

Hewlett Packard
Enterprise

Managing Hewlett Packard Enterprise Servers Using the RESTful API

Abstract

This guide describes how to get started using the iLO RESTful (iLO 4) and Moonshot iLO Chassis Management Module RESTful API to manage Hewlett Packard Enterprise servers.

Part Number: 795538-006
Published: August 2016
Edition: 1

© Copyright 2016 Hewlett Packard Enterprise Development LP

The information contained herein is subject to change without notice. The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Hewlett Packard Enterprise shall not be liable for technical or editorial errors or omissions contained herein.

Confidential computer software. Valid license from Hewlett Packard Enterprise required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Links to third-party websites take you outside the Hewlett Packard Enterprise website. Hewlett Packard Enterprise has no control over and is not responsible for information outside the Hewlett Packard Enterprise website.

Acknowledgments

Microsoft® and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Google Inc.© All rights reserved. Google™ is a trademark of Google Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Contents

1 Introduction to the RESTful API....5
2 Redfish 1.0 conformance....6
3 REST APIs architected using HATEOAS....7
Key benefits of the RESTful API....7
Resource operations....7
Return codes....8
4 Tips for using the RESTful API....9
RESTful Interface Tool and Python Examples....9
Example REST API operation....9
CURL example....10
Example HTTP response headers....10
Example Root JSON object....11
Type and resource version....12
OneView Information....12
Links....12
Navigating the data model....12
Basic Authentication example....14
Adding basic authentication using Postman....14
Adding Basic Authentication to a CURL command line....15
Session management....15
Navigating the data model....16
Collections....16
Computer system....18
Chassis....19
Manager....19
Example of how to iterate all the Manager resources in the data model....19
Schemas....20
Methods for finding message registries and schema....20
Offline .zip file in Service Pack for ProLiant....20
Handling Schema Version: Forward and backward compatibility....20
5 Examples using the iLO RESTful API....22
Accessing the iLO RESTful API using Python....22
BIOS....22
About BIOS....22
Changing settings and understanding SettingsResults....23
The benefits of using two separate resources....23
Reading BIOS Defaults example....23
BIOS resources and attribute registry overview....24
Attribute registry example....24
BIOS attributes....25
BIOS attribute registries....26
Updating the Administrator BIOS password example....26
Updating the BIOS settings example....26
Enabling BIOS UEFI Secure Boot example....27
Reverting BIOS UEFI settings to default example....27
Example iSCSI Software Initiator configuration....28
Boot settings....30
UEFI boot structured name string....30
UEFI boot structured name string examples....30
Change UEFI boot order example....32

BIOS administrator password considerations....33
Example reset all BIOS and boot order settings to factory defaults....33
Power....34
Reset a server example....34
6 Error messages and registries in the iLO RESTful API....35
7 Best practices for creating clients and avoiding incorrect assumptions....36
8 RESTful Events and the Event Service....38
Subscribing for Events examples....38
9 Troubleshooting....40
Resetting the RESTful API using a 3rd party REST web client....40
Resetting the iLO RESTful API using the RESTful Interface Tool....40
Resetting the iLO RESTful API using the Embedded UEFI Shell restclient command....41
Resetting the iLO RESTful API using the iLO SSH CLI....42
Unusual characters in JSON when Subscribing to an Alert EventType....42
10 iLO RESTful API functionality for Gen9 servers....43
11 Support and other resources....45
Accessing Hewlett Packard Enterprise Support....45
Accessing updates....45
Websites and documents....46
Customer self repair....46
Remote support....47
Documentation feedback....47
A Warranty and regulatory information....48
Warranty information....48
Regulatory information....48
Belarus Kazakhstan Russia marking....49
Turkey RoHS material content declaration....50
Ukraine RoHS material content declaration....50

1 Introduction to the RESTful API

The iLO RESTful API (iLO 4) and Moonshot iLO Chassis Management Module RESTful API is a programming interface enabling state-of-the-art server management. This document contains helpful information about how to interact with the RESTful API. The RESTful API uses the basic HTTP operations (`GET`, `PUT`, `POST`, `DELETE`, and `PATCH`) to submit or return a JSON formatted resource to or from a URI on iLO 4 or Moonshot iLO Chassis Management Module.

With modern scripting languages, you can easily write simple REST clients for RESTful APIs. Most languages, like Python, can transform JSON into internal-data structures, like dictionaries, allowing for easy access to data. This enables you to write custom code directly to the RESTful API, instead of using intermediate tools such as HPE's HPQLOCFG or CONREP.

This document has been updated with examples from iLO 4 version 2.30 firmware.

2 Redfish 1.0 conformance

The RESTful API was first released with iLO 4 2.00 on HPE Gen9 servers. The RESTful API also functioned as the starting point for the new Redfish 1.0 DMTF standard (see **http://www.dmtf.org/standards/redfish**).

Since the introduction of the RESTful API, a number of changes were introduced into the Redfish standard by the DMTF SPMF members. At a high level, the changes include:

- Use of OData annotations throughout the data to communicate meta-data
- Revision of Error structures
- Removal or Rename of certain JSON properties
- Use of different collection schema
- Use of a different set of entry-point URIs (`/redfish/v1/` vs. `/rest/v1`)

iLO 4 2.30 is Redfish 1.0 conformant while remaining backward compatible with the existing RESTful API. Moving forward, the RESTful API will become the iLO 4 enhanced implementation of Redfish. While conforming to the Redfish 1.0 standard, the RESTful API is extended with Hewlett Packard Enterprise-specific features such as BIOS configuration. You should plan to update your client code to confirm to the Redfish standard. iLO will eventually remove any properties that do not match the Redfish schema definitions (while preserving the OEM extensions.)

iLO 4 2.30 achieves Redfish 1.0 conformance and backward compatibility by:

1. Mirroring the resource model at both `/redfish/v1/` and `/rest/v1`.
2. Returning both compatibility and Redfish properties by default.
3. Returning only Redfish conformant properties (with Hewlett Packard Enterprise extensions) if the Redfish-required OData header is included in the request (OData-Version: 4.0).

3 REST APIs architected using HATEOAS

Representational State Transfer (REST) is a web service that uses basic CRUD (Create, Read, Update, Delete, and Patch) operations performed on resources using HTTP commands such as `POST`, `GET`, `PUT`, `PATCH`, and `DELETE`. The RESTful API is designed using a REST architecture called HATEOS (Hypermedia as the Engine of Application State). This architecture allows the client to interact with iLO through a simple fixed URL (`rest/v1`) and several other top-level URIs documented in the iLO Data Model. The rest of the data model is discoverable by following clearly identified “links” in the data. This has the advantage that the client does not need to know a set of fixed URLs. When you create a script to automate tasks using the RESTful API, you only need to hardcode this simple URL and design the script to discover the REST API URLs that are needed to complete a task. To learn more about REST and HATEOAS concepts, see:

- **http://en.wikipedia.org/wiki/Representational_state_transfer**
- **http://en.wikipedia.org/wiki/HATEOAS**

Key benefits of the RESTful API

The RESTful API is becoming the main management interface for iLO 4 and Moonshot iLO Chassis Management Module-based Hewlett Packard Enterprise servers. Its feature set will become larger than the existing iLO XML API (RIBCL) and IPMI interfaces. Using the RESTful API, you can take full inventory of the server, control power and reset, configure BIOS and iLO settings, fetch event logs, as well as many other functions.

The RESTful API follows the trend of the Internet in moving to a common pattern for new software interfaces. Many web services in a variety of industries use REST APIs because they are easy to implement, easy to consume, and offer scalability advantages over previous technologies. HPE OneView, OpenStack, and many other server management APIs are now REST APIs. Most Hewlett Packard Enterprise Management software offerings, as well as the entire Software Defined Infrastructure, are built upon REST APIs.

The RESTful API has the additional advantage of consistency across all present and projected server architectures. The same data model works for traditional rack-mount servers, blades, as well as newer types of systems like Moonshot. This advantage comes because the data model is designed to self-describe the service’s capabilities to the client and has room for flexibility designed in from the start.

Resource operations

Operation	HTTP Verb	Description
Create	`POST` resource URI (payload = resource data)	Creates a new resource or invokes a custom action. A synchronous `POST` returns the newly created resource.
Read	`GET` resource URI	Returns the requested resource representation.
Update	`PATCH` or `PUT` resource URI (payload = update data)	Updates an existing resource. You can only `PATCH` properties that are marked `readonly = false` in the schema.
Delete	`DELETE` resource URI	Deletes the specified resource.

Return codes

Return code	Description
2*xx*	Successful operation.
4*xx*	Client-side error with error message returned.
5*xx*	iLO error with error message returned.

NOTE: If an error occurs, indicated by a return code 4*xx* or 5*xx*, an `ExtendedError` JSON response is returned. The expected resource is not returned.

4 Tips for using the RESTful API

The iLO RESTful API is available on ProLiant Gen9 servers running iLO 4 2.00 or later with the iLO Standard license, although some features in the data might not be available without an Advanced license. The RESTful API for Moonshot iLO Chassis Management Module is available on Moonshot servers running iLO Chassis Manager 1.30 or later and does not require a license.

To access the RESTful API, you need an HTTPS-capable client, such as a web browser with the Postman REST Client plugin extension or CURL (a popular command line HTTP utility).

RESTful Interface Tool and Python Examples

Although not a requirement, you can use the RESTful Interface Tool with the RESTful API. This command line tool provides a level of abstraction and convenience above direct access to the RESTful API. For details see: **http://www.hpe.com/info/restfulapi**.

Also, Hewlett Packard Enterprise published example Python code that implements a number of common operations in a RESTful API client. This code can be downloaded at **https://github.com/HewlettPackard/python-proliant-sdk**. In some cases the examples in this document may refer to examples in the Python code with this notation:

Python: See `ex1_functionname()` in the Python example code. This means look for the specified function name in the python example code.

