Implementation of the DMTF Redfish API on Dell EMC PowerEdge Servers

Dell EMC Customer Solution Centers
Jonas Werner, Sr. Solution Architect

Dell EMC Server Solutions
P. Raveendra Reddy, Platform Software Staff Engineer
Texas Roemer, Test Principal Engineer
Paul Rubin, Sr. Product Manager

October 2017

A Dell EMC Technical White Paper

Revisions

Date	Description
March 2016	Initial release
June 2017	Updated for 14th generation of PowerEdge release
October 2017	Updated for iDRAC7/8 2.50.50.50 release

Contents

Executive summary

The growing scale of cloud- and web-based data center infrastructure is reshaping the requirements of IT administrators world-wide. New approaches to systems management are needed to keep up with the growing and changing market.

The Distributed Management Task Force (DMTF) Scalable Platforms Management Forum (SPMF) has published Redfish, an open industry-standard specification and schema designed to meet the needs of IT administrators for simple, modern, and secure management of scalable platform hardware. Dell EMC is a key contributor to the Redfish standard, acting as co-chair of the SPMF, promoting the benefits of Redfish, and working to deliver those benefits within Dell EMC industry-leading systems management solutions.

This technical white paper provides an overview of the Redfish Scalable Platforms Management API standard and describes the Dell implementation of Redfish for the12th, 13th, and 14th generation PowerEdge servers—delivered by the integrated Dell Remote Access Controller (iDRAC) with Lifecycle Controller.

1 Introduction

Since the inception of the x86 server in the late 1980's, IT administrators have sought the means to efficiently manage a growing number of distributed resources. Industry suppliers have responded by developing management interface standards to support common methods of monitoring and controlling heterogeneous systems.

While management interfaces such as SNMP and IPMI have been present in data centers for the past decade, they have not been able to meet the changing requirements due to security and technical limitations.

Further, the scale of deployment has grown significantly as IT models have evolved. Today, organizations often rely on a large number of lower-cost servers with redundancy provided in the software layer, making scalable management interfaces more critical.

To meet such market requirements, a new, unifying management standard was needed.

This technical white paperr describes Redfish—a next generation management standard using a data model representation inside a hypermedia RESTful interface. The data model is defined in terms of a standard, machine-readable schema, with the payload of the messages expressed in JSON and the protocol using OData v4. Because it is a hypermedia API, Redfish is capable of representing a variety of implementations by using a consistent interface. It has mechanisms for discovering and managing data center resources, handling events, and managing long-lived tasks.

Dell EMC is enhancing its leading Systems Management capabilities with the introduction of Redfish support on the iDRAC with Lifecycle Controller. This technical white paper provides the required information to create Redfish clients or use existing REST clients to deliver the benefits of the Redfish API on PowerEdge servers. This technical white paper can also be used to help legacy management consoles support or enable the Redfish standard.

2 The Redfish management standard

There are various Out-of-Band (OOB) systems management standards available in the industry today. However, there is no single standard that can be easily used within emerging programming standards, can be readily implemented within embedded systems, and can meet the demands of today's evolving IT solution models.

New IT solutions models have placed new demands on systems management solutions to support expanded scale, higher security, and multi-vendor openness, while also aligning with modern DevOps tools and processes.

Recognizing these needs, Dell EMC and other IT solutions leaders within the DMTF undertook the creation of a new management interface standard. After a multi-year effort, the new standard, Redfish v1.0, was announced in July, 2015.

Its key benefits include:

- Increased simplicity and usability
- Encrypted connections and generally heightened security
- A programmatic interface that can easily be controlled through scripts
- Ability to meet the Open Compute Project's Remote Machine Management requirements
- Based on widely-used standards for web APIs and data formats

Redfish has been designed to support the full range of server architectures from monolithic servers to converged infrastructure and hyper-scale architecture. The Redfish data model, which defines the structure and format of data representing server status, inventory and available operational functions, is vendor-neutral. Administrators can then create management automation scripts that can manage any Redfish compliant server. This is crucial for the efficient operation of a heterogonous server fleet.

Using Redfish also has significant security benefits—unlike legacy management protocols, Redfish utilizes HTTPS encryption for secure and reliable communication. All Redfish network traffic, including event notifications, can be sent encrypted across the network.

Redfish provides a highly organized and easily accessible method to interact with a server using scripting tools. The web interface employed by Redfish is supported by many programming languages, and its tree-like structure makes information easier to locate. Data returned from a Redfish query can be turned into a searchable dictionary consisting of key-value-pairs. By looking at the values in the dictionary, it is easy to locate settings and current status of a Redfish managed system. These settings can then be updated and actions issued to one or multiple systems.

Since its July, 2015 introduction, Redfish has continued to grow and evolve with specification updates released in 2016 covering key operations such as BIOS configuration, server firmware update, and detailed server inventory.

2.1 Next-generation server management

The DMTF white paper on Redfish, DSP2044, describes the need to move away from managing servers as individual “pets”. Rather, administrators should begin to treat their servers more as “cattle” and manage them as “herds”. While in the past, IT staff could spend time to adapt their management methods to match a smaller number of servers, they now have many more servers and much less time. Managing a large and growing infrastructure requires the capability to issue commands at scale with the expectation that the “herd” will follow regardless of make or model of the individual servers.

2.2 Rack, multi-node, and nested chassis

Another limitation of legacy management standards is an implied understanding that one management endpoint such as a Baseboard Management Controller or BMC means one server. Modern converged server infrastructure such as the PowerEdge M1000e and FX2 are becoming more prevalent, invalidating this assumption. Redfish explicitly addresses converged infrastructure and rack-level management with modeling that can scale for the management of multiple nodes, nested chassis, and server blades within a larger, actively managed enclosure.

2.3 Keeping up with changes in IT philosophy

Redfish has taken into account the recent changes in the IT field. These changes include not only new types of hardware but also important changes in IT philosophy that are impacting how administrators expect to manage their infrastructure.

Organizations are now looking for open management solutions that can be controlled in the same way they control other resources, irrespective of whether the resources are located in a cloud or in a data center. By adopting data structures and access methods as used for cloud- and web-based infrastructure, Redfish will enable management methods aligned with modern IT infrastructure. Utilizing a modern data model and RESTful API, Redfish can be readily integrated with the IT automation tools and processes employed by DevOps practices, a key requirement in many IT organizations.

System administrators can use Redfish to manage heterogeneous server fleets more efficiently throughout the server lifecycle— from bare metal deployments to maintenance and repurposing. Using a simple and powerful interface that supports modern automation technologies, Redfish can speed time-to-solution for IT developers.

2.4 Redfish key technologies

Redfish is a RESTful interface over HTTPS in JSON format based on ODATA v4 usable by clients, scripts, and browser-based GUIs. It utilizes a range of IT technologies that have been selected because of their widespread use. By adopting these accepted technologies, administrators will find it easier to use Redfish. Taken together, these technologies create a new foundation from which servers can be managed by using common programming and scripting languages, such as Python, Java, and C.

2.4.1 HTTPS communication

The Hypertext Transfer Protocol or HTTP is an application protocol for distributed, collaborative, hypermedia information systems and forms the foundation of data communication for the World Wide Web. Secure HTTP or HTTPS is a secure version of HTTP that enables secure communications by operating HTTP within a network connection encrypted by TLS or SSL. By utilizing HTTPS, Redfish significantly enhances the security of server management especially in comparison to legacy server management protocols.

2.4.2 RESTful application programming interface

REpresentational State Transfer or REST is a software architectural style used within the World Wide Web. Since 2000, when representational state transfer was introduced and defined by Roy Fielding in his University of California Irvine doctoral dissertation, REST has been applied for a range of purposes including the definition of web-based APIs. Systems that adhere to REST practices are often referred to as RESTful interfaces and typically use the HTTP Methods (GET, POST, DELETE, and more) that web browsers use to access web pages. RESTful architectures are now commonly used by many IT solutions. Leveraging this standardized approach, Redfish implements a RESTful API for accessing management information and for issuing commands to change the configuration or operational state of a server.

2.4.3 JSON data

Redfish represents data using JSON. JSON is a lightweight data-interchange format that is easy for people to read and write and also for machines to parse. JSON is based on a subset of the JavaScript Programming Language, using a text format that is completely language independent but uses conventions familiar to programmers of the C-family of languages such as C, C++, C#, Java, JavaScript, PERL, and Python. These properties make JSON an ideal data-interchange language.

2.4.4 OData

OData is an open protocol standard for the definition and exchange of information using RESTful APIs. OData was originally created in 2007 by Microsoft and subsequently standardized by the OASIS standards body. When implementing a common interface across multiple vendors, it becomes important to standardize the data formats. OData provides Redfish the required framework to ensure that the data structures remain interchangeable between server vendors.

2.4.5 Eventing

The Redfish specification includes support for eventing that enables the notification to a management client of significant events occurring in a server. Redfish provides push style event notifications to an event listener, defined as a Redfish compliant HTTPS server. The listener subscribes to the events of interest based on the event types defined in the Redfish specification. Event subscriptions remain in place until specifically deleted or until the Redfish manager such as iDRAC is reset to its default configuration.

