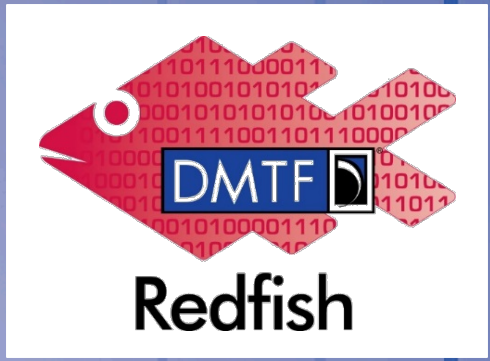
# Redfish Technical Overview

**Jeff Hilland**
**President, DMTF**
**DT Manageability, Hewlett Packard Enterprise**
**December, 2017**



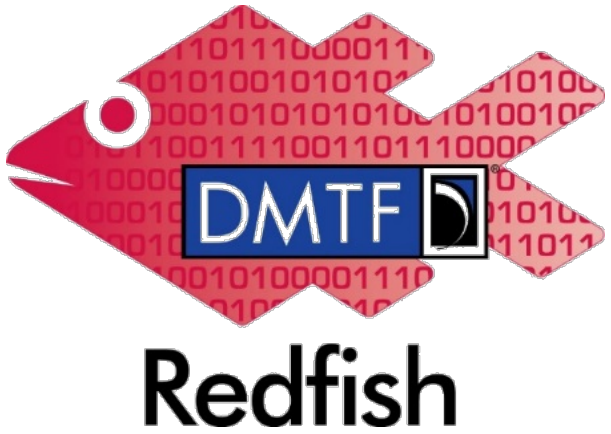
## Disclaimer

- The information in this presentation represents a snapshot of work in progress within the DMTF.
- This information is subject to change without notice. The standard specifications remain the normative reference for all information.
- For additional information, see the Distributed Management Task Force (DMTF) website.

# Agenda

- What is Redfish
- Design Tenets
- Basic Redfish Model
- How to Model in Redfish
- Model Deep Dive
- Ethernet Switching Approach
- Host Interface
- Redfish Device Enablement
- Profiles
- Open Source Efforts
- Developer Hub

# What is Redfish?




- **Industry Standard Software Defined Management for Converged, Hybrid IT**
  - HTTPS in JSON format based on OData v4
  - Schema-backed but human-readable
  - Equally usable by Apps, GUIs and Scripts
  - Extensible, Secure, Interoperable
- **Version 1 focused on Servers**
  - A secure, multi-node capable replacement for IPMI-over-LAN
  - Represent full server category: Rackmount, Blades, HPC, Racks, Future
  - Intended to meet OCP Remote Machine Management requirement
- **Expand scope over time to rest of IT infrastructure**
  - Additional features coming out approximately every 4 months
  - Working with SNIA to cover more advanced Storage (Swordfish)
  - Working with The Green Grid & ASHRAE to cover Facilities (Power/Cooling)
  - Work with the IETF to cover some level of Ethernet Switching

# Design Goals & Methods of Redfish

- Make it easy for the consumer
  - Easy to understand properties
  - Simple resource structures
  - Information model separate from protocol
- Human readable JSON payload
  - Assume consumers will never read specification or schema files
  - Start defining with Mockups!
- Don't expose the architecture
  - Think like a customer, not an implementer
- Design for Scalability
  - Minimize the number of IOs – few objects with lots of properties
  - Optimize common uses (80/20 rule)
  - Assume OData expand is implemented
- Resources are self-contained
  - No formal inheritance, No polymorphism
  - Only basic web programming skills needed to use the interface
- Don't invent, Re-use
- Add some structure
  - Know when to make an object in a resource vs. just some properties
  - Subordinate resources vs peer/referenced relationships
  - Know when to use arrays and when to use collections.
- "Future Proof"
  - Think about future extensions (e.g. Switches, Storage, etc.)
  - Think about OEM extensibility
- Know when to…
  - Use Actions
  - Use Annotations
  - Extend the schema language
- Pass thru data model
  - Except for OData-isms, don't put dependencies on the web servers to fix up payload

# Introduction to the Redfish data model

- All resources linked from a Service Entry point (root)
  - Always located at URL: /redfish/v1
- Major resource types structured in ‘collections’ to allow for standalone, multi-node, or aggregated rack-level systems
  - Additional related resources fan out from members within these collections

## Three Main Collections:

- **ComputerSystem**: properties expected from an OS console
  - Items needed to run the “computer”
  - Roughly a logical view of a computer system as seen from the OS
- **Chassis**: properties needed to locate the unit with your hands
  - Items needed to identify, install or service the “computer”
  - Roughly a physical view of a computer system as seen by a human
- **Managers**: properties needed to perform administrative functions
  - aka: the systems management subsystem (BMC)

# How to Model like Redfish

# Extending Redfish

- Two ways to extend the Redfish schema, both are valid
- Add to existing schema
  - New properties can be added
  - New embedded objects can be added
  - Values added to existing enumerations defined for a property
- Create new schema types
  - Can be a new subordinate resource
  - Or even a top-level collection
  - Check first for “inheritance by copy” – can you leverage an existing schema with just a “<thing>Type” property?
    - Example:  Rack Manager vs. BMC – both are Managers with a “ManagerType”

# Make it easy for the Consumer(s)

- Avoid mistakes from previous interface designs
  - Favor ease-of-user on the consumer (client) side
    - Even if it makes service implementation harder to develop
    - But this is a “balance” – ensure both sides can be implemented
  - Think like the consumer of the interface and how they use it
    - Previously, only a small developer community needed to understand the complex relationships. Now, there are thousands of programmers using these interfaces.
    - If you think “We can document that…”, think again.
  - Flatten the data model
    - Fewer, but richer (larger) resources to minimize I/O traffic and avoid over-use of references
- Follow use cases
  - Consumer will use a programming language or browser to access the interface
  - Think like the programmer using the variables/dictionary or a person reading the JSON via a browser-based REST client
  - Define the various “actors” and how they will use the interface, as there is more than one type of consumer of the interface.

# Human Readable JSON

- Start with a Mockup!
  - Copy the Redfish Mockup, modify it for your extensions
  - Use a web browser and a local web server to look at the data
  - Get data in the payload correct before developing schema to save development time
- Assume consumers will never read the Spec or the Schema
  - … nor should they be expected to
  - Property names should be intuitive in context
  - Use complete words like “Power” instead of “Pwr” unless name is too long
  - Don’t repeat
    - In the “Power” object, a property named “PowerCap” is redundant. It should just be “Cap”
    - The consumer is going to be using this in a variable or dictionary anyway, so they will already be referring to Power.Cap or Power/Cap. Power.PowerCap is not good design.