If you prefer not to implement a client in Python, this serves as a good pseudocode implementing the logic required to perform an operation.

Example REST API operation

Let's perform our first `GET` operation using the RESTful API. We will do an HTTP `GET` on the iLO HTTPS port, typically port 443 (although it could be different if you have previously configured iLO to use another port). Your client should be prepared to handle the HTTPS certificate challenge. The interface is not available over open HTTP (port 80), so you must use HTTPS.

Our `GET` operation will be against a resource at `/rest/v1` (without a trailing slash):

- Correct: `GET` **`https://myilo/rest/v1`**
- Incorrect: `GET` **`https://myilo/rest/v1/`**

It is best to perform this initial `GET` with a tool like the CURL or the Postman REST Client mentioned above. Later you will want to do this with your own scripting code, but for now it's useful to see the HTTP header information exchanged using a browser.

NOTE: This tutorial contains pseudocode for accessing resources in the iLO RESTful API. To best follow this code, we highly recommend you perform the operations in the Postman REST client or CURL so you can easily see how the data structures match the pseudocode. The same principles illustrated here also apply to Moonshot iLO Chassis Management Module.

Redfish: Redfish changes the entry point URI to `/redfish/v1/`.

The trailing slash must be included and iLO returns an HTTP redirect status (308) to the client if the clients accesses the following URIs:

- `/redfish`
- `/redfish/`
- `/redfish/v1`

In the future, iLO 4 will respond to all `/rest/v1/` style URIs with 308 redirect to `/redfish/v1/` equivalent URIs.

CURL example

CURL is a command line utility available for many Operating Systems that enables easy access to the RESTful API. CURL is available at **http://curl.haxx.se/**. Note that all the CURL examples will use a flag `-insecure`. This causes CURL to bypass validation of the HTTPS certificate. In real use iLO should be configured to use a user-supplied certificate and this option is not necessary. Notice also that we use the `-L` option to force CURL to follow HTTP redirect responses. If iLO changes URI locations for various items, it can indicate to the client where the new location is and automatically follow the new link.

```
> curl https://myilo/rest/v1 -i --insecure -L
-i returns HTTP response headers, --insecure bypasses TLS/SSL certification verification, -L follows redirects
```

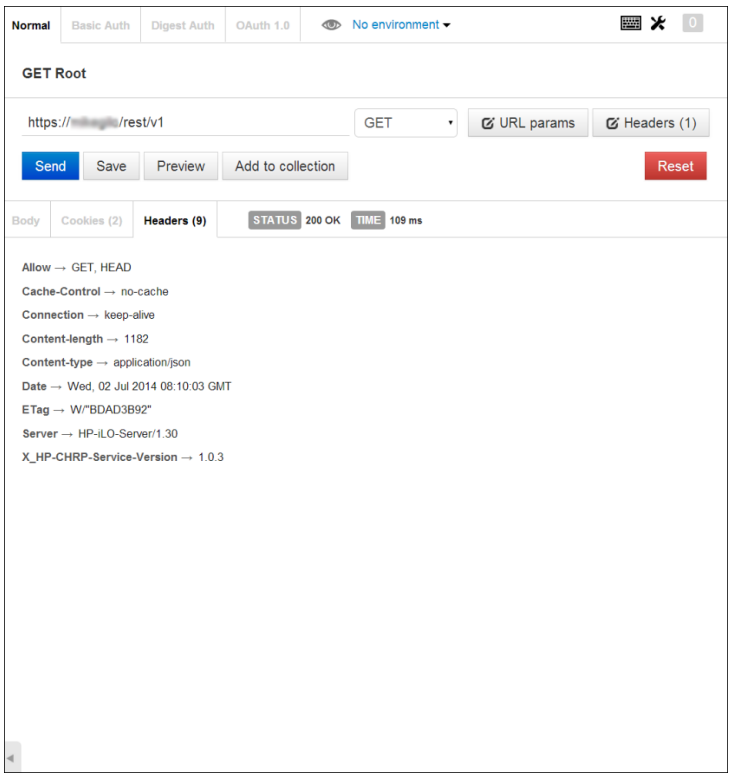
```
{"@odata.context":"/redfish/v1/$metadata#ServiceRoot","@odata.id":"/redfish/v1/","@odata.type":"#ServiceRoot.1.0.0.ServiceRoot",

 "Id":"v1","Name":"HP RESTful Root
Service","Oem":{},"ServiceVersion":"0.9.5","Time":"2014-08-05T13:12:02Z","Type":"ServiceRoot.1.0.0",
"UUID":"627490fe-bded-5d10-8176-0af0927c690e","links":{"AccountService":{"href":"/rest/v1/AccountService"},"Chassis":{"href":"/rest/v1/Chassis"},
"EventService":{"href":"/rest/v1/EventService"},"JSONSchema":{"href":"/rest/v1/Schemas"},"Managers":{"href":"/rest/v1/Managers"},
"Registries":{"href":"/rest/v1/Registries"},"Schemas":{"href":"/rest/v1/Schemas"},"SessionService":{"href":"/rest/v1/SessionService"},
"Sessions":{"href":"/rest/v1/SessionService/Sessions"},"Systems":{"href":"/rest/v1/Systems"},"self":{"href":"/rest/v1"}}}
```

Example HTTP response headers

When you perform a GET operation, you receive a response JSON object. Take a look at this JSON object and the HTTP headers returned above.

The first lines are the response HTTP headers identifying the content. Following that is the JSON response in compressed form. Even though it looks messy, JSON is highly structured and can be “pretty-printed” into easy to read documents.



The first detail to notice is a couple of the response headers. The `ETag` header is a digest of the JSON content. `ETag` handing is an advanced topic, so we will ignore it for now, except to say that the string value of an `ETag` is not intended to be parsed by the client. It is an opaque string and iLO might change its `ETag` generation algorithm in a future version.

The second interesting header is the `Allow` header. Allow is supplied on all `GET` operations by the RESTful API and indicates which HTTP operations are allowed on this resource. You will

notice that only `GET` is allowed against `/rest/v1`. This means it is a read-only object because you cannot use `PATCH`, `PUT`, or `DELETE` (the modification operations.) On many resources, you will discover that the interface supports `PATCH`. `PATCH` is specifically intended to update an existing object instead of replacing it (like `PUT` would.) This enables you to `PATCH` a partial object and not worry that you are removing any properties you omitted.

Example Root JSON object

Here is the full version of the above JSON response (edited for length):


```
{
  "@odata.context": "/redfish/v1/$metadata#ServiceRoot",
  "@odata.id": "/redfish/v1/",
  "@odata.type": "#ServiceRoot.1.0.0.ServiceRoot",
  "Id": "v1",
  "Name": "HP RESTful Root Service",
  "Oem": {},
  "ServiceVersion": "0.9.5",
  "Time": "2014-08-05T13:12:02Z",
  "Type": "ServiceRoot.1.0.0",
  "UUID": "627490fe-bded-5d10-8176-0af0927c690e",
  "links": {
    "AccountService": {
      "href": "/rest/v1/AccountService"
    },
    "Chassis": {
      "href": "/rest/v1/Chassis"
    },
    "EventService": {
      "href": "/rest/v1/EventService"
    },
    "JSONSchema": {
      "href": "/rest/v1/Schemas"
    },
    "Managers": {
      "href": "/rest/v1/Managers"
    },
    "Registries": {
      "href": "/rest/v1/Registries"
    },
    "Schemas": {
      "href": "/rest/v1/Schemas"
    },
    "SessionService": {
      "href": "/rest/v1/SessionService"
    },
    "Sessions": {
      "href": "/rest/v1/SessionService/Sessions"
    },
    "Systems": {
      "href": "/rest/v1/Systems"
    },
    "self": {
      "href": "/rest/v1"
    }
  }
}
```


In JSON, there is no strong ordering of property names, so iLO may return JSON properties in any order. Likewise, iLO cannot assume the order of properties in any submitted JSON. This is why the best scripting data structure for a RESTful client is a dictionary: a simple set of unordered key/value pairs. This lack of ordering is also the reason you see embedded structure within objects (objects within objects). This allows us to keep related data together that is more logically organized, aesthetically pleasing to view, and helps avoid property name conflicts or ridiculously long property names. It also allows us to use identical blocks of JSON in many places in the data model, like status.

Type and resource version

Notice that the JSON response has a property named `Type` with a value of `"ServiceRoot.1.0.0"`. Type is very important in the RESTful API. It is the key to understanding what all of the other properties in the object mean. Resources that have the same type follow the same *schema definition* (meaning that like-named properties mean the same thing.)

Type includes a name (`"ServiceRoot"`) as well as version information (`"1.0.0"`). Version information is formatted as `major.minor.errata`. Future iLO firmware releases will certainly expand the data included in its various resources, so as a client, you should be prepared to see different versions of a `Type`. Anything but a major version is backward compatible, so a new property added to an object might result in minor `Type` increments (for example, `Chassis.1.0.0` becomes `Chassis.1.1.0`). Moving to a major version change (for example, `Chassis.2.0.0`) is a breaking change without any guarantee of backward compatibility.

The initial releases of the RESTful API contained many resources of type version 0.9.5 where major version is 0. However, as of iLO 4 2.30 many types are transitioning to version 1.0.0. We have maintained backward compatibility with previous versions of iLO 4, so this type changes from 0 to 1 do not indicate breaking changes in this case but a maturing of the RESTful API data model.

Redfish: In Redfish, `Type` has been replaced with `@odata.type`. In a few cases, the type name string changed as well (e.g. `PowerMetrics` is now simply `Power`). Note that if you include the OData-Version header specified in the Redfish specification, you will not see a Type property in the response but only the `@odata.type`. Without this header you will see both. In the future when iLO removes non-Redfish content, `Type` will be removed leaving `@odata.type` as the type indicator.

