V1.xml
- 24 February 2014, *OData JSON Format Version 4.0*, https://docs.oasis-open.org/odata/odata-json-format/v4.0/os/odata-json-format-v4.0-os.html
- 24 February 2014, *OData Version 4.0 Part 2: URL Conventions*, https://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html