Upon receiving an event subscription request, iDRAC will add the hostname of the requestor to the list of targets to be notified when the event occurs. In the initial iDRAC Redfish implementation, all events are categorized under the Alert Event Type with a maximum of 20 event subscriptions per event listener. If delivery of an event notification fails, the event service will retry delivery. The parameters for re-delivery are configurable.

Redfish events are delivered only over HTTPS transport. In the initial iDRAC Redfish implementation, HTTPS certificates are disabled; this requires the event listener to support receiving event notifications without certificate validation. Eventing with certificate support is planned for a future release.

2.5 Redfish operational model

Redfish operations are initiated by a client using HTTPS for GET, POST, PATCH and DELETE operations and capable of interpreting the JSON responses from the managed server. The responses provide the requested information and indications of success or failure of the requested operation.

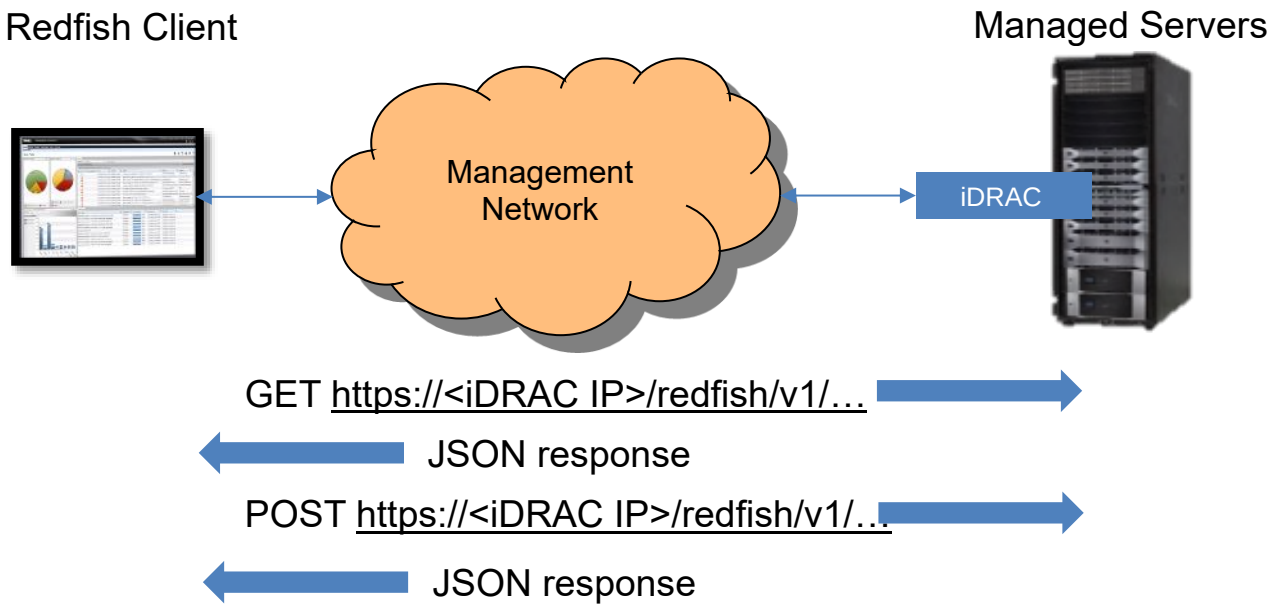


Figure 1 Redfish operational model

2.5.1 Redfish client

The REST principle of "Everything is a Resource" means that every Uniform Resource Identifier or URI represents a resource of a specific type. This can be a service, a collection, an entity or some other construct. In the Redfish context, a resource can be thought of as the content of the HTTP message returned when accessing a URI. A variety of REST Clients can be used for gaining access to Redfish resources such as:

- Web browser plug-ins such as “Advanced REST Client”, “Postman”, “REST Easy” and “RESTClient”
- cURL, Python, and other scripting/programming languages that provide support for dealing with URIs and for parsing JSON payloads.

2.6 Redfish architecture

Because the RESTful API employed by Redfish is web-based, access is provided using URIs, which can be typed into a web browser. The Redfish API uses a simple folder structure that starts with the Redfish root at

“/redfish/”. In the case of a PowerEdge server, the root is accessed through the URI https://<iDRAC IP>/redfish/v1/ - the “v1” at the end of the URI denotes the version of the API.

The URI is the primary unique identifier of resources. Redfish URIs consist of three parts as described in RFC3986: Part one defines the scheme and authority of the URI, part two specifies the root service and version and part three defines a unique resource identifier.

For example, in the following URI: https://mgmt.vendor.com/redfish/v1/Systems/SvrID

- https://mgmr.vendor.com is the scheme and authority
- /redfish/v1 is the root and version
- /Systems/SvrID is the resource identifier

2.6.1 Redfish tree structure

From the top-level root, the RESTful interface branches out to cover a number of “Collections”, which each in turn include multiple sub-items, creating a tree-like structure. The administrator can drill down through this structure to find information and settings of interest.

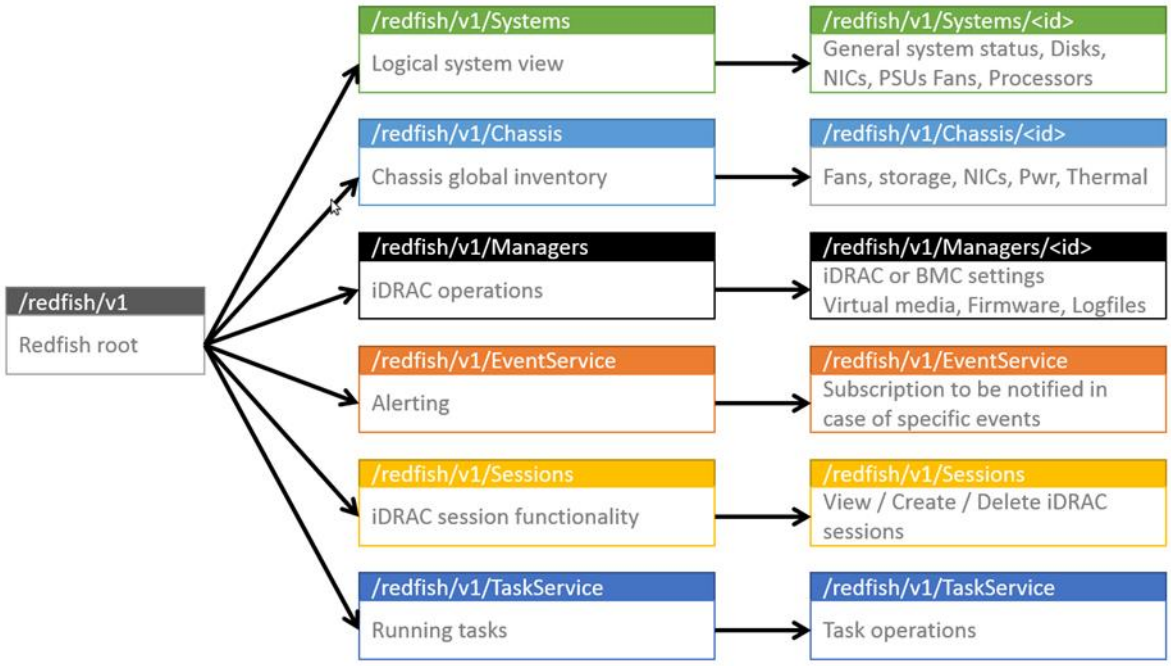
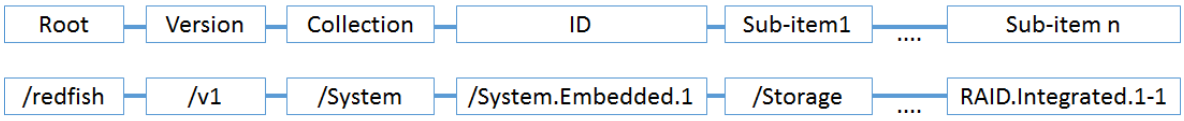


Figure 2 Redfish API tree structure

For example, accessing the Redfish structure for the PERC RAID controller on a PowerEdge R630 server would be navigated by using the following path:

https://<iDRAC IP>/redfish/v1/Systems/System.Embedded.1/Storage/Controllers/RAID.Integrated.1-1

Or, more graphically represented as:



Note that the API is best navigated starting from the root as some portions of an API path can vary depending upon the server hardware configuration. For example, the “RAID.Integrated.1-1” Sub-item in the preceding example may be different when another type of RAID controller is installed in the managed server.

2.6.2 Redfish operations

In Redfish, HTTP methods implement the operations of a RESTful API. This allows the user to specify the type of request being made. It adheres to a standard CRUD (Create, Retrieve, Update, and Delete) format. Depending on the desired result, a user can issue the following types of commands:

- GET View data
- POST Create resources or use actions
- PATCH Change one or more properties on a resource
- DELETE Remove a resource

Note: In the current implementation, HEAD and PUT operations are not supported for Redfish URIs.

Creation and removal of data are limited depending on the management characteristics of the resource being targeted. Generally, viewing and changing settings will be more common.