OneView Information

If the server is managed by OneView, information about the OneView instance is included in this root resource. See the property at JSON pointer path `/rest/v1#/Oem/Hp/Manager/0/IPManager`. The content of this sub-object includes a reference back to the OneView server.

NOTE: If the server is managed by OneView, many of the settings are specifically configured for OneView use. Use care in modifying any properties in the RESTful API directly when the server is managed by OneView as it might result in the server being out of sync with OneView's view of the system for some time.

Links

Finally, let's take a look at the links property. In the RESTful API, the navigation of the data is included in the data itself, not in the spec, and can therefore adapt as needed to different servers over time without the specification changes. This is called a hypermedia API where the navigation is built into the data (like web pages). The links property contains the pointers to other related resources. Inside the links objects are various defined relationship types, like Systems, Managers, Chassis, or Sessions. Each of these contain a property called `href` that is the URI of the related resource. A relationship type (for example, `Systems`) distinguishes the various links by what they navigate to. For instance, Systems navigates to a collection of `ComputerSystem` resources.

Navigating the data model

The RESTful API does not define all of the URIs to various items, like temperatures or power supplies on a server. You cannot assume the BIOS version information is always at a particular URI. This is more complex for the client, but is necessary to make sure the data model can change to accommodate various future server architectures without requiring specification changes. As an example, if the BIOS version is at `/rest/v1/Systems/1`, and a client assumed it is always there, the client would then break when the interface is implemented on a Moonshot with 180 compute nodes, each with its own BIOS version. Defining pointers to specific features

in the specification makes the RESTful API too rigid. For this reason, only a select few URIs are published and guaranteed to be stable starting points. Client code must not assume anything about the URIs that you see in the data model. You must treat them as opaque strings or your client will not interoperate with other implementations of the RESTful API.

Do not hardcode URI parsing templates in your client. Instead, you should start at /rest/v1 or one of the fixed entry point URIs below and crawl the data model using the defined relationship types and find instances of types that interest you.

Redfish: Redfish uses the `@odata.id` property to indicate a link to another resource. These links may be in a `"Links"` section (for related resources) or in the root object itself (for child resources.) Note that Redfish uses a `"Links"` (capital L) vs. `"links"` for the compatibility RESTful API. iLO will render either `Links` or `links` based upon the presence or absence of the OData-Version HTTP header in the request.

Fixed Entry Point URIs in the Data Model

The following URIs are fixed in the data model and client software may start with any of these as it accesses the RESTful API.

- `/rest/v1`

  The root resource.

- `/rest/v1/Systems`

  The collection of compute nodes.

- `/rest/v1/Chassis`

  The collection of chassis.

- `/rest/v1/Managers`

  The collection of management processors (iLO).

- `/rest/v1/Sessions`

  The Session management API.

- `/rest/v1/Registries`

  The collection of Registries.

- `/rest/v1/Schemas`

  The collection of Schema.

Let's look at some of the links:

Systems

This is a link to the system node collection. Systems are compute nodes (the terminology came from DMTF) with, for example, CPUs, memory, expansion slots, power management, and BIOS version. A DL or BL server has a single compute node in the Systems collection while Moonshot might have up to 180 items in the collection.

You should think of the System link as your portal into the logical view of the server or servers. The collection of Systems is documented to be `/rest/v1/Systems`.

Chassis

This is the physical view. A `Chassis` should be thought of as a physical container. The `Chassis` relationship type is a link to the collection of `Chassis`. On a ProLiant DL/ML/BL server, this will be a collection of one chassis. A Moonshot `Chassis` collection might include the cartridges, as well as the enclosure (as cartridges are a type of chassis.)

`Chassis` are containers that often have power supplies, temperature sensors, and have a physical location. `Chassis` contain logical computer nodes (systems) but can also contain another chassis (consider: a Moonshot chassis contains cartridges that are another kind of chassis, each containing nodes). Also, there is no 1:1 relationship between chassis and systems to accommodate multi-node systems and systems that spread across more than one chassis.

The collection of Chassis is documented to be `/rest/v1/Chassis`.

Managers

This is a link into the iLO 4 or Moonshot iLO Chassis Management Module itself for iLO management tasks traditionally done with RIBCL. For example, network settings, license administration, and iLO firmware management.

The collection of Chassis is documented to be `/rest/v1/Managers`.

Basic Authentication example

The following shows the error displayed on `GET /rest/v1/Systems` when no authentication is attempted:

A Look at ExtendedError/ExtendedInfo Responses

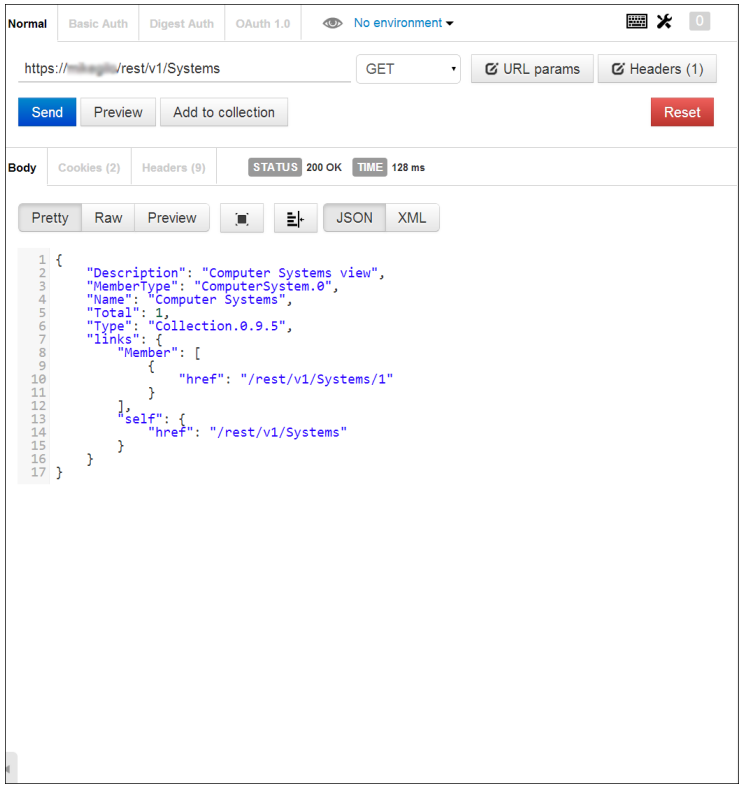
```
{
    "@odata.type": "#ExtendedInfo.ExtendedInfo",
    "Messages": [
        {
            "MessageID": "Base.0.10.NoValidSession",
        }
    ],
    "Type": "ExtendedError.1.0.0",
    "error": {
        "@Message.ExtendedInfo": [
            {
                "MessageId": "Base.0.10.NoValidSession"
            }
        ],
        "code": "iLO.0.10.GeneralError",
        "message": "A general error has occurred. See ExtendedInfo for more information."
    }
}
```

Redfish: The above response is the combination of compatibility and Redfish response. The Redfish properties are **highlighted**. If the request includes the OData-Version header, the non-Redfish properties will be hidden. Also, the response is compatibility type "`ExtendedError`", but is Redfish type "`ExtendedInfo`".

Adding basic authentication using Postman

1. Click the **Basic Auth** tab.
2. Enter the **Username** and **Password**.
3. Click **Refresh headers**.

   This generates the Basic Authentication Base64 hash.
4. Click **Send**.



Adding Basic Authentication to a CURL command line

- Add the `-u username:password` parameter.

Example

```
> curl https://myilo/rest/v1/Systems -i --insecure -u username:password -L
```

```
HTTP/1.1 200 OK
Allow: GET, HEAD
Cache-Control: no-cache
Content-length: 437
Content-type: application/json
Date: Tue, 05 Aug 2014 14:39:56 GMT
ETag: W/"9349EA93"
Link: </rest/v1/SchemaStore/en/ComputerSystemCollection.json>; rel=describedby
Server: HP-iLO-Server/1.30
X-Frame-Options: sameorigin
X_HP-CHRP-Service-Version: 1.0.3

{"@odata.context":"/redfish/v1/$metadata#Systems","@odata.id":"/redfish/v1/Systems/"
,"@odata.type":"#ComputerSystemCollection.ComputerSystemCollection","Description":"Computer Systems view",
"MemberType":"ComputerSystem.1","Members":[{"@odata.id":"/redfish/v1/Systems/1/"}],"Members@odata.count":1,
"Name":"Computer Systems","Total":1,"Type":"Collection.1.0.0","links":{"Member":[{"href":"/rest/v1/Syst
ems/1"}],"self":"/rest/v1/Systems"}}
```

Session management

If you perform a `GET` on any other resource other than the root `/rest/v1` resource, you receive an HTTP 401 (Forbidden) error indicating that you don't have the authentication needed to access the resource.

The RESTful API allows you to use HTTP Basic Authentication if you wish for one-time operations. This is one reason the Postman plugin or CURL is so useful. These tools can easily insert the correct base64 encoding for basic authentication credentials into your requests.

Logging In and Out

For more complex multi-resource operations, you should log in and establish a session. To log in, iLO has a session manager object at the documented URI `/rest/v1/Sessions`. To create a session we need to `POST` a JSON object to the Session manager:

```
POST /rest/v1/Sessions with the required HTTP headers and a body containing

   {
     "UserName": "username",
     "Password": "password"
   }
```

IMPORTANT: You must include the HTTP header `Content-Type: application/json` for all RESTful API operations that include a request body in JSON format.

If the session is created successfully, you receive an HTTP 201 (Created) response from iLO. There will also be two additional HTTP headers.