# Don’t expose the architecture

- As engineers, we tend to expose every little piece in the architecture. There is usually data or settings associated with every little piece.
  - Customers don’t care how you built it. They only care about access to the information. And they want it as clearly and concisely as possible.
  - If the architectural details don’t affect the use cases, mask for simplicity
  - Think like a customer, not an implementer
    - *“Don’t show the consumer your dirty laundry.”*
- Example 1:
  - You have a system with several management controllers inside that all work in concert with the main management controller.
  - Don’t expose them – make their properties part of the main management controller
  - The customer only cares about the functionality and not each little micro-controller
- Example 2:
  - You have a bunch of fans that are all connected to different I2C controllers
  - Just put them all in the chassis Fans array. Don’t create a bunch of additional “little chassis” in your implementation unless that affects performance or reliability.

# Scalability

- Minimize the number of I/Os
  - SSL overhead means I/O cost is in latency.  HTTP doesn't use socket connection model.
  - Each transaction is expensive but bandwidth (payload size) is cheap in comparison.
    - So create fewer objects with a larger set of properties
  - Use objects to collect & organize similar data for readability and shorten property names
- Optimize for the common use case (80/20 rule)
  - ComputerSystem has an embedded object that cover Processor Summary.  It also has links for more details on Processors.  Most consumers just need the summary.
  - By optimizing for the common use case, you can avoid additional I/Os.
- Develop the model assuming OData option "expand" is supported
  - But the Processor collection is a root link off of Computer System
  - With Expand Level=1, links to all of the processors are returned in the System object
  - With Expand Level=2, the processors and all their properties are returned in the System object
  - Thus, instead of 3 or more I/Os, supporting Expand returns it all in a single IO
  - More later under "Subordinate resources vs peer/referenced relationships"

# Scalability (continued)

- Model like you don’t have a query language
  - When grouping collections, determine what goes in each group
  - OData does have a query language, but it is not used in Redfish
    - Cumbersome for small footprint (management) services
  - It is easier for both the service and the client to return a whole collection
    - Requires less processing on the service side.
    - The client side is going to use the methods in the programming language to access the dictionary/array/construct to find the items it cares about anyway.
  - Put things logically into collections, subordinate or peer objects based on use case and consumer need

# Creating New Resources (Schemas)

- All resources are defined using a single “Resource” base class
  - Versioning within a schema is accomplished with traditional inheritance
- Redfish uses “*Inheritance by Copy*” to leverage properties
  - Copy useful property definitions from existing schema(s)
    - Add enumerations or additional properties as needed
  - Minimal use of true references to other schemas (e.g. “Status”)
    - Used for properties where model consistency is both useful and required
  - Keep existing semantics when leveraging properties to avoid customer confusion
    - Copy and rename slightly instead (e.g. Change “SpeedInRPM” to “SpeedInBPS”)
  - Keep leveraged properties in sync with original by hand
    - Avoid burdening customers with complex inheritance defined in multiple schema files
- Use “*Polymorphism by Union*” to re-use schemas for new “flavors”
  - For resources with multiple “types” (think CPU vs GPU), include properties for both
  - Add a “<thing>Type” property with enumeration so consumer can tell the difference
    - Example: “ProcessorType” with enumerations that include CPU and GPU

# Adding New Properties

- Make sure the property is useful to the consumer
  - Don’t expose unnecessary details or architecture
  - Ask “What will a user (or software) do with this property?”
- Naming
  - Property names should be clear and obvious in context of the schema
  - Use consistent terminology within the schema (and Redfish in general)
  - Balance between human readability and length
- Follow Redfish rules for CamelCase naming style
  - Acronyms larger than three letters become lower case
    - But use common sense, consider ignoring the rule for well known, common nomenclature
- Most properties are optional
- Many properties are null-able
  - Implementation will return property only if this instance supports it
  - If the instance doesn’t support the property, don’t return it
  - If the value is temporarily unavailable, return property with “null” value

# Property Data Types

- Numbers
  - Number types include units as part of the property name
    - Schema ‘units’ annotation is for programmatic consumption (not humans)
    - Use reasonably-sized units to allow for growth, but report typical values as integers
    - Do not create separate “units” properties to describe numbers
      - Unless there are multiple values than cannot be converted (e.g. FanSpeed in RPM vs. PWM value)
  - Don’t represent bitfields as integers
    - Forces users to read other specifications to decode
    - Software has to convert to strings for display, “1” / “0” vs. “Enabled” / “Disabled”
- Strings - Use enumerations instead whenever possible
- Enumerations
  - Increases interoperability by specifying values instead of free-form text strings
  - Allows discovery (<property>@Redfish.AllowableValues) of valid settings
  - Additional enumerations can be added in future schema versions
- Use schema annotation to reduce user errors
  - Min/max ranges, pattern properties, units, etc.

# Defining an embedded object vs. multiple properties

- If several properties go together, determine if they should be individual properties, or grouped into an embedded object
  - Embedded objects do create some additional burden on the consumer
  - Use them sparingly, but they have multiple applications
- Keep them as individual properties unless they hit any of the following:
  - If they start with the same name, put them in an object
  - If they are part of “polymorphism”: The resource has a “xxxType” property with enumerations and the properties do not apply to all of the types
  - If there are more than a few related properties
  - If the group of related properties is expected to expand over time
  - If a customer would expect them to “go together”
  - If you expect another implementation to leverage them (“inheritance by copy”)

# Subordinate resources vs peer/referenced relationships

- Two kinds of links in Redfish resources
  - Both contain “@odata.id”, the OData equivalent of “href”
- Subordinate resources
  - Represent more details about the ‘parent’ resource or contain collections
    - Keep the data model as flat as practical (minimize I/Os)
  - Example: “Processors” contains additional details for ComputerSystem
  - Link appears in root of the resource for simplicity and to optimize for OData Expand
- Related resources
  - Show a relationship (peer or reference) to the ‘parent’ resource
  - Example: “ManagedBy” shows the Manager for a ComputerSystem
  - These are all contained in a “Links” embedded object to differentiate
  - Avoids first-level usage of OData Expand (where it would not be useful)
- Optimize definition assuming OData Expand will be implemented by service
  - Expand levels, when requested by client, allows subordinate resources to be included in the returned ‘parent’ payload, reducing the number of I/Os
  - Collection resources (or a subset of the resource) can be auto-expanded by service