2.6.3 Authentication

Depending upon the sensitivity of a given resource, Redfish clients will be required to authenticate their access. The required credentials and supported forms of authentication are determined by the platform being managed. In the case of iDRAC, authentication is supported using local iDRAC credentials or any of the other supported authentication methods, such as LDAP and Active Directory.

Access to iDRAC data is allowed by authenticated and authorized users only, except as noted below. Authentication is achieved using a subset of the common HTTP headers supported by a Redfish service – in particular, the X-Auth-Token header. More details on authentication are provided in the “Session Management” section of the Redfish specification.

Authorization covers both user privilege and license authorization. Note that iDRAC Redfish support does not require any special or separate licensing. The following table details the authentication and authorization required for each iDRAC Redfish action:

Redfish Actions	Authentication Required	Authorization Required
Read operation on any instrumentation data	Yes	Yes
Write operation on any instrumentation data	Yes	Yes
Execute operation on any instrumentation data	Yes	Yes
View Service root	No	No
View Metadata document	No	No
View OData Service Document	No	No
View Message Registry	No	No

Table 1 iDRAC Redfish authentication and authorization requirements

Unlike certain management interfaces that restrict authentication to a single command, the Redfish Service provides access to Redfish URIs by using two methods:

- **Basic authentication**: In this method, user name and password are provided for each Redfish API request.
- **Session based authentication**: This method is used when issuing multiple Redfish operation requests.
  - Session login is initiated by accessing the Create session URI. The response for this request includes an “X-Auth-Token” header with a session token. Authentication for subsequent requests is made using this “X-Auth-Token” header.
  - Session logout is performed by issuing a DELETE of the Session resource provided by the Login operation including the X-Auth-Token header.
  - Using this approach, Redfish supports multiple transactions within a session with the help of X-Auth-token, session token and Location headers.

2.6.4 Privileges

Privilege model requirements are aligned to the Redfish specification and schema. The following table shows the relationship between Redfish Privileges and native iDRAC Privileges:

Redfish Privileges	iDRAC Privileges
Login	Login
ConfigureManager	Config iDRAC
ConfigureUser	Config User
ConfigureManager	System Control
ConfigureComponents	Virtual Console
ConfigureComponents	Virtual Media
ConfigureManager	Clear Logs

Table 2 Mapping Redfish privileges to iDRAC privileges

3 Using the Redfish API on PowerEdge systems

The following tests were carried out using a PowerEdge R740 server loaded with firmware version 3.00.00.00, conformant with Redfish 2016.R1 and R2 releases. If a Redfish conformant server is not available, a simulated server Redfish interface is available at the DMTF website: http://redfish.dmtf.org/redfish/v1.

3.1 Web browser access

Because REST is a web-based API, a typical web browser such as Microsoft Internet Explorer or Google Chrome can be used for access. Start the web browser and enter the iDRAC IP address or hostname followed by /redfish/v1/ and the Redfish root is displayed as shown in the following figure:

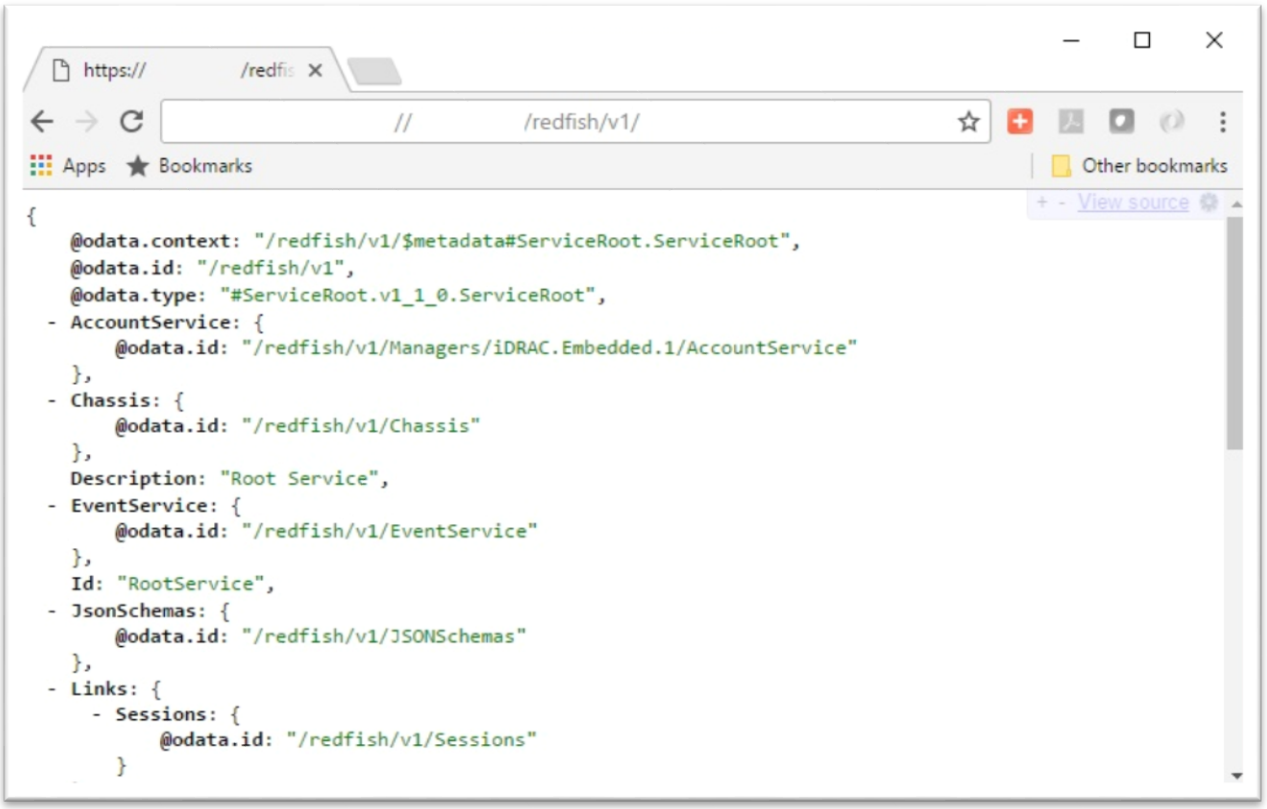


Figure 3 Accessing PowerEdge Redfish interface using a web browser

This figure illustrates the JSON response to a GET query using the Postman browser plug-in.

Each of the “@odata.id” tags can be explored individually to allow a user to drill down deeper into the Redfish tree, but further access will prompt for authentication.

For example, Figure 4 shows how to access the Chassis collection. When the /redfish/v1/Chassis/System.Embedded.1 URI is accessed, a pop-up box appears prompting for the entry of

an iDRAC user name and password. After these credentials are validated, more system details and additional @odata.id tags are presented:

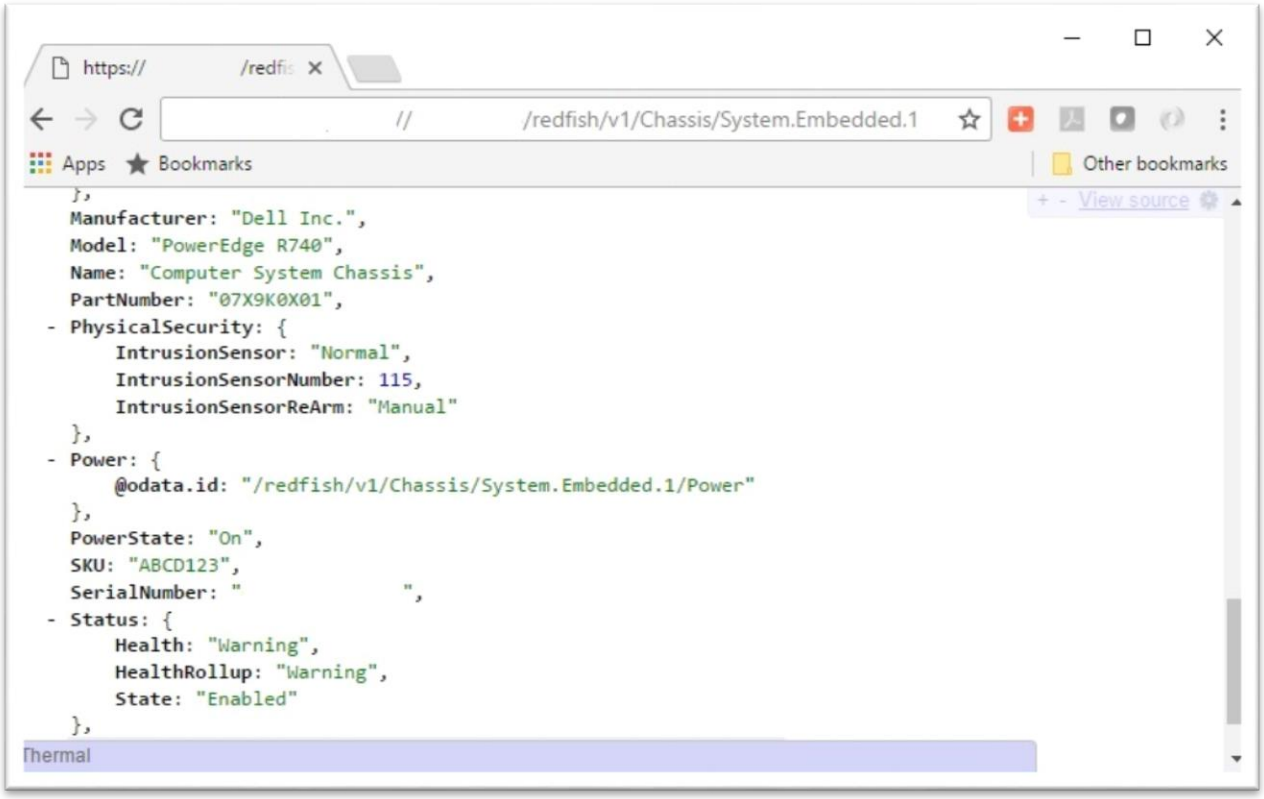


Figure 4 iDRAC Redfish authenticated server details access

3.2 Accessing Redfish by using the cURL application

Interacting with Redfish using scripting tools can be extremely powerful. The following examples use a Dell Latitude E7440 running Ubuntu to access the Redfish API from the command line using the cURL application.

cURL is a powerful open source command line tool for interacting with various web-based services. It supports both HTTP and HTTPS in addition to other protocols. In the case of Redfish, cURL can be used to test the availability and function of the REST interface.