Header	Value	Description
X-Auth-Token	Your session token (string).	This is a unique string for your login session. It must be included as a header in all subsequent HTTP operations in the session.
Location	The URI of the newly created session resource.	iLO allocates a new session resource describing your session. This is the URI that you must `DELETE` against in order to log out. If you lose this location URI, you can find it by *crawling* the HREF links in the Sessions collection. Store this URI to facilitate logging out.

Navigating the data model

Once you can log in, you may now traverse the entire data model. Because the model is linked together by href links, it is easy to create a client that can crawl the entire model by simply iterating on each object (with recursion for sub-objects and arrays) looking for hrefs within links objects.

Note that even though you start at a well-known URI, the data model is not, strictly speaking, a tree because there can be cross references throughout the data model (for example, from chassis to system and vice versa), so you will need to build a dictionary of visited URIs as you crawl the model to avoid infinite traversal. As you `GET` a URI, add it to this dictionary, and then if you see that URI again, do not perform another `GET`. As additional advice, you should keep the visited dictionary URI keys as lowercase, for example, `tolower()`. URIs in the data model should use consistent lowercase and uppercase conventions, but bugs happen, and if there is inconsistency in casing, keeping the dictionary keyed as lowercase URI strings protects the traversal algorithm.

As you crawl the data model, you will notice a large number of objects are of type `Collection`.

Collections

Collections are a frequently used type of object in the RESTful API. Collections always have a generic type of *Collection* (which means they all conform to the same schema and share the same property names). Collections are very versatile and are distinguished by a property called `MemberType` that indicates the type of the members it collects. The `MemberType` only specifies down to the type name and major version, allowing mixing of minor versions of the same type in the same collection.

Collections are very flexible in implementation and can present members in several different ways. The two basic data structures in a collection: the links to the members of the collection

(`/links/Member` array) and the `Items` array. Depending on how the collection is implemented, you might see either the `Members` or `Items` array omitted from a collection:

- **Form 1**

  Links to Child Resources

  - Contains `/links/Member` array.
  - `/links/Member` array has hrefs to child resources.
  - Collection operations (if supported):

    `GET`

    `POST` to create

- **Form 2**

  Links to Child Resources & Embedded Items

  - Contains `/links/Member` array.
  - `/links/Member` array has hrefs to child resources.
  - Contains `Items` array.

    Each is full `MemberType` representative (*item peek*).
  - `/links/Member` array has frags to Items elements.
  - Collection operations (if supported):

    `GET`

    `POST` to create

- **Form 3**

  Embedded Items only (read only)

  - The `/links/Member` array might be omitted.
  - Contains `Items` array.

    Each is full `MemberType` representative.
  - The `/links/Member` array having frags to Items elements might be omitted.
  - Collection operations (if supported):

    `GET`

A description of each form follows:

- Form 1 is a fully modifiable collection that requires you to iterate by using the `GET` command for each href in the Member array. This is the most flexible and generic form, but is less efficient because it requires more HTTP operations to fully iterate. Collections of this form might allow `POST` to create items, and the child resources might allow `DELETE` to remove items. The children might allow `PATCH` to modify items. This form of a collection is used with Sessions and Accounts for which creation and deletion of members are supported and iteration of the collection is less important.
- Form 2 is a variant of Form 1 and includes an `Items` array that contains either a subset or the full item representation in the collection object itself, as well as a link to the child resource. If the subset of the member is well chosen by the implementer, you might *peek* at collection members by looking at the top-level information in the `Items` array, and decide to do a

separate `GET` for items of interest (drill down). This increases scalability by enabling the client to better determine which children to `GET`. The `Items` array representation does not need to be fully compliant with the `MemberType` schema because it might omit items to save space. The `Items` array does not replace the child resource, but provides a peek at the key differentiating data.

- Form 3 is for read-only collections (for example, PCI Devices or Slots) where item creation, deletion, or modification is not needed. There are no child resources and there might not be a links section. The `Items` array carries entire schema-compliant members. The advantages are size efficiency, no links section, and no need to iterate using separate `GET` operations. This is ideal for things that are best represented as read-only arrays.

To iterate collection in the RESTful API see `collection()` in the Python example code—this is a Python generator function that returns each collection member using the yield keyword.

Example collection

```
> curl https://myilo/rest/v1/Systems -i --insecure -u username:password -L
```

```
{
    "@odata.context": "/redfish/v1/$metadata#Systems",
    "@odata.id": "/redfish/v1/Systems/",
    "@odata.type": "#ComputerSystemCollection.ComputerSystemCollection",
    "Description": "Computer Systems view",
    "MemberType": "ComputerSystem.1",
    "Members": [
        {
            "@odata.id": "/redfish/v1/Systems/1/"
        }
    ],
    "Members@odata.count": 1,
    "Name": "Computer Systems",
    "Total": 1,
    "Type": "Collection.1.0.0",
    "links": {
        "Member": [
            {
                "href": "/rest/v1/Systems/1"
            }
        ]
    }
}
```

Redfish: Redfish no longer has a generic "`Collection`" type. Collections still all look alike but there is no common type. Instead there is a `ComputerSystemCollection` and a `ChassisCollection`, etc, eliminating the need for the `MemberType` property. The Members array is now in the root of the object and the Total is now replaced by `Members@odata.count`. The Redfish specific properties are **highlighted** and non-Redfish properties are hidden if the client supplies the OData-Version header. The Redfish properties communicate the same information as the compatible properties but in a different form.

Computer system

ComputerSystems represent compute nodes in the data model. A ProLiant DL rack server may have a single compute node while a fully loaded Moonshot chassis may contain 180 compute nodes. For an example of how to iterate all the ComputerSystem resources in the data mode:

Python: See `ex1_change_bios_setting()` in the Python example code. This uses the top-level loop using the collection iteration function with `/rest/v1/Systems`.



Chassis

Chassis represent physical or virtual containers for compute resources. For instance, a ProLiant DL rack server has a single chassis representing the metal container of the server. Chassis resources describe the physical aspects of the server. For an example of how to iterate all the Chassis resources in the data mode:

NOTE: **Python**: See `ex29_get_PowerMetrics_Average()` in the Python example code.

After you `GET` a chassis object, you can see the attributes, including `ChassisType`, shared properties from the system (for example, `AssetTag`), and dimension information.

The `ChassisType` is overloaded to a variety of things, so `Chassis` can be within another `Chassis`. For example, a blade chassis is within a blade enclosure chassis, which is within a rack—also considered a chassis. The `contains` and `contained by` relationships are in the links section of the chassis object.

Manager

The Manager resource represents iLO 4 itself.

After you `GET` the iLO object, including `Manager`, you can see the attributes, including `Model` and `Firmware` information and links to various iLO services. Services include the license service, network service, and iLO NIC information and configuration resources.

Example of how to iterate all the Manager resources in the data model

Python: See `ex7_find_iLO_MAC_address()` in the Python example code.

NOTE: The top level loop using the collection iteration function with `/rest/v1/Managers`.

Schemas

Each resource type has a schema file that defines the form of the object, allowed properties, required properties, types, and other information. The data model uses the draft 4 JSON Schema standard to do this. For more information, see: **http://json-schema.org**. To download a package for Python schema validation see: **https://github.com/Julian/jsonschema**. Packages are available for other languages to validate resources against the corresponding schema. This is part of the strategy to leverage industry-standard tool chains.

Schema defines the *class* of resource and each instance of the class might implement a subset of the available properties. A client must be aware that not every property in the schema is implemented in every instance. Omissions might be the result of one or more of the following:

- The property might not apply to a specific Hewlett Packard Enterprise server model.

NOTE: Each property is identified as `readonly = true` or `readonly = false`. This indicates that a `PATCH` operation can be performed on the property, but only if the resource has an `Allow` header that allows `PATCH`. Two resources might share the same `Type` but have different `Allow` headers, that allow or disallow a `PATCH` operation, even for properties that are `readonly = false`.

Methods for finding message registries and schema

There are two methods, online and offline, for finding the schema defines the data in the RESTful API. Online, means the schema information is available by performing a `GET` operation to iLO. Offline means the schema files are available on the HPE ProLiant Support Pack DVD.

Offline .zip file in Service Pack for ProLiant

The Service Pack for ProLiant (SPP) contains a folder called `hp_restful_api` which contains several ZIP files consisting of the schema and registries of the RESTful API.

Within this .zip file is a structure identical to the structure found in the RESTful interface on iLO except that you begin the search by reading either `/Schemas/index.json` or `/Registries/index.json`. The format of these two `index.json` files matches the `SchemaFile.0.9.5` format of the iLO resources referenced by `/rest/v1.Schemas/href`.

In addition, the SPP contains a .zip file of the published schemas and registries for the UEFI BIOS of each supported ProLiant server. Those .zip files are named as `hp-rest-classes-bios-P89-1.00_07_11_2014.zip`, where the `P89` is the ROM family name (for each ProLiant system), and the `1.00-07_011_2014` is the ROM version and date.

To download the offline schema and registry .zip files, see: **http://www.hpe.com**.

Handling Schema Version: Forward and backward compatibility

Resources in the RESTful API are typed by name and version (e.g. `"ServiceRoot.1.0.0"`). This provides enough information to identify not only the correct schema but also the precise version. There are cases where the precise schema version may not be available either online or offline (for example a BIOS update may use a newer schema than iLO presents in its schema repository.) For this reason it is necessary for client code to perform some form of *best match* when searching schemas. For instance, a resource may be of type `"HpBaseConfigs.1.1.0"` and the only schema available is `"HpBaseConfigs.1.0.3"` or vice versa.