# Using Arrays and Collections

- Use arrays when:
  - Members will not come and go (empty array elements are “bad”)
  - Membership is fixed and finite
  - Ordinal value “natural” in context
  - Few interdependencies (reference) - links to members requires a JSON Pointer
  - Quantity too small to warrant a collection
  - Retrieving all members is a primary use case (like fans)
  - Consumer can access all members without harm (think permissions)
- Use collections when:
  - Members can be created or deleted or Provider will make them come and go.
  - There is no natural ordinal relationship
  - There will be interdependencies, peer references or external references
  - The member will have a lot of properties
  - A client will need to access members individually (think permissions)

# Future Proof

- Assume Redfish will span management for software defined infrastructure
  - Craft models that favor the direction the industry is headed
  - Specification and schema should last a decade or more
  - Think about disaggregation, fabrics, composability
  - Do not preclude future technical growth
    - Improper modelling of common functionality makes it impossible to leverage in the future. Such modeling should be avoided.
  - Be technology agnostic in names and structure whenever possible
    - Allow leverage for the next “big thing”, which is probably similar to the current “big thing”. (e.g. “DiskSize” not “ScsiDiskSize”).
- Think about growth in Redfish as it expands in breadth and depth
  - Accommodate expansion of scope to switches, storage, power infrastructure and other data center attributes
    - Think about them even if you’re not defining them
- Enable OEM extensibility
  - Ensure that models accommodate the common OEM extension sections in the base property set as well as the link section.

# Using and defining Actions

- In REST, Actions are accomplished through a series of operations
  - The “S” in REST stands for state (as in state machine)
  - But for complex operations, this becomes burdensome for the user
- Define Actions:
  - When atomicity is needed across multiple objects
    - We don’t have (or want) a commit/rollback mechanism. A single operation is much easier
  - When simplicity or efficiency is needed
    - If a log has 10,000 entries, deleting records individually is neither simple nor efficient
    - Don’t force consumer polling to track the machine state for long-duration state transitions
    - Example: “Reset” modeled using PATCH would require tracking the machine state (such as in a graceful shutdown), and also adding “current” and “requested” state
  - When the provider needs to control the operation
    - Exposing the gory details could prevent interoperability between implementations.
- But not for simple use cases:
  - Don’t use an action as a substitute for a POST to a collection. POSTs to create are allowed to have side affects to other resources in Redfish.
  - Don’t use an action when a PATCH would do, like setting an indicator LED.

# Using Annotations in the model

- Redfish uses common properties for unification
  - Name, Id, Members, @odata.id, @odata.type, status, etc.
  - Remember “inheritance by copy”?
    - It is common to reuse properties by copying them or referencing them.
- Annotations provide a method for including functionality without a copy
  - Example is @Redfish.Settings which provides for settings and a way for clients to determine if settings take place dynamically
  - Non-Redfish annotations are precluded outside of OEM section
    - Only @Redfish and @odata annotations are allowed
- Use Annotations when:
  - Functionality can be added to multiple resource types that would otherwise complicate the schema
- Don’t use them for:
  - Mixing metadata in with payload data
  - Concepts that are not likely to expand to other areas of the model

# Schema Creation

- Redfish supports two schema formats
  - Common Schema Definition Language (CSDL) as defined by OData v4
  - JSON-schema (www.json-schema.org) draft 4
- Both formats are normative and equivalent
  - Contents of the CSDL and JSON-schema version of a schema are the same (any inconsistency is an error)
  - Schema language semantics or capabilities may differ between the two formats, but functionality used by Redfish is equivalent
  - Schema writers are encouraged to publish in both formats
    - Both formats can be used for payload validation
    - JSON-schema is generally considered easier for humans to read
    - CSDL format is required for OData conformance and interoperability
- Tools to aid in schema creation and validation
  - SPMF is working on open source tools to create, transform and validate both CSDL and JSON-schema formats for Redfish schemas

# Extending the Schema language (CSDL / JSON Schema)

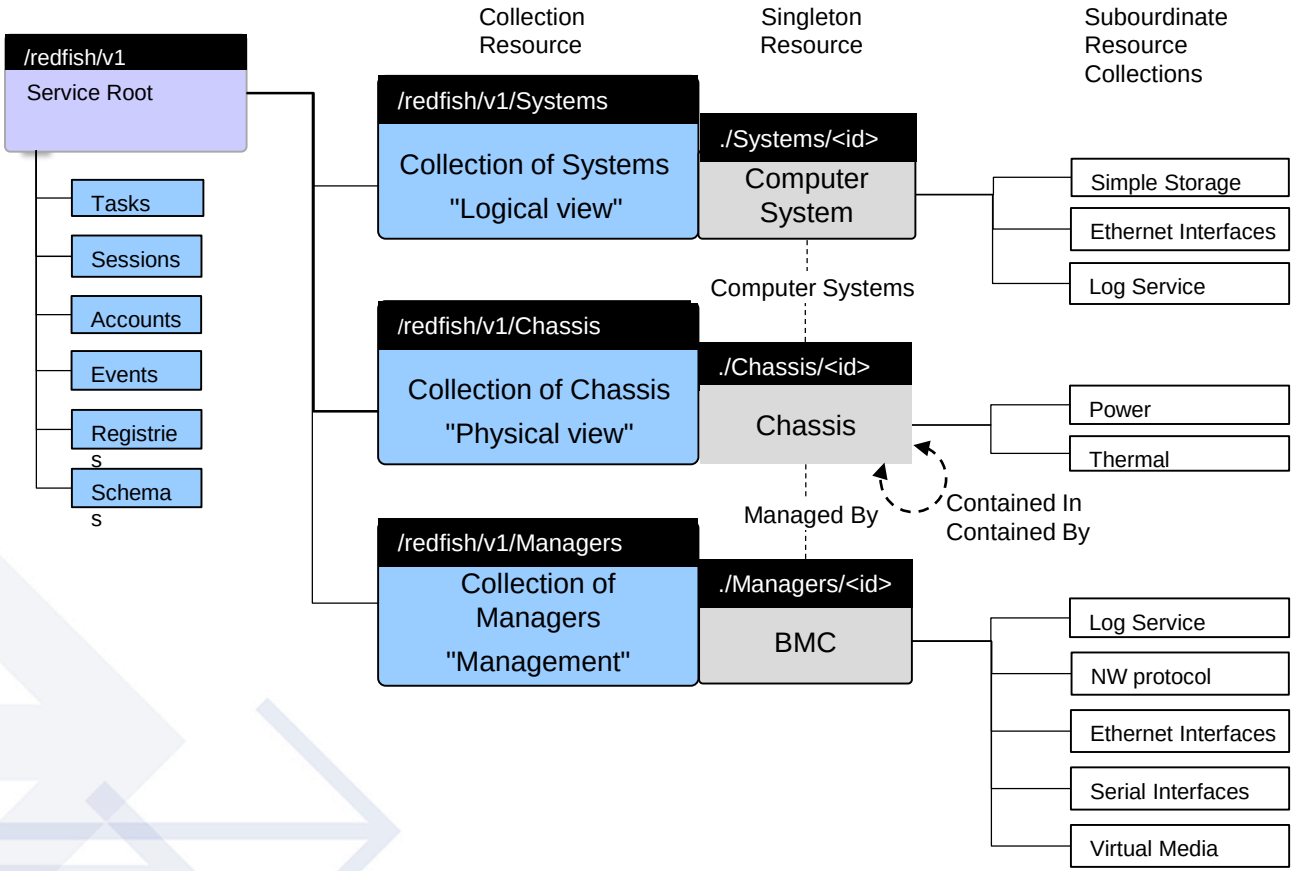
- CSDL and JSON schema do not have the same “qualifiers”
  - These are called annotations in CSDL and keywords in JSON Schema
  - Thus Redfish extended CSDL & JSON Schema to create a common set.
    - Added to JSON Schema: readonly, requiredOnCreate, longDescription, copyright, enumDescriptions
    - Added to CSDL: AllowableValues, Required, RequiredOnCreate, Settings, AutoExpand, IPv6Format, PropertyPattern, Copyright, Minimum, Maximum
- Add schema language extensions judiciously
  - Existing mechanisms can define much of what is in MOF
  - Ideas that the SPMF has had for extensions:
    - ETag, CorrelatableID, Default, Dependency, ModelCorrespondence, Profile
- Other Extensions are disallowed by the Redfish Specification
  - So work with SPMF to get qualifiers defined