Supplying cURL with the iDRAC IP address and the Redfish root will access the service root. If the server has a self-signed certificate, as in this case, the certificate check can be skipped using the cURL option “-k”.

Command:

```
curl "https://<iDRAC IP>/redfish/v1/" -k
```

Response:


```
{"@odata.context":"/redfish/v1/$metadata#ServiceRoot","@odata.id":"/redfish/v1",
"@odata.type":"#ServiceRoot.1.0.0.ServiceRoot","AccountService":{"@odata.id":"/r
edfish/v1/Managers/iDRAC.Embedded.1/AccountService"},"Chassis":{"@odata.id":"/re
dfish/v1/Chassis"},"Description":"Root
Service","EventService":{"@odata.id":"/redfish/v1/EventService"},"Id":"RootServi
ce","JsonSchemas":{"@odata.id":"/redfish/v1/JSONSchemas"},"Links":{"Sessions":{"
@odata.id":"/redfish/v1/Sessions"}},"Managers":{"@odata.id":"/redfish/v1/Manager
s"},"Name":"Root
Service","RedfishVersion":"1.0.0","Registries":{"@odata.id":"/redfish/v1/Registr
ies/Messages/En"},"SessionService":{"@odata.id":"/redfish/v1/SessionService"},"S
ystems":{"@odata.id":"/redfish/v1/Systems"},"Tasks":{"@odata.id":"/redfish/v1/Ta
skService"}}
```


For cleaner output formatting, the JSON output can be passed to Python and the module json.tool (output limited for brevity):

Command:

```
curl "https://<iDRAC IP>/redfish/v1/" -k | python -m json.tool
```

Response:


```
{
    "@odata.context": "/redfish/v1/$metadata#ServiceRoot",
    "@odata.id": "/redfish/v1",
    "@odata.type": "#ServiceRoot.1.0.0.ServiceRoot",
    "AccountService": {
        "@odata.id": "/redfish/v1/Managers/iDRAC.Embedded.1/AccountService"
    },
    "Chassis": {
        "@odata.id": "/redfish/v1/Chassis" …
```


3.2.1 Using cURL with authentication

As discussed above, Redfish supports Basic and Session based authentication. Here are the cURL commands used for authentication operations:

Basic authentication: Drilling down further into the Redfish API will require authentication. This can be done using “-u username:password” on the cURL command line:

```
curl "https://<iDRAC IP>/redfish/v1/Chassis" -k -u root:calvin
```

This authenticates the single operation using the provided credentials.

Session based authentication: This requires a two stage process, creating a session called Login and deleting session called Logout. For login, run the following command in verbose mode by appending option –v; this will output the X-Auth-Token value for use in the subsequent commands:

```
curl -k -X POST -d '{"UserName":"root","Password":"calvin"}' https://<iDRAC IP>/redfish/v1/Sessions -v
```

Once logged in, subsequent commands can be sent using X-Auth-Token as shown below:

```
curl -k https://<iDRAC IP>/redfish/v1/Chassis -v --header "X-Auth-Token: <X-Auth-Value>"
```

```
curl -k https://<iDRAC IP>/redfish/v1/Systems -v --header "X-Auth-Token: <X-Auth-Value>"
```

After executing the desired commands, the session can be terminated using Logout:

```
curl -k -X DELETE https://<iDRAC IP>/redfish/v1/Sessions/6 -v --header "X-Auth-Token: <X-Auth-Value>"
```

3.3 Accessing Redfish by using Python scripting

One of the goals of the Redfish API is to enable easy access from common scripting languages such as Python. The following Python script examples implement key Redfish API use cases. In these examples, it is assumed that the Python HTTP "requests" library (found here: http://docs.python-requests.org/en/master/) has been downloaded. Also, note that the examples assume the credentials of "root"/"calvin" for access to the Redfish API – these should be changed to the appropriate credentials for a specific iDRAC target.

These Python examples as well as example PowerShell cmdlets are available as open source on Github: https://github.com/dell/iDRAC-Redfish-Scripting

3.3.1 View general system information and status

In this use case, the SSL/TLS certificate check is skipped by setting "verify=False". Note that the "RAID.Integrated.1-1" path may differ between systems depending on the type of RAID controller installed. Therefore, it is recommended to navigate the API path starting from the root.

```
import requests
import json

system = requests.get('https://<iDRAC
IP>/redfish/v1/Systems/System.Embedded.1',verify=False,auth=('root','calvin'))
storage = requests.get('https://<iDRAC
IP>/redfish/v1/Systems/System.Embedded.1/Storage/Controllers/RAID.Integrated.1-
1',verify=False,auth=('root','calvin'))

systemData = system.json()
storageData = storage.json()

print "Model:           {}".format(systemData[u'Model'])
```

```
print "Manufacturer:    {}".format(systemData[u'Manufacturer'])
print "Service tag      {}".format(systemData[u'SKU'])
print "Serial number:   {}".format(systemData[u'SerialNumber'])
print "Hostname:        {}".format(systemData[u'HostName'])
print "Power state:     {}".format(systemData[u'PowerState'])
print "Asset tag:       {}".format(systemData[u'AssetTag'])
print "Memory size:     {}".format(systemData[u'MemorySummary']
                                        [u'TotalSystemMemoryGiB'])
print "CPU type:        {}".format(systemData[u'ProcessorSummary'][u'Model'])
print "Number of CPUs:  {}".format(systemData[u'ProcessorSummary'][u'Count'])
print "System status:   {}".format(systemData[u'Status'][u'Health'])
print "RAID health:     {}".format(storageData[u'Status'][u'Health'])
```

Output:

```
Model:           PowerEdge R740
Manufacturer:    Dell Inc.
Service tag      BN7----
Serial number:   CN7-----------
Hostname:        WIN-GHLELBK2V2M
Power state:     On
Asset tag:
Memory size:     64.0
CPU type:        Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Number of CPUs:  2
System status:   OK
RAID health:     OK
```

3.3.2 View system health across multiple servers

This example displays the service tag and overall system status for multiple systems; the source servers are identified by a file consisting of iDRAC IP addresses and FQDNs as input.

```
import requests
import json

with open("serverList.txt", "r") as serverList:
    for server in serverList.readlines():
        req = requests.get("https://" + server.rstrip() +
"/redfish/v1/Systems/System.Embedded.1",verify=False,auth=('root','calvin'))
        reqJson = req.json()
        print "System {}: Health status:
{}".format(reqJson[u'SKU'],reqJson[u'Status'][u'Health'])
```

3.3.3 View system event log

```
import requests
import json

system = requests.get('https://<iDRAC
IP>/redfish/v1/Managers/iDRAC.Embedded.1/Logs/Sel',verify=False,
auth=('root','calvin'))

systemData = system.json()

for logEntry in systemData[u'Members']:
    print "{}: {}".format(logEntry[u'Name'],logEntry[u'Created'])
    print " {}\n".format(logEntry[u'Message'])
```

Output (shortened for brevity):

```
Log Entry 93: 2016-02-26T09:35:55+09:00
The chassis is closed while the power is off.

Log Entry 92: 2016-02-26T09:35:50+09:00
The chassis is open while the power is off.

Log Entry 91: 2015-09-24T14:04:59+09:00
OEM software event.

Log Entry 90: 2015-09-24T14:04:59+09:00
C: boot completed.
```

3.3.4 Check system power state

This script displays the current server power state - on or off.

```
import requests
import json

response = requests.get('https://<iDRAC
IP>/redfish/v1/Systems/System.Embedded.1',verify=False,auth=('root','calvin'))
data = response.json()
print data[u'PowerState']
```

3.3.5 Turn on a system

Server power is controlled by using a POST operation to the **ComputerSystem.Reset** URI to request the desired action.

```
import requests
import json

url = 'https://<iDRAC
IP>/redfish/v1/Systems/System.Embedded.1/Actions/ComputerSystem.Reset'
payload = {'ResetType': 'On'}
headers = {'content-type': 'application/json'}

response = requests.post(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))
```