In order to facilitate best match, the following guidance is offered:

- A schema file (offline in a folder) is typically named for the schema with a .json extension, but this is not the official name of the schema. The official name base name of the schema is included in the `title` property of the schema object. Additionally, because of some type

renames in Redfish, sometimes an old name of the schema is available in an `oldtitle` property. For example:

```
"title": "EthernetInterface.1.0.0",
"oldtitle": "EthernetNetworkInterface.0.92.0",
```

- The major version of a schema, along with the name, defines a compatible family of schema. For example, `EthernetInterface.2.x.y` is not necessarily compatible with `EthernetInterface.1.x.y`. The exception is version 1 should be considered forward compatible with version 0 schema because the version 0 to 1 transition is a maturing of the initial versions of the schema.
- Higher minor and errata version numbers indicate forward compatible differences. For example, `EthernetInterface1.1.1` is a superset of `EthernetInterface.1.0.0`, so if a client is only using 1.0.0 property definitions, the client may use the 1.1.1 schema as easily as the 1.0.0 schema. However, if the client needs to use properties new to 1.1.1, the new data is not present in a resource of type 1.0.0.

One possible *best-match* schema algorithm might attempt to calculate the mathematical difference between the version of a resource and all candidate schema, and choose the schema with the smallest difference.

5 Examples using the iLO RESTful API

This section contains examples to complete a specific task using the REST API for iLO.

Accessing the iLO RESTful API using Python

Python is a popular language for REST API development and includes powerful tools for rest queries. The following is a simple example of a python query to the `ServiceRoot` resource:

```
>>> from httplib import HTTPSConnection
>>> from base64 import b64encode
>>> userid='iloUserID'
>>> passwd='iloPassword'
>>> auth='BASIC ' + b64encode(userid + ":" + passwd)
>>> header = {'Authorization': auth}
>>> conn = HTTPSConnection(host='iloHostName', strict=True)
>>> conn.request('GET', '/rest/v1', headers=header)
>>> conn.getresponse().read()
```

This is a simple request example that shows the basic libraries involved and how to properly form the request. There are many additional parameters that can be used to tighten or loosen the security around SSL and the certificate verification process. The httplib documentation provides a more robust picture of the libraries capabilities.

Beyond the built-in libraries like httplib, there are many other popular libraries available for Python to perform REST operations. The requests library is one of the most popular HTTP python libraries available today, and is also well suited for REST API development.

To see a more complete set of how to use Python with the RESTful API, review the set of examples at **https://github.com/HewlettPackard/python-proliant-sdk** .

BIOS

The examples in this section are not applicable to Moonshot servers running iLO Chassis Manager.

Python: See `ex1_change_bios_setting()` in the Python example code this demonstrates finding the BIOS settings and PATCHing new values.

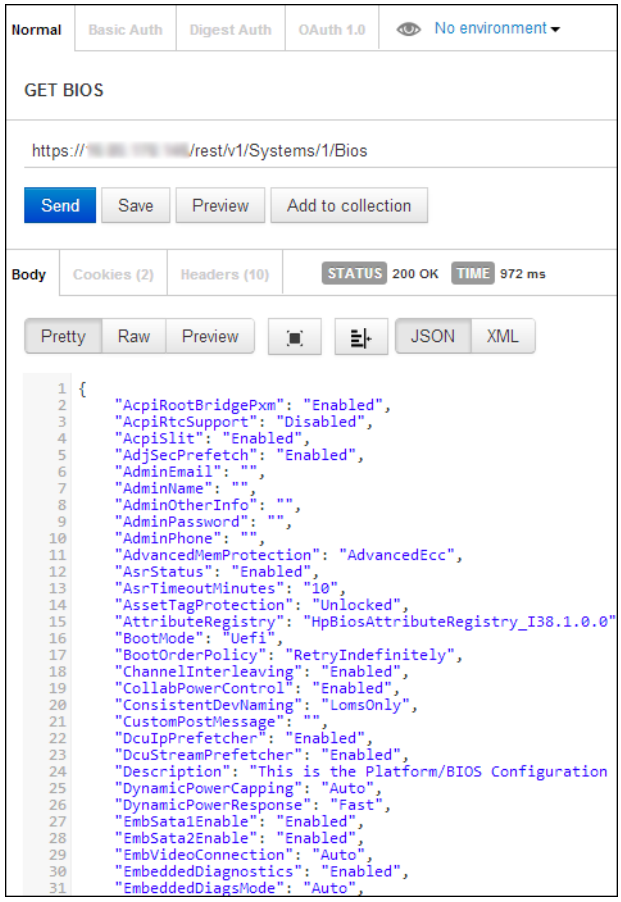
NOTE: The iLO RESTful API can be used to change BIOS settings on ProLiant Gen9 servers or later.

About BIOS

To `GET` the current BIOS configuration:

The iLO RESTful API enables a UEFI BIOS configuration. BIOS is a system-level entity on current architectures. The link to the BIOS configuration is from the computer system node object.

```
SystemCollection = GET /rest/v1/Systems
For each item in SystemCollection.links.Member # this is a collection
  System = GET item.href
  BIOS = GET System.Oem.Hp.links.BIOS.href
```



The result of the `GET` BIOS operation is in a flat object. You cannot use `PATCH` to update a property value. You can only perform `GET` operations on this resource.

Changing settings and understanding `SettingsResults`

The current configuration object for BIOS is read only. This object contains a link to a `Settings` resource that you can perform a `PATCH` operation on, which is the pending settings. If you `GET` the `Settings` resource, you will see that it allows `PATCH` commands. You can change properties and then `PATCH` them back to the `Settings` URI. Changes to settings do not take effect until the server is reset. Before the server is reset, the current and pending settings are independently available. After the server is reset, the pending settings are applied and you can view any errors in the `SettingsResults` property on the main object.

The benefits of using two separate resources

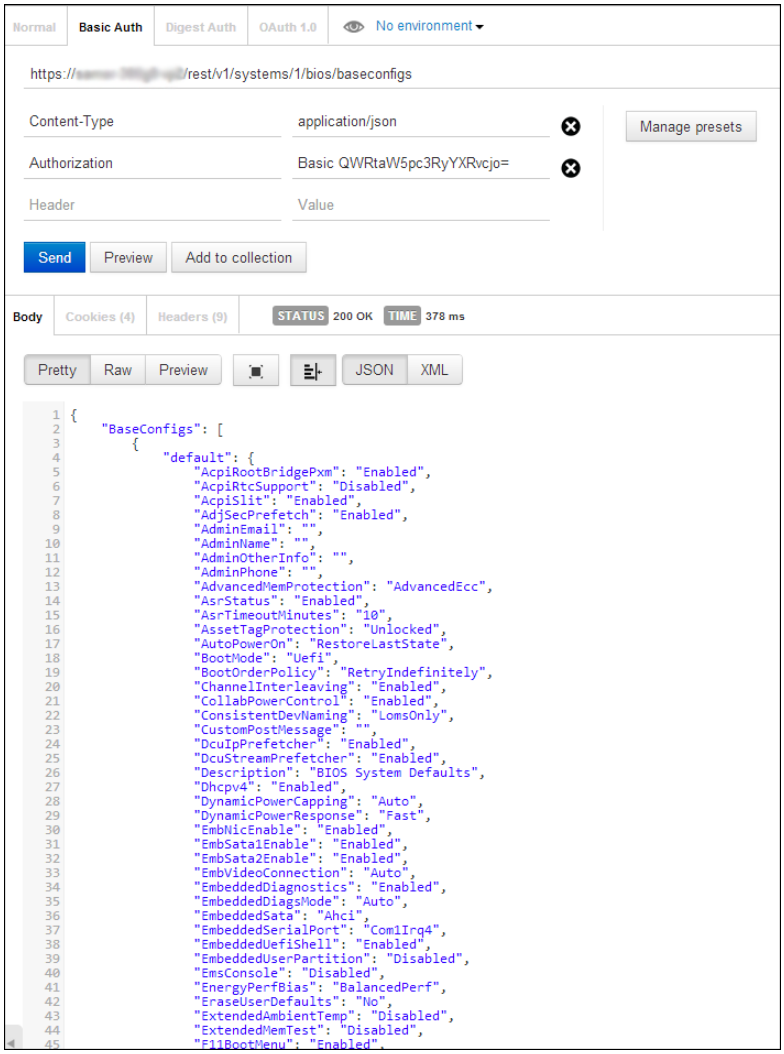
- Enables offline components (for example, BIOS) to process changes to settings in a deferred manner.
- Allows current and pending values to remain available for review until the offline component processes the pending settings.
- Avoids the need for complex job queues.

Reading BIOS Defaults example

The BIOS current configuration object contains a link to a separate read-only object, the `BaseConfigs`, which list the BIOS default settings. To `GET` the BIOS BaseConfigs resource:

```
SystemCollection = GET /rest/v1/Systems
For each item in SystemCollection.links.Member # this is a collection
  System = GET item.href
```

```
BIOS = GET System.Oem.Hp.links.BIOS.href
BaseConfig = GET BIOS.links.BaseConfigs.href
```



Notice that `BaseConfigs` contains an array of default sets (or base configuration sets). Each base config set contains a list of BIOS properties and their default values. The default base config set contains the BIOS manufacturing defaults. It is possible for `BaseConfigs` to contain other sets, like `default.user` for user custom defaults.

BIOS resources and attribute registry overview

The BIOS resources are formatted differently than most other resources. BIOS resources do conform to a schema type as all objects do. However, BIOS settings vary widely across server types and BIOS revisions, so it is extremely difficult to publish a standard schema defining all the possible BIOS setting properties. Furthermore, it is not possible to communicate some of the advanced settings such as inter-setting dependencies, and menu structure in json-schema. Therefore, BIOS uses an *Attribute Registry*.