# Pass thru data model

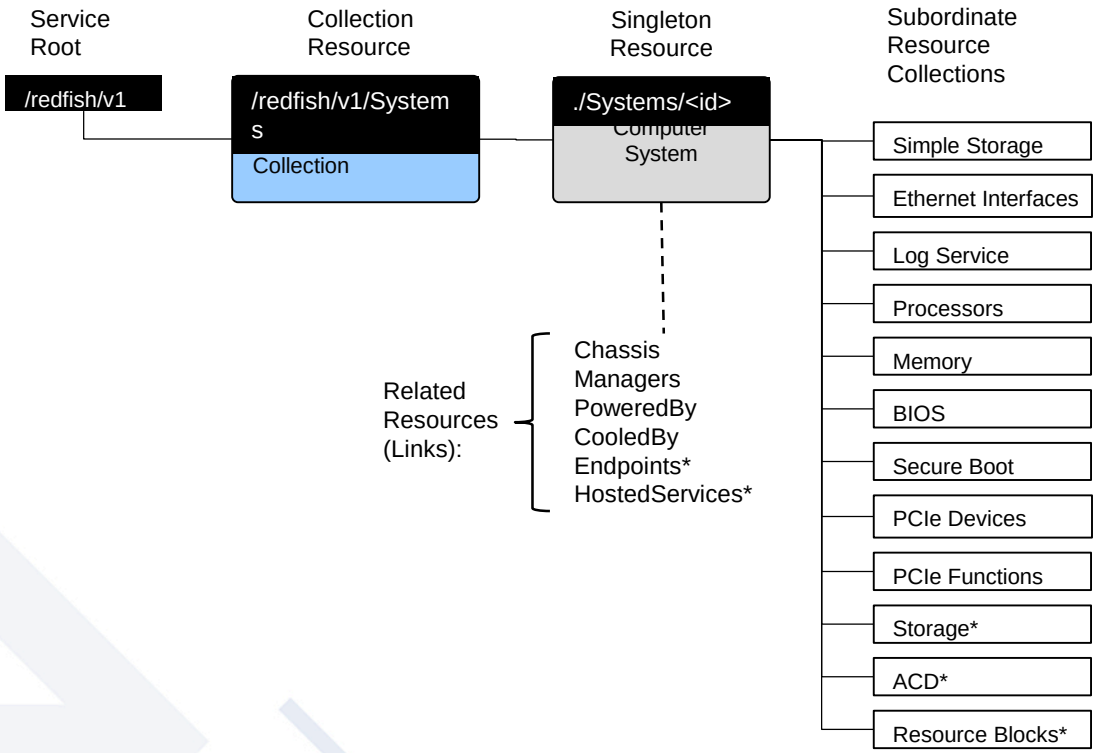
- In order to accommodate light-weight environments, one of the design goals was a “Pass Thru” data model.
  - Fixing up payloads with links puts a big burden on the web server implementation
- The “Providers” should be self contained
  - Should not reference items outside of their domain
  - This applies to properties, links, annotations and actions
- Except for @odata annotations for the specific resource, dependencies on the web server to fix-up payload are strongly discouraged
  - Don’t require the web server to fix-up links either in the root or in the links section, populate annotations or actions