3.3.6 Turn off a system

```
import requests
import json

url = 'https://<iDRAC
IP>/redfish/v1/Systems/System.Embedded.1/Actions/ComputerSystem.Reset'
payload = {'ResetType': 'ForceOff'}
headers = {'content-type': 'application/json'}

response = requests.post(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))
```

3.3.7 View system power usage

Use this script to view current, average, minimum, and maximum system power consumption.

```
import requests
import json

system = requests.get('https://<iDRAC
IP>/redfish/v1/Chassis/System.Embedded.1/Power/PowerControl',verify=False,
auth=('root','calvin'))

systemData = system.json()

print "Consumed power:      {}".format(systemData[u'PowerConsumedWatts'])
print "Average reading:
      {}".format(systemData[u'PowerMetrics'][u'AverageConsumedWatts'])
print "Max reading:
      {}".format(systemData[u'PowerMetrics'][u'MaxConsumedWatts'])
```

```
print "Min reading:
      {}".format(systemData[u'PowerMetrics'][u'MinConsumedWatts'])
```

Output:

```
Consumed power:      149
Average reading:     155
Max reading:         169
Min reading:         144
```

3.3.8 Update general System Information with PATCH operation

This example demonstrates how to update the System read/write properties as defined in the Redfish specification. Users can provide information about the properties in the System and can update single or multiple properties. If the user provides invalid information for updating a property due to an invalid data type or unacceptable data, the iDRAC Redfish service provides extended information along with an error indication.

```
import requests
import json

url = 'https://<iDRAC IP>/redfish/v1/Systems/System.Embedded.1'
payload = {' Hostname ': ' Ubuntu ' }
headers = {'content-type': 'application/json'}

response = requests.patch(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))

print "Status Code:      {}".format(response.status_code)
print "Extended Info Message:      {}".format(response.json())
```

Output:

```
Status Code:      200
Extended Info Message:      {u'Success': {u'Message': u'Successfully Completed
Request', u'Resolution': u'None', u'Severity': u'Ok', u'MessageId':
u'Base.1.0.Success'}}
```

Using the above script, if the user provides an incorrect type for the input data as shown below, they will receive extended information specifying the error. For example, if the payload was improperly input as:

`payload = {' Hostname ': 1234 }` this would result in the following output:

Output:

```
Status Code:      400

Extended Info Message:      {u'error': {u'code': u'Base.1.0.GeneralError',
u'message': u'A general error has occurred. See ExtendedInfo for more
information', u'@Message.ExtendedInfo': [{u'Severity': u'Warning', u'MessageId':
u'Base.1.0.PropertyValueTypeError', u'RelatedProperties': [u'HostName'],
u'Message': u'The value integer or boolean for the property HostName is of a
different type than the property can accept.', u'Resolution': u'Correct the
value for the property in the request body and resubmit the request if the
operation failed.', u'MessageArgs': [u'integer or boolean', u'HostName']}]}}
```

If there are internal processing errors for a request, a 500 status code will be returned with an internal error message:

Output:

```
Status Code:      500

Extended Info Message:      {u'error': {u'code': u'Base.1.0.GeneralError',
u'message': u'A general error has occurred. See ExtendedInfo for more
information', u'@Message.ExtendedInfo': [{u'Severity': u'Critical',
u'MessageId': u'Base.1.0.InternalError', u'RelatedProperties': [u'HostName'],
u'Message': u'The request failed due to an internal service error. The service
is still operational.', u'Resolution': u'Resubmit the request. If the problem
persists, consider resetting the service.'}]}}
```

The same approach is implemented for POST and DELETE operations.

3.3.9 Change boot device temporarily

Instructing a server to boot once into BIOS or to boot from an alternate source, such as PXE can be done by modifying the **BootSourceOverrideTarget** value. Note that a PATCH action is required because the script updates the existing boot target value.

```
import requests
import json

url = 'https://<iDRAC IP>/redfish/v1/Systems/System.Embedded.1'
payload = {'Boot': {'BootSourceOverrideTarget': 'BiosSetup'} }
headers = {'content-type': 'application/json'}

response = requests.patch(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))
```

3.3.10 Update / modify iDRAC user account

The iDRAC has pre-defined slots for internal user accounts. To modify an account, a PATCH action is used rather than a POST. “RoleId” is used to specify the type of access permissions to be granted to the user; in this case, the Operator role is used.

```
import requests
import json

url = 'https://<iDRAC
IP>/redfish/v1/Managers/iDRAC.Embedded.1/Accounts/<Account-id>'

plUserName = {'UserName': 'user03'}
plPass = {'Password': 'calvin'}
plRoleId = {'RoleId': 'Operator'}

headers = {'content-type': 'application/json'}

for payload in plUserName,plPass,plRoleId:
    response = requests.patch(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))
```

Note: You can configure up to 16 local users in iDRAC with specific access permissions. Before you create an iDRAC user, verify if any current users exist. You can set user names, passwords, and roles with the privileges for these users. User 1 is reserved for the IPMI anonymous user and you cannot change this configuration. By default, User 2 is the “root” user.

Note: iDRAC local users are deleted by setting the user name to NULL.

3.3.11 Redfish 2016 for 12th, 13th and 14th generation PowerEdge servers

New for the 14th generation of PowerEdge servers, the iDRAC9 supports Redfish 2016 features including:

- BIOS configuration including set attributes, change boot order, enable/disable boot device state;
- Secure boot and iDRAC configuration
- Firmware inventory and streamed local updates. To perform a streamed update, the firmware image must be stored locally on the system where the Redfish update API is executed.

iDRAC9 includes enhancements to the iDRAC RESTful API for Server Configuration Profiles (SCP) support and iDRAC configuration including:

- Firmware update via a networked repository during SCP import
- Auto Config, RACADM, WS-Man and Redfish SCP operations via HTTP/HTTPS in addition to CIFS and NFS
- SCP operations via local file streaming
- SCP JSON format for export / import in addition to XML format

iDRAC7/8 also support Redfish 2016 and iDRAC RESTful API enhancements beginning with firmware version 2.50.50.50. That support includes:

- BIOS configuration – set attributes, only
- Secure boot configuration
- Firmware update via a networked repository during SCP import
- SCP operations via local file streaming
- SCP JSON format for export / import in addition to XML format

For more information about RESTful server configuration, see the Dell EMC technical white paper *Zero-Touch Bare Metal Server Provisioning using Dell EMC iDRAC with Lifecycle Controller Auto Config*, available on the Dell Techcenter.

3.3.12 View and Configure BIOS Attributes

iDRAC7/8 firmware 2.50.50.50 or later and iDRAC9 firmware 3.00.00.00 or later implement the Redfish 2016 API for BIOS configuration. Here is a script to view all BIOS attributes and a script to change a single BIOS attribute.

```
#
# redfish_get_bios_attribute_settings.py
# Get BIOS attributes and current settings
# Print to STDOUT and save to file "bios_attributes.txt"
# Synopsis:
#   redfish_get_bios_attribute_settings.py <iDRAC IP addr> <user> <password>
#
import requests, json, sys, re, time, os

try:
    idrac_ip = sys.argv[1]
    idrac_username = sys.argv[2]
    idrac_password = sys.argv[3]
except:
    print "- FAIL: You must pass in script name along with iDRAC IP / iDRAC
username / iDRAC password"
    sys.exit()

try:
    os.remove("bios_attributes.txt")
except:
    pass
#
# Function to get BIOS attributes /current settings
#
def get_bios_attributes():
    f=open("bios_attributes.txt","a")
    global current_value
```

```
    global pending_value
    response =
requests.get('https://%s/redfish/v1/Systems/System.Embedded.1/Bios' %
idrac_ip,verify=False,auth=(idrac_username,idrac_password))
    data = response.json()
    a="\n--- BIOS Attributes ---\n\n%-30s%-30s\n\n" % ("Attribute", "Value")
    print a
    f.writelines(a)
    for i in data[u'Attributes'].items():
        attribute_name = "%-30s" % (i[0])
        #print attribute_name
        f.writelines(attribute_name)
        attribute_value = "%-30s\n" % (i[1])
        #print attribute_value
        f.writelines(attribute_value)
        print "%-30s%-30s" % (i[0],i[1])

    print "\n- Attributes are also captured in \"bios_attributes.txt\" file"
    f.close()

# Run Code

get_bios_attributes()
```