Attribute registry example

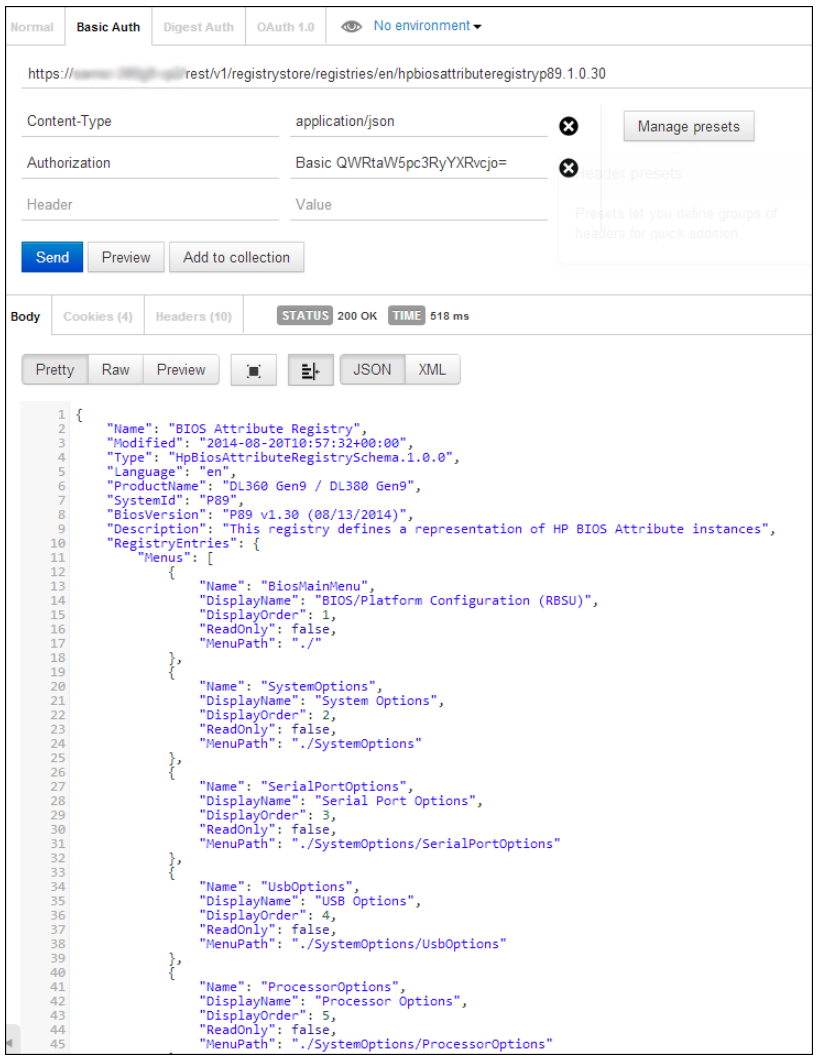
When you `GET` the BIOS configuration object, observe that it still has the basic structure including a `Type`. However, it also has a property called `AttributeRegistry`. This property points to a registry file. This file contains a set of data for each BIOS setting (for example, `PowerProfile`) including menu structure information, any dependencies with other settings, and other information.

To find the BIOS Attribute registry resource:

```
SystemCollection = GET /rest/v1/Systems
For each item in SystemCollection.links.Member # this is a collection
  System = GET item.href
  BIOS = GET System.Oem.Hp.links.BIOS.href
  AttributeRegistryName = the value of the "AttributeRegistry" property in BIOS object

RegistryCollection = GET /rest/v1/Registries
For each item in RegistryCollection.links.Member # this is a collection
  Registry = GET item.href
  If Registry.Schema beginswith AttributeRegistryName
  For each language in Registry.Location
    If language.Language == "en"  # or choose another
      AttributeRegistry = GET language.Uri.extref
```

Due to their size, BIOS Attribute Registries are compressed JSON resources (gzip), so the returned HTTP headers indicate a content-encoding of `gzip`. The REST client will need to decompress the resource. This is done automatically when using a web client (like the Postman plugin).



BIOS attributes

List of BIOS attributes (settings) and their meta-data, including:

- Type of each BIOS attribute (enum, string, numeric, or Boolean).
- Possible values for enum type attributes.
- Display strings (localized to the registry language) for the attributes and their possible values.
- Help text and warning text (localized).
- Location and display order information, including menu hierarchy for an attribute.

- Value limits, including maximum, minimum, and step values for numeric attributes, and minimum and maximum character lengths, as well as regular expressions for string attributes.
- And other meta-data.

BIOS attribute registries

The BIOS attribute registries contains three top-level arrays:

- **Menus**: Array containing the BIOS attributes menus and their hierarchy. This can be used (for instance) to build a user interface that resembles the local BIOS Setup, or to group properties that are related such as `ProcessorOptions` and `UsbOptions`.
- **Dependencies**: Array containing a list of dependencies of BIOS attributes on this server. This includes inter-setting dependencies that might cause one BIOS setting to change its value or its `ReadOnly` property based on the value of another BIOS setting.
- **BaseConfigs**: Array containing a list of default manufacturing settings of BIOS attributes. This is equivalent to reading the BaseConfigs resource and parsing the object named `default`.

Updating the Administrator BIOS password example

To change the Administrator and Power On passwords, you must change two properties for each password:

- Administrator password: Set `AdminPassword` to the new password and `OldAdminPassword` to the old password.
- Power On password: Set `PowerOnPassword` to the new password and `OldPowerOnPassword` with the old password.

If you the old password is not set, you must use a blank string ("") for the old password property.

1. Iterate through `/rest/v1/Systems` and choose a member ComputerSystem.

   Result = *{ilo-ip-address}*`/rest/v1/Systems/1/BIOS`
2. Find a link in the `Oem/Hp/links` called `Bios` and note the `BiosURI`.
3. `GET BiosObj` from `BiosURI` and note that it only allows `GET` (this is the current settings).
4. Find a link in `BiosObj` called `Settings` and note this URI.
5. Create a new JSON object with the `AdminPassword` and `OldAdminPassword` property changed to `{"AdminPassword":"@Pa$$w0rd", "OldAdminPassword":""}`.
6. Update the BIOS settings. You only need to send the updated `AdminName` property in the request body.

   `PATCH` *`{ilo-ip-address}`*`/rest/v1/Systems/1/BIOS/Settings`

   When the server is reset, the BIOS settings are validated and adopted.

Updating the BIOS settings example

The minimum required session ID privileges is `Configure`.

1. Iterate through `/rest/v1/Systems` and choose a member ComputerSystem.

   Result = *{ilo-ip-address}*`/rest/v1/Systems/1/BIOS`
2. Find a link in the `Oem/Hp/links` called `Bios` and note the `BiosURI`.
3. `GET BiosObj` from `BiosURI` and note that it only allows `GET` (this is the current settings).
4. Find a link in `BiosObj` called `Settings` and note this URI.
5. Obtain the BIOS settings using the URI from step 4.

   `GET` *`{ilo-ip-address}`*`/rest/v1/Systems/1/BIOS/Settings`

6. Create a new JSON object with the `AdminName` property changed to `{"AdminName":"Joe Smith"}`.
7. Update the BIOS settings. You only need to send the updated `AdminName` property in the request body.

   `PATCH {ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`
8. Obtain the BIOS settings to verify you made the change to the `AdminName` property.

   `GET {ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`

   When the server is reset, the BIOS settings are validated and adopted.

More information

Python: See `ex1_change_bios_setting()` in the Python example code—this demonstrates finding the BIOS settings and PATCHing new values.

Enabling BIOS UEFI Secure Boot example

The minimum required session ID privileges is `Configure`.

1. Iterate through /rest/v1/Systems and choose a member ComputerSystem. Find a child resource of type `HpSecureBoot` that allows `PATCH` operations (there might be more than one but for this exercise, just choose the first one).

   `{ilo-ip-address}/rest/v1/Systems/1/SecureBoot`
2. Obtain the secure boot settings.

   `GET {ilo-ip-address}/rest/v1/Systems/1/SecureBoot`
3. Create a new JSON object with the `SecureBootEnable` property changed to `{"SecureBootEnable":true}`.
4. Update the secure boot settings. You only need to send the updated `SecureBootEnable` property in the request body.

   `PATCH {ilo-ip-address}/rest/v1/Systems/1/SecureBoot`

   When the sever is reset, the boot settings are validated and adopted.

Reverting BIOS UEFI settings to default example

The BIOS Settings resource supports a special feature that allows you to revert BIOS settings to default for the selected resource. This is accomplished by performing the `PATCH` or `PUT` operation on a special property in the BIOS settings object: `{"BaseConfig": "default"}`. This can be combined with other property sets to first set default values and then set specific settings all in one operation.

NOTE: The `BaseConfig` property might not already exist in the BIOS or BIOS Settings resources. To determine if the BIOS resource supports reverting the settings to default, `GET` the BIOS BaseConfigs resource, and view the `Capabilities` property.

The minimum required session ID privileges is `Configure`.

1. Iterate through `/rest/v1/Systems` and choose a member `ComputerSystem`. Find a child resource of type `HpBios` that allows `PUT` operations (there might be more than one but for this exercise, just choose the first one).

   `{ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`
2. Obtain the BIOS UEFI settings.

   `GET {ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`

3. Change or add the `BaseConfig` property to `{"BaseConfig":"default"}` in the response body.
4. Update the BIOS UEFI settings.

   `PUT {ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`

   When the sever is reset, the BIOS UEFI settings are reverted to default.

NOTE:

- You might also view the default values for BIOS settings by finding the resource type `HpBaseConfigs`.

  `{ilo-ip-address}/rest/v1/Systems/1/BIOS/BaseConfigs`

- `BaseConfig` can be combined with other property values to first reset everything to default and then apply some specific settings in one operation.