# Death by Data Model

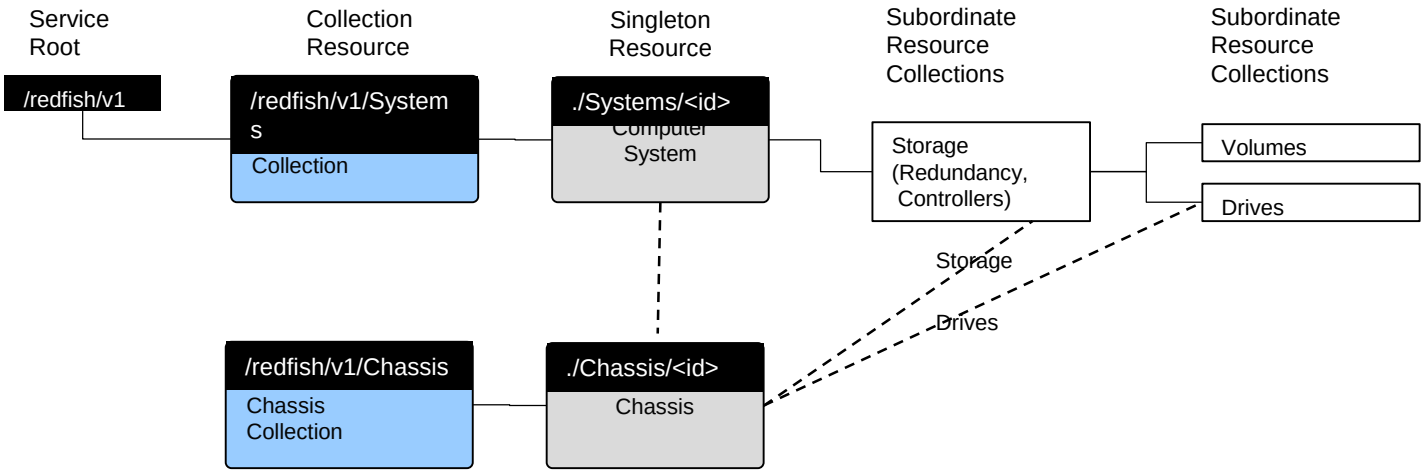
# Initial Redfish Data Model



# Current Server Redfish Data Model



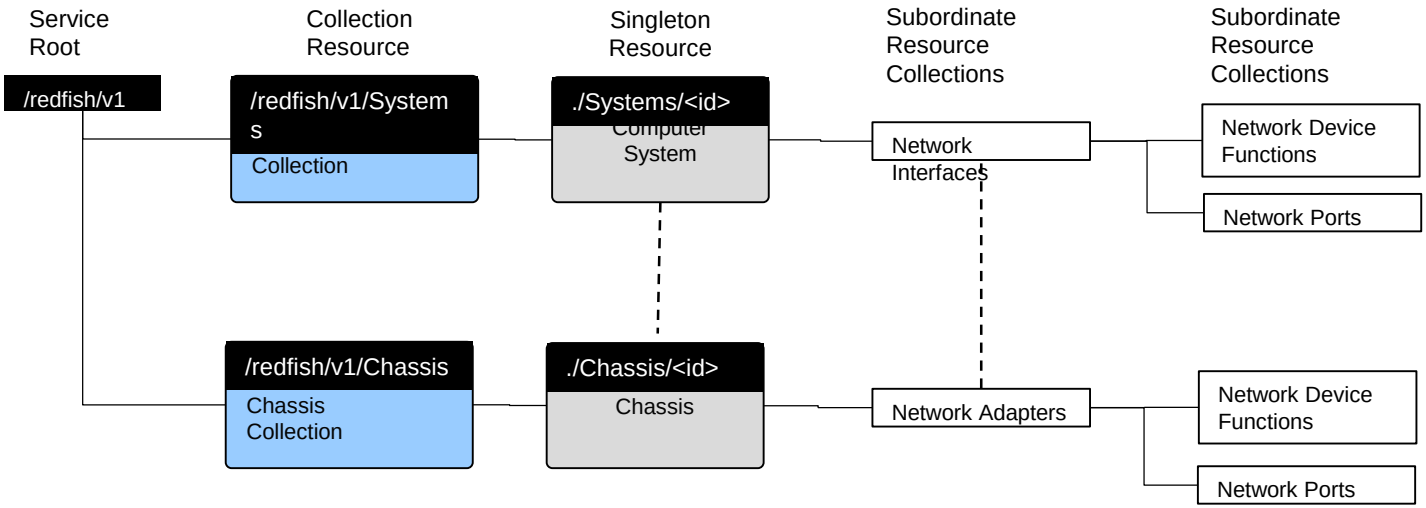
# Server Storage in Redfish




Redfish is a hypermedia interface, so we're not showing the path to the drives. They can be subordinate to the chassis (/chassis/1/drives/1) and related to the controller or the other way around (/system1/storage/1/drives/1). So you can show a JBOD to a controller in a different box, allow for dynamic assignment or composition or just show what is in this chassis.

This is a hardware representation. This is a different approach than Swordfish which is more akin to a service oriented model.

# Server Adv. Comm. Devices (ACD) in Redfish

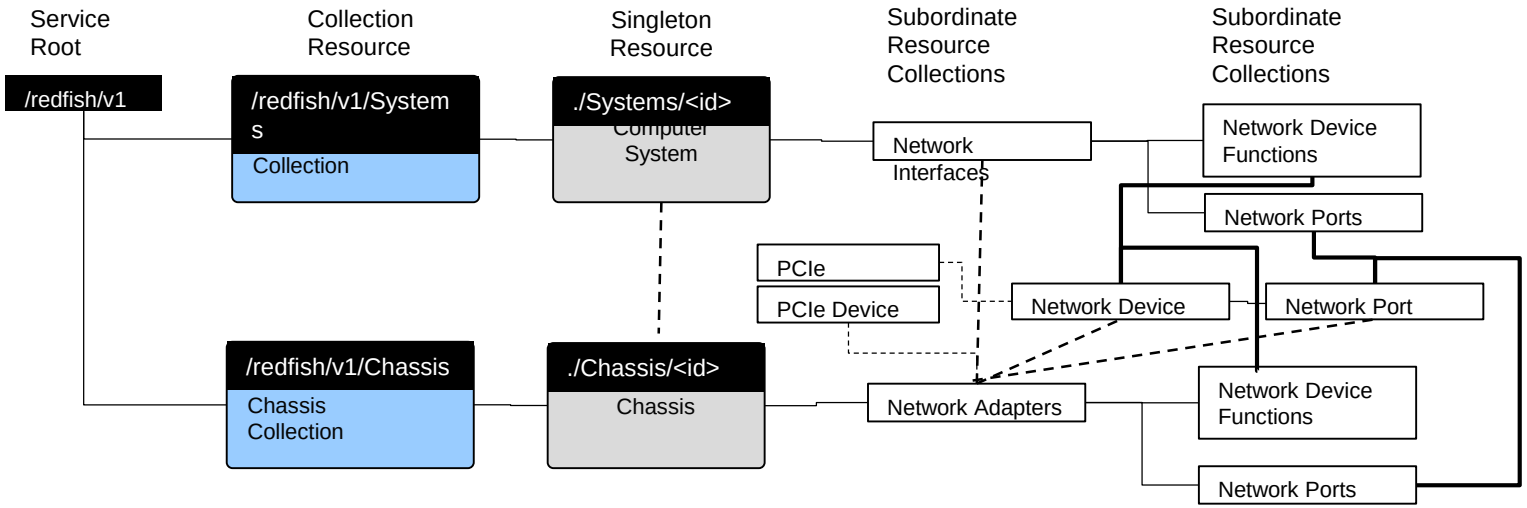



Network Interfaces is the logical view of the ACD. Network Adapter is the physical view and the contents may not be the same

- Such as in Compose where the OS sees only the part of the ACD it is allowed to know about).