Output (shortened for brevity):

```
--- BIOS Attributes ---

Attribute                     Value

NodeInterleave                Disabled
IscsiDev1Con1EnDis            Disabled
MemFrequency                  MaxPerf
HttpDev3EnDis                 Disabled
SataPortADriveType            Unknown Device
MemPatrolScrub                Standard
SataPortNDriveType            Unknown Device
FailSafeBaud                  115200
MeFailureRecoveryEnable       Enabled
SystemBiosVersion             1.0.0
WriteDataCrc                  Disabled
MemoryRmt                     Disabled
WriteCache                    Disabled
IscsiDev1Con2EnDis            Disabled
SysProfile                    PerfPerWattOptimizedDapc
…
```

```
#
# redfish_set_One_bios_attribute.py
# Set a single BIOS attributes to a new value
# Synopsis:
#   redfish_set_one_bios_attribute.py <iDRAC IP> <user> <password>
#                                     <Attribute name> <New value>
#
import requests, json, sys, re, time

from datetime import datetime

try:
    idrac_ip = sys.argv[1]
    idrac_username = sys.argv[2]
    idrac_password = sys.argv[3]
    attribute_name = sys.argv[4]
    pending_value = sys.argv[5]
except:
    print "- FAIL: You must pass in script name along with iDRAC IP / iDRAC
username / iDRAC password / attribute name / attribute value. Example:
\"script_name.py 192.168.0.120 root calvin MemTest Enabled\""
    sys.exit()

### Function to get BIOS attribute current value

def get_attribute_current_value():
    global current_value
    response =
requests.get('https://%s/redfish/v1/Systems/System.Embedded.1/Bios' %
idrac_ip,verify=False,auth=(idrac_username, idrac_password))
    data = response.json()
    current_value = data[u'Attributes'][attribute_name]
    if current_value == pending_value:
        answer = raw_input("\n- WARNING, %s is already set to %s, do you still
want to set the attribute? Type (y) or (n): " % (attribute_name, current_value))
        if answer == "n":
            sys.exit()
        else:
            pass

### Function to set BIOS attribute pending value

def set_bios_attribute():
    print "\n- WARNING: Current value for %s is: %s, setting to: %s\n" %
(attribute_name, current_value, pending_value)
    time.sleep(2)
```

```
    url = 'https://%s/redfish/v1/Systems/System.Embedded.1/Bios/Settings' %
idrac_ip
    payload = {"Attributes":{attribute_name:pending_value}}
    headers = {'content-type': 'application/json'}
    response = requests.patch(url, data=json.dumps(payload), headers=headers,
verify=False,auth=(idrac_username, idrac_password))
    statusCode = response.status_code
    if statusCode == 200:
        print "\n- PASS: Command passed to set BIOS attribute %s pending value
to %s\n" % (attribute_name, pending_value)
    else:
        print "\n- FAIL, Command failed, errror code is %s" % statusCode
        detail_message=str(response.__dict__)
        print detail_message
        sys.exit()
    d=str(response.__dict__)

### Function to create BIOS target config job

def create_bios_config_job():
    global job_id
    url = 'https://%s/redfish/v1/Managers/iDRAC.Embedded.1/Jobs' % idrac_ip
    #payload = {"Target":"BIOS.Setup.1-1","RebootJobType":"PowerCycle"}
    payload =
{"TargetSettingsURI":"/redfish/v1/Systems/System.Embedded.1/Bios/Settings"}
    headers = {'content-type': 'application/json'}
    response = requests.post(url, data=json.dumps(payload), headers=headers,
verify=False,auth=(idrac_username, idrac_password))
    statusCode = response.status_code
    #print "Status Code: {0}".format(response.status_code)
    #print "Extended Info Message: {0}".format(response.json())
    if statusCode == 200:
        print "\n- PASS: Command passed to create target config job, status code
200 returned.\n"
    else:
        print "\n- FAIL, Command failed, status code is %s\n" % statusCode
        detail_message=str(response.__dict__)
        print detail_message
        sys.exit()
    d=str(response.__dict__)
    z=re.search("JID_.+?,",d).group()
    job_id=re.sub("[,']","",z)
    print "- WARNING: %s job ID successfully created\n" % job_id

### Function to verify job is marked as scheduled before rebooting the server

def get_job_status():
```

```
    while True:
        req =
requests.get('https://%s/redfish/v1/Managers/iDRAC.Embedded.1/Jobs/%s' %
(idrac_ip, job_id), auth=(idrac_username, idrac_password), verify=False)
        statusCode = req.status_code
        if statusCode == 200:
            print "\n- PASS, Command passed to check job status, code 200
returned\n"
            time.sleep(20)
        else:
            print "\n- FAIL, Command failed to check job status, return code is
%s" % statusCode
            print "Extended Info Message: {0}".format(req.json())
            sys.exit()
        data = req.json()
        if data[u'Message'] == "Task successfully scheduled.":
            print " JobID = "+data[u'Id']
            print " Name = "+data[u'Name']
            print " Message = "+data[u'Message']
            print " PercentComplete = "+str(data[u'PercentComplete'])+"\n"
            break
        else:
            print "\n- WARNING: JobStatus not scheduled, current status is:
%s\n" % data[u'Message']

### Function to reboot the server

def reboot_server():
    url =
'https://%s/redfish/v1/Systems/System.Embedded.1/Actions/ComputerSystem.Reset' %
idrac_ip
    payload = {'ResetType': 'ForceOff'}
    headers = {'content-type': 'application/json'}
    response = requests.post(url, data=json.dumps(payload), headers=headers,
verify=False, auth=(idrac_username,idrac_password))
    statusCode = response.status_code
    if statusCode == 204:
        print "\n- PASS, Command passed to power OFF server, code return is
%s\n" % statusCode
    else:
        print "\n- FAIL, Command failed to power OFF server, status code is:
%s\n" % statusCode
        print "Extended Info Message: {0}".format(response.json())
        sys.exit()
    time.sleep(10)
    payload = {'ResetType': 'On'}
    headers = {'content-type': 'application/json'}
```

```
    response = requests.post(url, data=json.dumps(payload), headers=headers,
verify=False, auth=('root','calvin'))
    statusCode = response.status_code
    if statusCode == 204:
        print "\n- PASS, Command passed to power ON server, code return is %s\n"
% statusCode
    else:
        print "\n- FAIL, Command failed to power ON server, status code is:
%s\n" % statusCode
        print "Extended Info Message: {0}".format(response.json())
        sys.exit()

### Function to loop checking the job status until marked completed or failed

def loop_job_status():
    start_time=datetime.now()
    while True:
        req =
requests.get('https://%s/redfish/v1/Managers/iDRAC.Embedded.1/Jobs/%s' %
(idrac_ip, job_id), auth=(idrac_username, idrac_password), verify=False)
        current_time=(datetime.now()-start_time)
        statusCode = req.status_code
        if statusCode == 200:
            print "\n- PASS, Command passed to check job status, code 200
returned\n"
        else:
            print "\n- FAIL, Command failed to check job status, return code is
%s" % statusCode
            print "Extended Info Message: {0}".format(req.json())
            sys.exit()
        data = req.json()
        if str(current_time)[0:7] >= "0:30:00":
            print "\n- FAIL: Timeout of 30 minutes has been hit, script
stopped\n"
            sys.exit()
        elif "Fail" in data[u'Message'] or "fail" in data[u'Message']:
            print "- FAIL: %s failed" % job_id
            sys.exit()
        elif data[u'Message'] == "Job completed successfully.":
            print "\n JobID = "+data[u'Id']
            print " Name = "+data[u'Name']
            print " Message = "+data[u'Message']
            print " PercentComplete = "+str(data[u'PercentComplete'])+"\n"
            break
        else:
```

```
            print "- WARNING, JobStatus not completed, current status is:
\"%s\", current job polling time is: %s\n" %
(data[u'Message'],str(current_time)[0:7])
            time.sleep(30)

### Function to check attribute new current value

def get_new_current_value():
    response =
requests.get('https://%s/redfish/v1/Systems/System.Embedded.1/Bios' %
idrac_ip,verify=False,auth=(idrac_username, idrac_password))
    data = response.json()
    current_value_new = data[u'Attributes'][attribute_name]
    if current_value_new == pending_value:
        print "\n- PASS, BIOS attribute \"%s\" new current value is: %s" %
(attribute_name, pending_value)
    else:
        print "n\- FAIL, BIOS attribute \"%s\" attribute not set to: %s" %
(attribute_name, current_value)
        sys.exit()

### Run code

get_attribute_current_value()
set_bios_attribute()
create_bios_config_job()
get_job_status()
reboot_server()
loop_job_status()
get_new_current_value()
```

Output:

```
% redfish_set_one_bios_attribute.py 100.65.99.66 root calvin  MemTest Enabled

- WARNING: Current value for MemTest is: Disabled, setting to: Enabled

- PASS: Command passed to set BIOS attribute MemTest pending value to Enabled

- PASS: Command passed to create target config job, status code 200 returned.

- WARNING: JID_956505296759 job ID successfully created

- PASS, Command passed to check job status, code 200 returned

 JobID = JID_956505296759
 Name = ConfigBIOS:BIOS.Setup.1-1
```

```
 Message = Task successfully scheduled.
 PercentComplete = 0

- PASS, Command passed to power OFF server, code return is 204

- PASS, Command passed to power ON server, code return is 204

- PASS, Command passed to check job status, code 200 returned

- WARNING, JobStatus not completed, current status is: "Task successfully
scheduled.", current job polling time is: 0:00:00

- PASS, Command passed to check job status, code 200 returned

(Edited for bevity…)

- WARNING, JobStatus not completed, current status is: "Job in progress.",
current job polling time is: 0:02:35

- PASS, Command passed to check job status, code 200 returned

 JobID = JID_956505296759
 Name = ConfigBIOS:BIOS.Setup.1-1
 Message = Job completed successfully.
 PercentComplete = 100

- PASS, BIOS attribute "MemTest" new current value is: Enabled
```