Example iSCSI Software Initiator configuration

The iSCSI Software Initiator allows you to configure an iSCSI target device to be used as a boot source. The BIOS current configuration object contains a link to a separate resource of type `HpiSCSISoftwareInitiator`. The BIOS current configuration resource and the iSCSI Software Initiator current configuration resources are read-only. To change iSCSI settings, you need to follow another link to the `Settings` resource, which allows `PUT` and `PATCH` operations.

The iSCSI target configurations are represented in an `iSCSIBootSources` property, that is an array of objects, each containing the settings for a single target. The size of the array represents the total number of iSCSI boot sources that can be configured at the same time. Many mutable properties exist, including `iSCSIBootAttemptInstance`, which can be set to a unique integer in the range [1, *N*], where *N* is the boot sources array size. By default, this *instance number* is 0 for all objects, indicating that the object should be ignored when configuring iSCSI.

Each object also contains two read-only properties—`StructuredBootString` and `UEFIDevicePath`, which are only populated after the target has been successfully configured as a boot source. More information about each property is available in the corresponding schema.

The iSCSI initiator name is represented by the `iSCSIInitiatorName` property.

An additional read-only property, `iSCSINicSources`, is only shown in the iSCSI current configuration resource. This property is an array of strings representing the possible NIC instances that can be used as targets for iSCSI boot configuration. To confirm which NIC device each string corresponds to, it is recommended to cross-reference two other resources:

- A resource of type `HpBiosMapping` can be found through a `Mappings` link in the BIOS current configurations resource. Within its `BiosPciSettingsMappings` property is an array of mappings between BIOS-specific device strings (such as the `NIC` source string) and a `CorrelatableID` string that can be used to refer to the same device in non-BIOS contexts.
- A collection of `HpServerPciDevices` may be found through a PCIDevices link in the `ComputerSystem` resource. The specific PCI device corresponding to the NIC instance can be found by searching for the `CorrelatableID` that will usually match a `UEFIDevicePath`. Once the `HpServerPciDevice` resource is found, you have access to all the human-readable properties useful for describing a NIC source.

Changing the `iSCSIBootSources` and `iSCSIInitiatorName` settings can be done through `PATCH` operations, very similar to how `HpBios` settings are changed. However, whereas all BIOS settings are located in a single flat object, iSCSI settings are nested into arrays and sub-objects. When doing a `PATCH` operation, use empty objects (`{}`) in place of those boot source objects that you **do not** want to alter.

The following example covers a situation where you have configured two iSCSI boot sources, and you would like to edit some existing settings, and add a third source.

1. Iterate through `/rest/v1/Systems` and choose a member `ComputerSystem`. Find a child resource of type `HpiSCSISoftwareInitiator` that allows `PATCH` operations.

   `{ilo-address}/rest/v1/Systems/1/BIOS/iSCSI/Settings`

2. Inspect the existing `iSCSIBootSources` array. You need to inspect the `iSCSIBootAttemptInstance` property of each object to find the boot sources you are prefer to change.

   Existing example resource:

```
{
           "iSCSIBootSources": [
               {
                   "iSCSIBootAttemptInstance": 1,
                   ...
               },
               {
                   "iSCSIBootAttemptInstance": 2,
                   ...
               },
               {
                   "iSCSIBootAttemptInstance": 0,
                   ...
               },
               {
                   "iSCSIBootAttemptInstance": 0,
                   ...
               }
           ],
           ...
       }
```

3. Create a new JSON object with the `iSCSIBootSources` property.

```
{
           "iSCSIBootSources": [
               {},
               {
                   "iSCSIConnectRetry": 2
               },
               {
                   "iSCSIBootAttemptInstance": 3,
                   "iSCSIBootAttemptName": "Name",
                   "iSCSINicSource": "NicBootX",
                   ...
               },
               {}
           ]
       }
```

   Use an empty object in the position of instance `1` to indicate that it should not be modified. Use an object in the position of instance `2` containing the properties that should be modified—all omitted properties will remain unmodified.

   To add a new boot source, find any position of instance `0` and replace it with an object containing all the new settings, and most importantly, a new unique value of `iSCSIBootAttemptInstance`.

4. Change the iSCSI software initiator settings.

   `PATCH {ilo-address}/rest/v1/Systems/1/BIOS/iSCSI/Settings`

Boot settings

UEFI boot structured name string

This UEFI boot structured name string is unique and represents each UEFI boot option in the system. Software can identify and manipulate devices using the string's fixed format as defined in this specification. Software can assume that the string unique for each boot device in the UEFI BootOrder.

The UEFI boot structured name string is divided into sections separated by '.' characters, using the following format:

<DeviceType>.<Location>.<Instance>.<Sub-instance>.<Qualifier>

- **DeviceType**: The first section describes the device type (For example, `HD`, `CD`, `NIC`, and `PCI`.).
- **Location**: The second and the third section together describes the location of the device (For example, `Slot.7` or `Emb.4`).
- **Instance**: The third section is used with the *`Location`* section to describe the device location (for example, the slot number or embedded instance number).
- **Sub-instance**: The fourth section is optional, and is used as a sub-instance number in case of multiple boot options using the same instance. For example, this can be the port number for a multi-port NIC.
- **Qualifier**: The fifth section is optional, and describes the logical protocol (for example, IPv4, IPv6, and iSCSI).

The Structured Boot String information is part of the `BootSources[]property` in the `HpServerBootSettings` object and the `StructuredName` property in the `HpServerPciDevice` object.

UEFI boot structured name string examples

Table 1 Examples

HD.Emb.4.2	The second instance of a hard drive in embedded SA controller bay 4
NIC.Slot.7.2.IPv4	Port 2 of a NIC in PCIe slot 7, which is enabled for PXE IPv4
NIC.FlexLOM.1.1.IPv6	Port 1 of an embedded NIC FlexLOM, which is enabled for PXE IPv6
PCI.Slot.6.1	PCIe card in slot 6
HD.FrontUSB.2.2	Second partition of a flash drive in front USB port 2

Table 2 Examples of currently supported Structured Boot Strings

Device Type	Location	Instance	Sub instance	Qualifier	Structure Boot String Examples
Smart Array Hard Drive	Embedded	Bay number	Incremental by LUN		HD.Emb.1.1
	Slot	Slot number	Incremental by LUN		HD.Slot.1.1
Smart Array Controller	Embedded	Controller Instance	1		RAID.Emb.1.1
	Slot	Slot number	1		RAID.Slot.1.1

Table 2 Examples of currently supported Structured Boot Strings *(continued)*

Device Type	Location	Instance	Sub instance	Qualifier	Structure Boot String Examples
Dynamic Smart Array Controller (Software RAID)	Embedded	1	1		Storage.Emb.1.1
Dynamic Smart Array Controller (Software RAID)	Slot	Controller Instance	1		Storage.Slot.1.1
SATA Hard Drive	Embedded	SATA port # 1			HD.Emb.1.1
SATA Controller	Embedded	Controller Instance	1		SATA.Emb.1.1
All other storage controllers (FC, SAS, etc…)	Embedded	1	1		Storage.Emb.1.1
	Slot	Slot #	1		Storage.Slot.1.1
Network Adapter	LOM	NIC number 1 for 1st NIC 2 for 2nd NIC	Port number	IPv4 or IPv6 or iSCSI or FCoE	NIC.LOM.1.2.IPv4 NIC.LOM.1.2.IPv6
	FlexibleLOM	FlexibleLOM number 1 for 1st FlexLOM 2 for 2nd FlexLOM	Port number	IPv4 or IPv6 or iSCSI or FCoE	NIC.FlexLOM.2.1.IPv4 NIC.FlexLOM.2.1.IPv6
	Slot	Slot number	Port number	IPv4 or IPv6 or iSCSI or FCoE	NIC.Slot.3.2.Ipv4
Fiber Channel Adapter	Slot	Slot number	Port number	IPv4 or IPv6 or iSCSI or FCoE	PCI.Slot.3.1
OS Boot entry (such as “Windows Boot Manager”)	Slot Embedded		Incremental		HD.Emb.1.2 HD.Slot.1.2
USB Key	Front USB	USB Port #	Incremental by LUN		HD.FrontUSB.1.1
	Rear USB	USB Port #	Incremental by LUN		HD.RearUSB.1.1
	Internal USB	USB Port #			HD.InternalUSB.1.1
	iLO virtual media				HD.Virtual.1.1
ISO image	iLO virtual media				CD.Virtual.2.1
Virtual Install Disk (VID)	Embedded store	USB Port #			HD.VirtualUSB.1.1
Embedded User Partition	Embedded store	USB Port #			HD.VirtualUSB.2.1
USB CD/DVD	Front USB	USB Port #			CD.FrontUSB.1.1
	Rear USB	USB Port #			CD.RearUSB.1.1

Table 2 Examples of currently supported Structured Boot Strings *(continued)*

Device Type	Location	Instance	Sub instance	Qualifier	Structure Boot String Examples
	Internal USB	USB Port #			CD.InternalUSB.1.1
SD card	SD slot	USB Port #			HD.SD.1.1
Floppy	Front USB Rear USB	USB Port #			FD.FrontUSB.1.1 FD.RearUSB.1.1
Embedded UEFI Shell	Embedded	1	1		Shell.Emb.1.1
UEFI applications (embedded in the ROM firmware) (Diag, System Utility, etc..)	Embedded	1	Incremental		App.Emb.1.1 App.Emb.1.2 App.Emb.1.3
File	URL	Different URL Increased by 1	1		File.URL.1.1
HPE RAM Disk Device	RAM Memory	1	Port number		RAMDisk.Emb.1.1
Special USB device class with Device Path: UsbClass(0xFFFF, 0xFFFF, 0xFF, 0xFF, 0xFF)	Any USB device in the system	1			Generic.USB.1.1
Empty slot, no device	Slot	Slot number	1		PCI.Slot.2.1
Unknown device	Embedded Slot Unknown location	Slot number or 1	Incremental		Unknown.Slot.1.1 Unknown.Unknown.1.1
NVMe	Slot	Slot number	NVMe drive number (The number is based on bus enumeration sequence).		NVMe.Slot.1.1
NVMe	Embedded	Bay number	1 (Each drive bay has 1 NVMe drive.)		NVMe.Emb.1.1

Change UEFI boot order example

The BIOS current configuration object contains a link to a separate read-only resource of type `HpServerBootSettings` that lists the UEFI Boot Order current configuration. This is the system boot order when the system is configured in the UEFI Boot Mode. The UEFI Boot Order current configuration resource contains a `BootSources` property, which is an array of UEFI boot sources. Each object in that array has a unique `StructuredBootString`, among other properties that identify that boot source.