Again, Redfish is a hypermedia interface, so we're not showing the path to the Network Device Functions and Ports. In fact, we're not showing the instances yet, just the collections. So lets add them…

# Server Adv. Comm. Devices (ACD) in Redfish (con't)

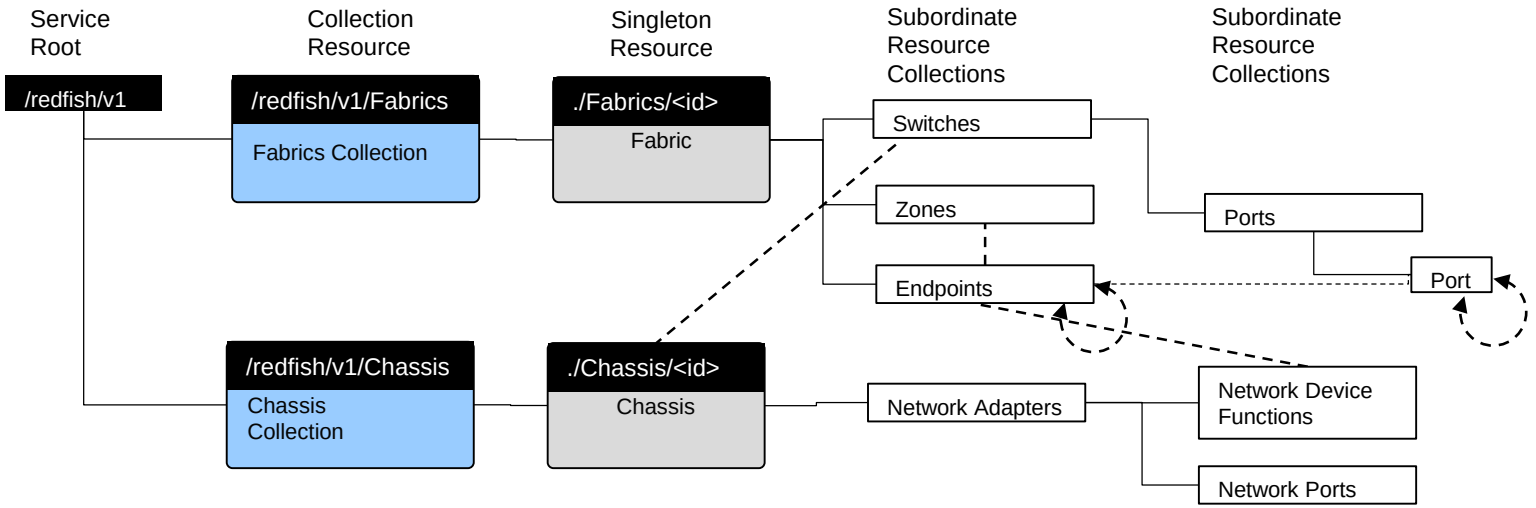



The lines are just hrefs basically. So from the chassis view you can make functions (PCIe Functions) on the Adapter and give the systems access (in the case of compose).

Collections are really just arrays of hrefs. So while this looks complicated, it allows for dynamic assignment of PCIe Functions to Ports and Systems and allows the creation of the Functions on the Network Adapters

Why are PCIe Devices & Functions different resources? Because of Fabrics…www.dmtf.org

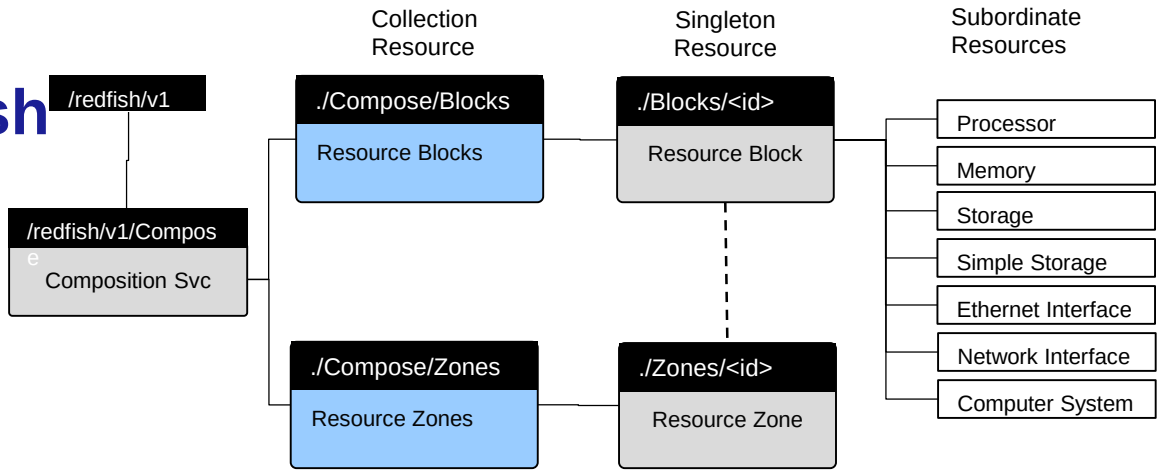
# Fabrics in Redfish




- Zones specify allowable access between endpoints.
- Devices/Entities are associated with Endpoints. The endpoint model is the underlying transport.
- Devices/Entities could be in Chassis or Computer Systems (kept out of picture to keep it simple)
- Adding Initiator/Target information to endpoints to help facilitate fabric management.

| Common Model | PCIe | SAS |
|---|---|---|
| Fabric | N/A | N/A |
| Zone | PCIe Zone | SAS Zone |
| Endpoint | PCIe Function | SAS EP |
| Switch | PCIe Switch | SAS Swtich |
| Port | PCIe Port | N/A |
| N/A | PCIe Device | N/A |

# Compose in Redfish




Blocks represent the resources.

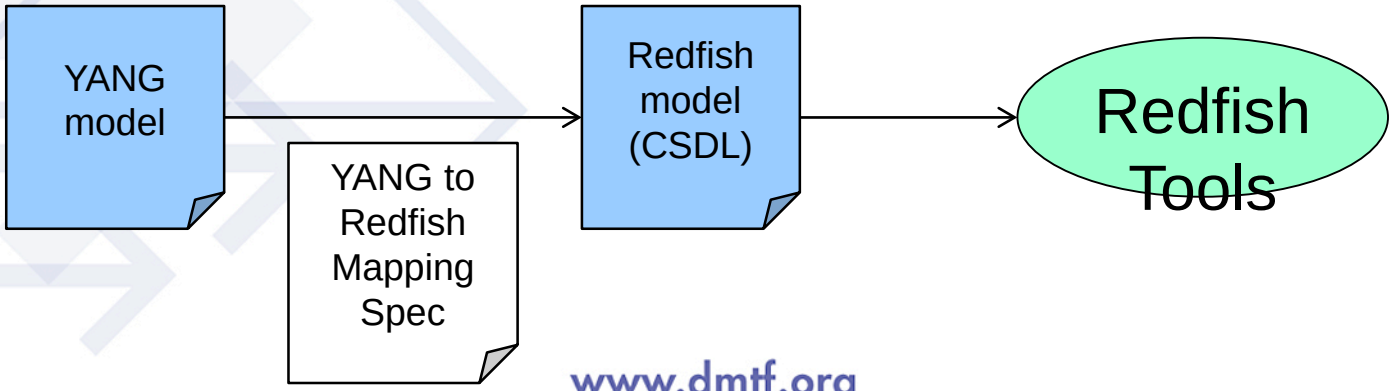
Zones represent the groups of resources that can be “composed” together. ResourceBlocks can be in more than one ResourceZone.