3.3.13 Viewing server firmware inventory

14th generation PowerEdge servers implement the Redfish API for a detailed inventory of the installed server firmware. The following script produces a report of the currently installed server firmware.

```
#
# redfish_get_FW_inventory.py
#   Get and print the current inventory of a server's firmware
# Synopsis:
#   redfish_get_FW_inventory.py <iDRAC IP addr> <user> <password>
#
import requests, json, sys, re, time, os, ConfigParser, logging

from datetime import datetime

try:
    idrac_ip = sys.argv[1]
    idrac_username = sys.argv[2]
    idrac_password = sys.argv[3]
```

```
except:
    print "\n- FAIL, you must pass in script name along with iDRAC IP / iDRAC
username / iDRAC password. Example: \"script_name.py 192.168.0.120 root
calvin\""
    sys.exit()

    print "\n- Getting current firmware version(s) for all devices in the system
iDRAC supports\n"
    time.sleep(3)
    req = requests.get('https://%s/redfish/v1/UpdateService/FirmwareInventory/'
% (idrac_ip), auth=(idrac_username, idrac_password), verify=False)
    statusCode = req.status_code
    data = req.json()
    number_of_devices=len(data[u'Members'])
    count = 0
    installed_devices=[]
    while count != len(data[u'Members']):
        a=data[u'Members'][count][u'@odata.id']
        a=a.replace("/redfish/v1/UpdateService/FirmwareInventory/","")
        if "Installed" in a:
            installed_devices.append(a)
        count +=1
    installed_devices_details=["\n--- Firmware Inventory ---"]
    a="-"*75
    installed_devices_details.append(a)
    l=[]
    ll=[]
    for i in installed_devices:
        req =
requests.get('https://%s/redfish/v1/UpdateService/FirmwareInventory/%s' %
(idrac_ip, i), auth=(idrac_username, idrac_password), verify=False)
        statusCode = req.status_code
        data = req.json()
        a="Name: %s" % data[u'Name']
        l.append(a.lower())
        installed_devices_details.append(a)
        a="Firmware Version: %s" % data[u'Version']
        ll.append(a.lower())
        installed_devices_details.append(a)
        a="Updateable: %s" % data[u'Updateable']
        installed_devices_details.append(a)
        a="-"*75
        installed_devices_details.append(a)

    for i in installed_devices_details:
        print i
```

Output:

```
C:\Python26>redfish_get_FW_inventory.py 100.65.99.66 root calvin
- Getting current firmware version(s) for all devices in the system iDRAC
supports

--- Firmware Inventory ---
-------------------------------------------------------------------------
Name: PERC H330 Mini
Firmware Version: 25.5.2.0001
Updateable: True
-------------------------------------------------------------------------
Name: OS COLLECTOR, 3.0, A00
Firmware Version: 3.0
Updateable: True
-------------------------------------------------------------------------
Name: Disk 1 in Backplane 1 of Integrated RAID Controller 1
Firmware Version: TT31
Updateable: True
-------------------------------------------------------------------------
Name: Disk 0 in Backplane 1 of Integrated RAID Controller 1
Firmware Version: TT31
Updateable: True
-------------------------------------------------------------------------
Name: Disk 2 in Backplane 1 of Integrated RAID Controller 1
Firmware Version: VT31
Updateable: True
-------------------------------------------------------------------------
Name: Intel(R) Ethernet Converged Network Adapter XL710-Q2 - 3C:FD:FE:15:99:AA
Firmware Version: 18.0.16
Updateable: True
-------------------------------------------------------------------------
Name: BP14G+EXP 0:1
Firmware Version: 2.14
Updateable: True
-------------------------------------------------------------------------
Name: iDRAC Service Module Installer, 3.0.1, A00
Firmware Version: 3.0.1
Updateable: True
-------------------------------------------------------------------------
Name: Power Supply.Slot.1
Firmware Version: 00.23.32
Updateable: True
-------------------------------------------------------------------------
Name: BIOS
Firmware Version: 1.0.0
Updateable: True
```

```
----------------------------------------------------------------------------
Name: Dell OS Driver Pack, 17.05.21, A00
Firmware Version: 17.05.21
Updateable: True
----------------------------------------------------------------------------
Name: Integrated Dell Remote Access Controller
Firmware Version: 3.00.00.00
Updateable: True
----------------------------------------------------------------------------
Name: Dell 64 Bit uEFI Diagnostics, version 4301, 4301X07, 4301.8
Firmware Version: 4301X07
Updateable: True
----------------------------------------------------------------------------
Name: QLogic 577xx/578xx 10 Gb Ethernet BCM57800 - 18:66:DA:8E:28:26
Firmware Version: 08.07.00
Updateable: True
----------------------------------------------------------------------------
Name: System CPLD
Firmware Version: 1.0.0
Updateable: True
----------------------------------------------------------------------------
Name: Lifecycle Controller
Firmware Version: 3.00.00.00
Updateable: False
----------------------------------------------------------------------------
```

3.3.14 Updating server firmware

14th generation PowerEdge servers implement the Redfish 2016 API to update server component firmware. The following script can be used to perform a firmware update using a single Dell Update Package such as a BIOS update, a PERC firmware update, or as shown in the example, a Diagnostics update.

```
#
# redfish_single_device_update.py
#   Update a single server component's firmware
# Synopsis:
#   redfish_single_device_update.py <iDRAC IP addr> <user> <password>
#                                   <FW file folder> <FW file name>
#                                   <Install option>
#   Install option - Now: update now, do not reboot,
#                    NowandReboot: update now and reboot,
#                    NextReboot: update at next reboot
#
import requests, json, sys, re, time, os, ConfigParser, logging

from datetime import datetime
```

```
# Validate all correct parameters are passed in

try:
    idrac_ip = sys.argv[1]
    idrac_username = sys.argv[2]
    idrac_password = sys.argv[3]
    firmware_image_location = sys.argv[4]
    file_image_name= sys.argv[5]
    Install_Option = sys.argv[6]
except:
    print "\n- FAIL, you must pass in script name along with iDRAC IP / iDRAC
username / iDRAC password / Image Path / Filename / Install Option. Example: \"
script_name.py 192.168.0.120 root calvin c:\Python26 bios.exe NowAndReboot\""
    sys.exit()

# Convert install option to correct string due to case sensitivity in iDRAC.

if Install_Option == "now":
    install_option = "Now"
elif Install_Option == "nowandreboot":
    install_option = "NowAndReboot"
elif Install_Option == "nextreboot":
    install_option = "NextReboot"
else:
    install_option = Install_Option

# Download the image payload to the iDRAC

def download_image_payload():
    print "\n- WARNING, downloading DUP payload to iDRAC\n"
    global Location
    global new_FW_version
    global dup_version
    req = requests.get('https://%s/redfish/v1/UpdateService/FirmwareInventory/'
% (idrac_ip), auth=(idrac_username, idrac_password), verify=False)
    statusCode = req.status_code
    data = req.json()
    filename = file_image_name.lower()
    ImageLocation = firmware_image_location
    ImagePath = ImageLocation + "\\" + filename
    ETag = req.headers['ETag']
    url = 'https://%s/redfish/v1/UpdateService/FirmwareInventory' % (idrac_ip)
    files = {'file': (filename, open(ImagePath, 'rb'), 'multipart/form-data')}
    headers = {"if-match": ETag}
    response = requests.post(url, files=files, auth = (idrac_username,
idrac_password), verify=False, headers=headers)
    d = response.__dict__
```

```
    if response.status_code == 201:
        print "\n- PASS: Command passed, 201 status code returned\n"
        z=re.search("\"Message\":.+?,",d['_content']).group().rstrip(",")
        z=re.sub('"',"",z)
        print "- %s" % z
    else:
        print "\n- FAIL: Post command failed to download, error is %s" %
response
        print "\nMore details on status code error: %s " % d['_content']
        sys.exit()
    d = response.__dict__
    z=re.search("Available.+?,",d['_content']).group()
    z = re.sub('[",]',"",z)
    new_FW_version = re.sub('Available','Installed',z)
    zz=z.find("-")
    zz=z.find("-",zz+1)
    dup_version = z[zz+1:]
    entry = "- FW file version is: %s" % dup_version; print entry
    Location = response.headers['Location']

# Install the downloaded image payload and loop checking job status

def install_image_payload():
    global job_id
    print "\n- WARNING, installing downloaded firmware payload to device\n"
    url =
'https://%s/redfish/v1/UpdateService/Actions/Oem/DellUpdateService.Install' %
(idrac_ip)
    InstallOption = install_option
    payload = "{\"SoftwareIdentityURIs\":[\"" + Location +
"\"],\"InstallUpon\":\""+ InstallOption +"\"}"
    headers = {'content-type': 'application/json'}
    response = requests.post(url, data=payload, auth = (idrac_username,
idrac_password), verify=False, headers=headers)
    d=str(response.__dict__)
    job_id_location = response.headers['Location']
    job_id = re.search("JID_.+",job_id_location).group()
    print "\n- PASS, %s job ID successfully created\n" % job_id
    #time.sleep(20)

# Check the new firmware version installed

def check_new_FW_version():
    print "\n- WARNING, checking new firmware version installed for updated
device\n"
```

```
    req =
requests.get('https://%s/redfish/v1/UpdateService/FirmwareInventory/%s' %
(idrac_ip, new_FW_version), auth=(idrac_username, idrac_password), verify=False)
    #print req
    statusCode = req.status_code
    data = req.json()
    if dup_version == data[u'Version']:
        print "\n- PASS, New installed FW version is: %s" % data[u'Version']
    else:
        print "\n- FAIL, New installed FW incorrect, error is: %s" % data
        sys.exit()

# Check the job status for host reboot needed

def check_job_status_host_reboot():
    # Loop get command to check the job status of completed, completed with
errors or failed
    start_time=datetime.now()
    time.sleep(15)
    while True:
        req = requests.get('https://%s/redfish/v1/TaskService/Tasks/%s' %
(idrac_ip, job_id), auth=(idrac_username, idrac_password), verify=False)
        statusCode = req.status_code
        data = req.json()
        message_string=data[u"Messages"]
        current_time=(datetime.now()-start_time)
        if statusCode == 202 or statusCode == 200:
            print "\n- Query job ID command passed\n"
            time.sleep(10)
        else:
            print "Query job ID command failed, error code is: %s" % statusCode
            sys.exit()
        if "failed" in data[u"Messages"] or "completed with errors" in
data[u"Messages"]:
            print "- FAIL: Job failed, current message is: %s" %
data[u"Messages"]
            sys.exit()
        elif data[u"TaskState"] == "Completed":
            print "\n- Job ID = "+data[u"Id"]
            print "- Name = "+data[u"Name"]
            try:
                print "- Message = "+message_string[0][u"Message"]
            except:
                print data[u"Messages"][0][u"Message"]
            print "- JobStatus = "+data[u"TaskState"]
            print "\n- %s completed in: %s" % (job_id, str(current_time)[0:7])
            break
```

```
        elif data[u"TaskState"] == "Completed with Errors" or data[u"TaskState"]
== "Failed":
            print "\n- Job ID = "+data[u"Id"]
            print "- Name = "+data[u"Name"]
            try:
                print "- Message = "+message_string[0][u"Message"]
            except:
                print "- "+data[u"Messages"][0][u"Message"]
            print "- JobStatus = "+data[u"TaskState"]
            print "\n- %s completed in: %s" % (job_id, str(current_time)[0:7])
            sys.exit()
        else:
            print "- Job not marked completed, current status is: %s" %
data[u"TaskState"]
            print "- Message: %s\n" % message_string[0][u"Message"]
            print "- Current job execution time is: %s\n" %
str(current_time)[0:7]
            time.sleep(1)
            continue

# Run code here

download_image_payload()
install_image_payload()
if install_option == "NowAndReboot" or install_option == "Now":
    check_job_status_host_reboot()
    check_new_FW_version()
else:
    check_job_status()
```

Output:

```
C:\Python26>redfish_single_device_update.py <iDRAC9 IP address> root calvin
c:\Python26 Diagnostics_Application_JF9MW_WN64_4301X06_4301.7.EXE Now

- WARNING, downloading DUP payload to iDRAC

- PASS: Command passed, 201 status code returned

- Message: Package successfully downloaded.
- FW file version is: 4301X06

- WARNING, installing downloaded firmware payload to device

- PASS, JID_956512038576 job ID successfully created

- Query job ID command passed
```

```
- Job ID = JID_956512038576
- Name = Firmware Update: Diagnostics
- Message = Job completed successfully.
- JobStatus = Completed

- JID_956512038576 completed in: 0:00:15

- WARNING, checking new firmware version installed for updated device

- PASS, New installed FW version is: 4301X06

C:\Python26>
```

3.3.15 Extended information

When errors occur during operations, Redfish provides Extended Information detailing the error. iDRAC includes a Dell EMC-provided Message Registry, accessible by the **MessageId** returned as part of the Extended Information, that gives direction for resolution of the error.

The following is a Python scripting example illustrating Extended Information:

```
import requests
import json

url = 'https://<iDRAC
IP>/redfish/v1/Managers/iDRAC.Embedded.1/SerialInterfaces/iDRAC.Embedded.1%23Ser
ial.1'
payload = {'BitRate':19200}
headers = {'content-type':'application/json'}

response =
requests.patch(url,data=json.dumps(payload),headers=headers,verify=False,auth=('
root','calvin'))

print "Status Code:{}".format(response.status_code)
print "Extended Error Message:{}".format(response.json())
```

The above script updates the property **BitRate** to the value 19200, a valid setting.

Output:

```
Status Code:      200
Extended Error Message:{u'Success': {u'Message': u'Successfully Completed
Request', u'Resolution': u'None', u'Severity': u'Ok', u'MessageId':
u'Base.1.0.Success'}}
```

If the script is modified to attempt an update with an unsupported value, an error will occur and Extended Information will be returned:

```
payload = {'BitRate':1900}
```

Output:

```
Status Code:      400
Extended Error Message:      {u'error': {u'code': u'Base.1.0.GeneralError',
u'message': u'A general error has occurred. See ExtendedInfo for more
information', u'@Message.ExtendedInfo': [{u'Severity': u'Warning', u'MessageId':
u'Base.1.0.PropertyValueNotInList', u'RelatedProperties': [u'BitRate'],
u'Message': u'The value 1900 for the property BitRate is not in the list of
acceptable values.', u'Resolution': u'Choose a value from the enumeration list
that the implementation can support and resubmit the request if the operation
failed.', u'MessageArgs': [u'1900', u'BitRate']}]}}
```

The following example uses an incorrect data type – a text string rather than a numeric value. This illustrates the distinctive error information and resolutions provided by Extended Information and the Dell Message Registry.

```
payload = {'BitRate':'19200'}
```

Output:

```
Status Code:400
Extended Error Message:      {u'error': {u'code': u'Base.1.0.GeneralError',
u'message': u'A general error has occurred. See ExtendedInfo for more
information', u'@Message.ExtendedInfo': [{u'Severity': u'Warning', u'MessageId':
u'Base.1.0.PropertyValueTypeError', u'RelatedProperties': [u'BitRate'],
u'Message': u'The value string or boolean for the property BitRate is of a
different type than the property can accept.', u'Resolution': u'Correct the
value for the property in the request body and resubmit the request if the
operation failed.', u'MessageArgs': [u'string or boolean', u'BitRate']}]}}
```

4 Summary

The DMTF Redfish standard is emerging as a key new tool for efficient, scalable, and secure server management. Utilizing an industry-standard interface and data format, Redfish supports rapid development of automation for one-to-many server management. System administrators and IT developers will appreciate Redfish's features that can increase efficiency, lower costs and boost productivity across their organizations.

Dell EMC is a committed leader in the development and implementation of open, industry standards. Supporting Redfish within the iDRAC with Lifecycle Controller further enhances the manageability of PowerEdge servers, providing another powerful tool to help IT administrators reduce complexity while increasing the efficiency of their operations.

5 Additional Information

- DMTF white papers, Redfish Schemas, specifications, webinars and work-in-progress documents

  https://www.dmtf.org/standards/redfish

- The Redfish standard specification is available from the DMTF website

  http://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.0.1.pdf

- Open source iDRAC REST API with Redfish Python and PowerShell examples

  https://github.com/dell/iDRAC-Redfish-Scripting

- The iDRAC with Lifecycle Controller home page on Dell TechCenter provides access to product documents, technical white papers, how-to videos and more

  http://en.community.dell.com/techcenter/systems-management/w/wiki/3204

- JSON lightweight data interchange format

  http://www.json.org/

- OData4 open protocol standard for the definition and exchange of information using RESTful APIs

  http://www.odata.org/

5.1 Acronyms

API	Application Programming Interface
BMC	Baseboard Management Controller
DMTF	Distributed Management Task Force
DSP	DMTF Standard Publications
FQDD	Fully Qualified Device Descriptor
HTTP	Hyper Text Transfer Protocol
HTTPS	HTTP Secure or HTTP over TLS/SSL
iDRAC	Integrated Dell Remote Access Controller
IPMI	Intelligent Platform Management Interface
JSON	Java Script Object Notation
LC	Lifecycle Controller
OData	Open Data Protocol
OOB	Out-of-Band
REST	Representational State Transfer
SNMP	Simple Network Management Protocol
SPMF	Scalable Platforms Management Forum
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier

5.2 Definitions

- **cURL**: an open source command line tool and library for transferring data with URL Syntax
- **DMTF**: Distributed Management Task Force, defines management standards supported by numerous hardware, software and service vendors.(www.dmtf.org)
- **Redfish Client**: Name for the functionality that communicates with a Redfish Service and accesses one or more resources or functions of the Service.
- **Event**: A record that corresponds to an individual alert.
- **Subscription**: The act of registering a destination for the reception of events.
- **Notification**: One-way message sent to indicate that an event has occurred
- **Redfish Event Listener**: The name for the functionality that receives alerts from a Redfish Service. This functionality is typically software running on a remote system that is separate from the managed system.