Compose a system by POSTing to Computer System

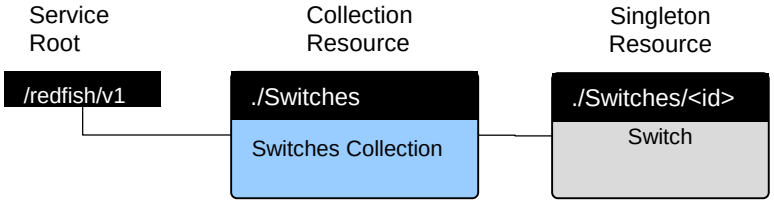
# Back to Redfish

# Ethernet Switch Representation: YANG to Redfish

- Enable converged infrastructure management
  - One interface (one tool chain) to manage compute, storage and network
  - Switches have platform components in common with servers and storage
  - Network Functions Virtualization (NFV) will need common manageability for compute and networking
- DMTF wants to leverage the networking industry's expertise
  - YANG is the basis for general network industry manageability
  - Large body of existing YANG work
  - Model driven approach to network management



# Ethernet Switches in Redfish

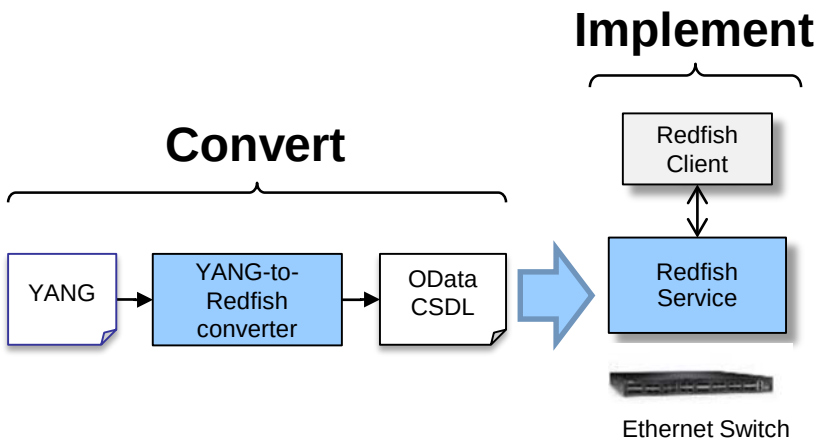



YANG Models

- Phase 1 - convert a small set of YANG models to Redfish models to manage an Ethernet Switch
  - Prove out the process
  - Publish and validate the programmatic conversion tool
- Phase 2 – larger list of YANG models

**Phase 1 RFCs**

- RFC7223 (Interfaces)
- RFC7224 (IANA Interface types)
- RFC7277 (IPv4 and IPv6)
- RFC7317 (system, system_state, platform, clock, ntp)

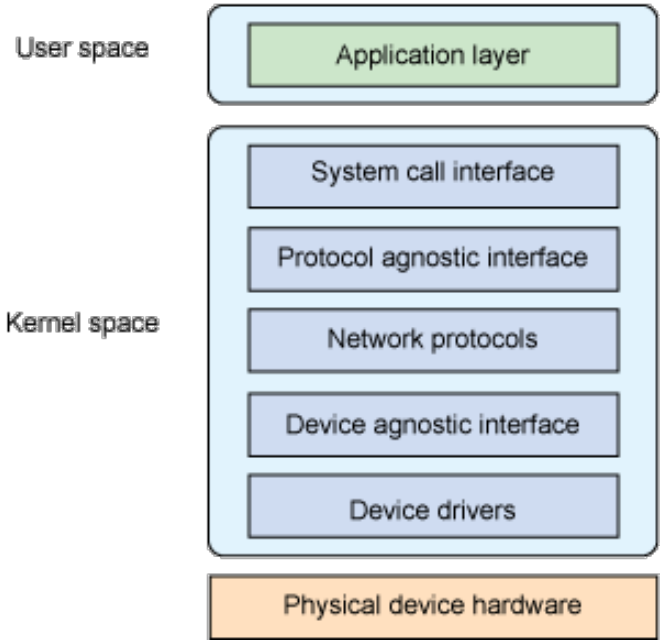


# Timeline of Redfish™ Specification

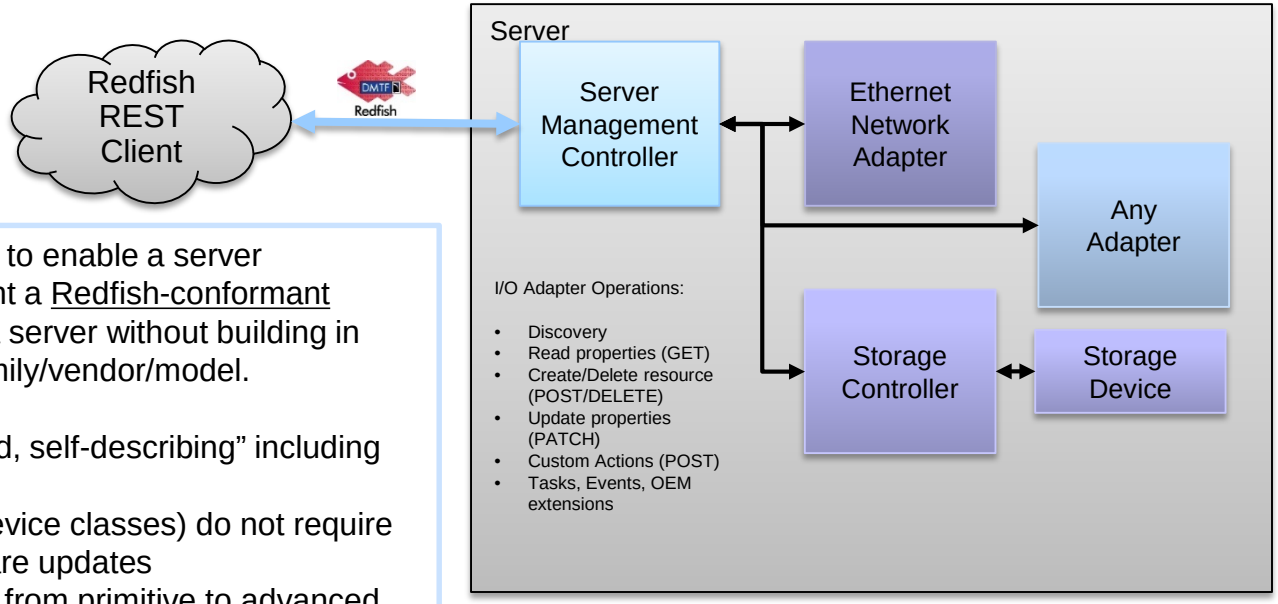
- The DMTF Redfish technology
  - Sep 2014: SPMF Formed in DMTF.
    - Released multiple work-in-progress for public feedback
  - Aug 2015: Redfish Specification with base models (v1.0)
  - May 2016: Models for BIOS, disk drives, memory, storage, volume (2016.1)
  - Aug 2016: Models for endpoint, fabric, switch, PCIe device, zone, software/firmware inventory & update (2016.2)
  - Dec 2016: Adv. communications devices (multi-function NICs), host interface (KCS replacement), privilege mapping (2016.3)
  - May 2017: Composability (2017.1)
    - WIP for Telemetry
  - Aug 2017: Location, errata (2017.2)
    - WIPs for Ethernet Switching, DCIM, OCP & Profiles
- Alignment with other standard organizations
  - Aug 2016: SNIA releases first model for network storage services (Swordfish)
  - Working with IETF to create a YANG Redfish mapping algorithm
  - DMTF created work registers with UEFI Forum, TGG, OCP, ASHRAE, Broadband Forum, ETSI-NFV, ODCA for work on applying Redfish

# Host Interface

- Replacement for IPMI KCS, etc.
- Exposes a NIC from Management Controller to OS
  - SMBIOS records provide information to allow kernel access
- Same access in-band as out-of-band
  - Kernel mode or user mode accessible
  - Encouraging OS vendors to begin consuming Redfish data.
  - This means you can get to the iLO homepage from the OS.
  - This means you can write your tools for the iLO homepage or Redfish and run them in the host OS.
  - Anything that accesses the out of band can be run on the host OS to access to local management subsystem.



# Redfish Device Enablement: PLDM Redfish Providers




PMCI WG developing a standard to enable a server Management Controller to present a Redfish-conformant management of I/O Adapters in a server without building in code specific to each adapter family/vendor/model.

- Support adapter "self-contained, self-describing" including value-add (OEM) properties
- New managed devices (and device classes) do not require Management Controller firmware updates
- Support a range of capabilities from primitive to advanced devices (lightweight/low bandwidth options)
- Leveraging PLDM, a provider architecture is being specified that can binary encode the data in a small enough format for devices to understand and support.
- MC acts as a broker to encode/decode the data to/from the provider
- PLDM works over I2C & PCIe VDM. Additional mappings under consideration.

# Interoperability Profiles: Goals

- An “Interoperability Profile” provides a common ground for Service implementers, client software developers, and users
  - A profile would apply to a particular category or class of product (e.g. “Front-end web server”, “NAS”, “Enterprise-class database server”)
  - It specifies Redfish implementation requirements, but **is not** intended to mandate underlying hardware/software features of a product
  - Provides a target for implementers to meet customer requirements
  - Provide baseline expectations for client software developers utilizing Redfish
  - Enable customers to easily specify Redfish functionality / conformance in RFQs
- Create a machine-readable Profile definition
  - Document must be human-readable
  - Can be created by dev/ops personnel and non-CS professionals
- Enable authoring of Profiles by DMTF, partner organizations, and others
- Create open source tools to document and test conformance

# Redfish Developer Hub: redfish.dmtf.org

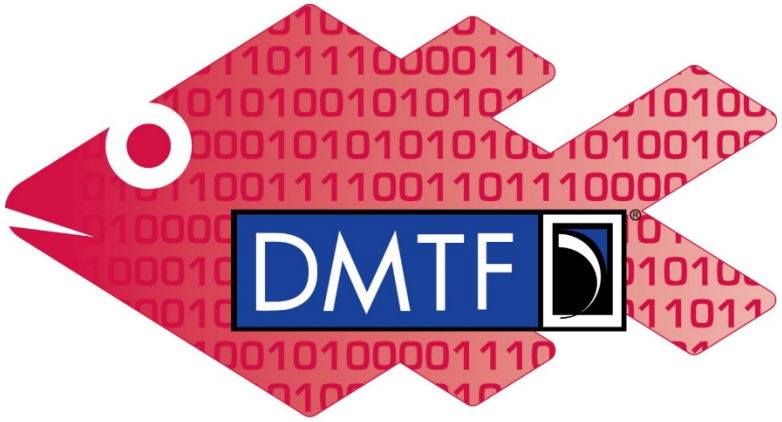
- Resources
  - Schema Index
  - Specifications
  - GitHub for Redfish Tools
  - Registries
  - Other Documentation
- Mockups
  - Simple Rack-mounted Server
  - Bladed System
  - Proposed OCP Redfish Profile
  - More being added
- Education/Community
  - Redfish User Forum
  - Whitepapers, Presentations
  - YouTube shorts & Webinars

# Tools Task Force: Redfish Tools Description

SPMF Tools TF open source tools to enable Redfish  http://www.dmtf.org/standards/opensource

| | Tool | Description |
|---|---|---|
| Extend | CSDL Validator | Validates the CSDL conforms to Redfish requirements |
| | CSDL-to-JSON schema convertor | Generates json-schema files from CSDL |
| | YANG to Redfish | Converts a YANG model into a set of Redfish CSDL files, enabling Ethernet switching standard access via Redfish |
| | Document Generator | Generates documentation from json-schema |
| Working Svc | Mockup Server | Exposes a mockup as a static HTTP service (GETs only) |
| | Mockup Creator | Creates a mockup from a Redfish service |
| | Profile Simulator | Dynamic simulator of the proposed Redfish profile for OCP |
| | Interface Emulator | Emulate a Redfish interface statically or dynamically. |
| Test | Service Validator | Validates a Redfish service is conformant |
| | JSON Schema Response Validator | Validates any JSON resource against DMTF provided JSON schemas |
| | Reference Checker | Validates the reference URLs in CSDL files |
| | Use Case Checker | Collection of tools to validate common use cases for Redfish Services. |
| | Service Conformance Tool | Verifies conformance of a Redfish service to assertions in the Redfish Specification |
| Client | CLI (redfishtool) | A command line tool for interacting with a Redfish service (similar to ipmitool) |
| | Event Listener | A lightweight HTTPS server that can be executed to read and record events from a Redfish Service |
| | C Library (libRedfish) | C libraries for interacting with Redfish services |
| | Python Utility & Library | A Command line tool with UI and python libraries for interacting with Redfish services |

# Thank you!




Redfish

www.dmtf.org

# Backup Material