The UEFI boot order list itself is represented in a separate `PersistentBootConfigOrder` property that is an ordered array of boot sources, each referenced by its `StructuredBootString`. In addition, a `DesiredBootDevices` property lists a separate ordered list of desired boot sources that might not be listed in the `BootSources` property. This is useful for configuring boot from a specific SCSI or FC LUN or iSCSI target that might have not been configured (and discovered by BIOS) yet.

As with the BIOS current configuration resource, the UEFI Boot Order current configuration resource is read only (as evident by the allow header, which do not list `PATCH` as an allowed operation). To change the UEFI Boot Order, you need to follow the link to a separate Settings resource that you can perform a `PATCH` operation on that contains the pending UEFI Boot Order settings, and update that `PersistentBootConfigOrder` and/or the `DesiredBootDevices` properties in that Settings resource. The settings remain pending until next reboot, and the results are reflected back in the `SettingsResults` property in the UEFI Boot Order current configuration resource.

Prerequisites

- Minimum required session ID privileges: Configure

Changing UEFI boot order example

1. Iterate through `/rest/v1/Systems` and choose a member `ComputerSystem`. Find a child resource of type `HpServerBootSettings` that allows `PATCH` operations (there might be more than one but for this exercise, just choose the first one).

   `{ilo-ip-address}/rest/v1/Systems/1/BIOS/Boot/Settings`
2. Obtain the UEFI boot order.

   `GET {ilo-ip-address}/rest/v1/Systems/1/BIOS/Boot/Settings`
3. Create a new JSON object with the `PersistentBootConfigOrder` property and change the boot order.
4. Change the UEFI boot order. You only need to send the updated `PersistentBootConfigOrder` property in the request body.

   `PATCH {ilo-ip-address}/rest/v1/Systems/1/BIOS/Boot/Settings`

When the sever is reset, the new boot order is validated and used.

BIOS administrator password considerations

If you have enabled a BIOS administrator password, you must supply extra information upon a BIOS Settings `PATCH` or `PUT` operation. The `PATCH` or `PUT` operation must include an extra HTTP header:

HTTP Header Name	Value
X-HPRESTFULAPI-AuthToken	A string consisting of the uppercase SHA256 hex digest of the administrator password. In Python this is `hashlib.sha256(bios_password.encode()).hexdigest().upper()`.

Example reset all BIOS and boot order settings to factory defaults

Reseting all BIOS and boot order settings to factory defaults

1. Iterate through `/rest/v1/Systems` and choose a member `ComputerSystem`. Find a child resource of type `HpBios` that allows `PATCH` operations (there might be more than one but for this exercise, just choose the first one).

   `{ilo-ip-address}/rest/v1/Systems/1/BIOS/Settings`
2. Obtain the BIOS and boot order settings.

   `GET {ilo-ip-address}/rest/v1/Systems/1/BIOS`

Reseting all BIOS and boot order settings to factory defaults

3. Create a new JSON object with the `RestoreManufacturingDefaults` property and change the value to yes.
4. Reset the BIOS and boot order settings. You only need to send the updated `RestoreManufacturingDefaults` property in the request body.

   `PATCH {ilo-ip-address}/rest/v1/Systems/1/BIOS`

Power

Server power control is a system-node-level entity, not a chassis-level control. For example, you can turn on one node in a multi-node chassis. You control power by performing an HTTP operation on a computer system node object. For more information about computer system node objects, see “Computer system” (page 18).

Some operations in the interface are not truly RESTful `GET`, `PUT`, `POST`, `DELETE`, or `PATCH`. They are called custom actions and are performed with an HTTP `POST` containing a specific request payload. Typically, actions are defined when the action you want to perform is not adequately represented by the properties available in the type. For example, a power button is not readable, so you cannot `GET` the status of the power button. In this case, pressing the power button is an action.

Actions are `POST` operations with an `Action` property that names the action to perform and zero or more parameter properties.

Reset a server example

Python: See `ex2_reset_server()` in the Python example code.

Prerequisites

- Minimum required session ID privileges: Configure

Resetting a server example

1. Iterate through `/rest/v1/Systems` and choose a member `ComputerSystem` that allows `POST` operations.

   `{ilo-ip-address}/rest/v1/Systems/1`
2. Construct an Action object to submit to iLO.

   `{"Action":"Reset","ResetType":"ForceRestart"}`
3. Change the `Action` and `ResetType` properties to `{"Action":"Reset","ResetType":"ForceRestart"}`.
4. Reset the server.

   `POST {ilo-ip-address}/rest/v1/Systems/1`

The server resets and reboots.

6 Error messages and registries in the iLO RESTful API

Error messages appear in several places in the iLO RESTful API.

- An immediate response to an HTTP operation.
- A `SettingsResult` in the data model where other providers, such as BIOS, processed the settings at some point and want to communicate status in the model.

All error cases use a basic error JSON structure called `ExtendedError` (`ExtendedInfo` in Redfish), which has a `Type` of `ExtendedError.0.9.5`. The most important property in `ExtendedError` is `MessageID`, a string containing a lookup key into a message registry.

`MessageID` helps to keep the iLO service small by keeping much of the explanatory text for an error out of the code. Instead, iLO supplies an `ExtendedError` response, where the `MessageID` provides enough information so that you can look up more details from another file.

For example, if you `POST` to the iLO license service to install an iLO license, but you supply an incorrect key string, iLO responds with an error similar to the following:

```
HTTP response 400
{
  "Type": "ExtendedError.0.9.5",
  "MessageID": "iLO.1.0.InvalidLicenseKey"
}
```

HTTP response 400 is the standard RESTful API response to an error. In the example above, the error is easy to understand, but some errors are not easy to understand. To display a more meaningful error message, parse the string `iLO.0.9.InvalidLicenseKey` into the following components:

- `iLO.0.9`—This is the base name of the message registry to consult. Look for a matching registry file.
- `InvalidLicenseKey`—This is the lookup key into the message registry.

The search returns a result similar to the following:

```
"InvalidLicenseKey":{
  "Description": "The supplied license key is not valid.",
  "Message": "The supplied license key is not valid.",
  "Severity": "Warning",
  "NumberofArgs": 0,
  "ParamTypes": [],
  "Resolution": "Provide a valid license key."
}
```

Many error messages can also return parameters. These parameters might be plugged into the strings in the registry to form detailed messages tailored to the instance of the error message.

Redfish: Redfish has an alternate error response. Consult the Redfish 1.0 specification and consult the `ExtendedInfo` schema.

7 Best practices for creating clients and avoiding incorrect assumptions

When developing a client for the RESTful API, be sure to not code based upon assumptions that are not guaranteed. The reason avoiding these assumptions is so important is that implementations may vary across systems and firmware versions, and we want your code to work consistently.

API Architecture

The RESTful API is a hypermedia API by design. This is to avoid building in restrictive assumptions to the data model that will make it difficult to adapt to future hardware implementations. A hypermedia API avoids these assumptions by making the data model discoverable via links between resources.

The client should not interact with a URI as if it will remain static. Only specific top-level URIs (any URI in this sample code) can be assumed as static.

All URIs, with the exception of known top-level URIs, must be discovered dynamically by following the href links in the data model. Clients should not make assumptions about the URIs for the resource members of a collection. For instance, the URI of a collection member will NOT always be /rest/v1/.../collection/1, or 2. On Moonshot a System collection member might be /rest/v1/Systems/C1N1.

Traversing the data model to find URIs

Although the resources in the data model are linked together, because of cross link references between resources, a client may not assume the resource model is a tree. It is a graph instead, so any crawl of the data model should keep track of visited resources to avoid an infinite traversal loop.

A reference to another resource is any property called `href` (`@odata.id` in Redfish) no matter whereit occurs in a resource.

An external reference to a resource outside the data model is referred to by a property called "extref". Any resource referred to by extref should not be assumed to follow the conventions of the API.

Type Names and Versions

Each resource has a `Type` property with a value of the format `Typename.x.y.z` where:

- x = major version - incrementing this is a breaking change to the schema.
- y = minor version - incrementing this is a non-breaking additive change to the schema.
- z = errata - non-breaking change. . For example alternate description text, spelling fixes, and so forth.

On HTTP POST to Create new Resources

When POSTing to create a resource (e.g. create an account or session), a successful response includes a `Location` HTTP header indicating the resource URI of the newly created resource. The POST may also include a representation of the newly created object in a JSON response body but may not. Do not assume the response body, but test it. It may also be an `ExtendedError` object.

HTTP Redirect

All clients must correctly handle HTTP redirect (e.g. 308, 301, and so on.) iLO 4 will use redirection as a way to alias portions of the data model and to migrate the data model to the Redfish specified URIs (for example, `/redfish/...`).

FUTURE: Asynchronous tasks

In the future some operations may start asynchronous tasks. In this case, the client should recognized and handle HTTP 202 if needed and the `Location` header will point to a resource with task information and status.

Schema vs. Implementation

The json-schema available at `/rest/v1/Schemas` governs the content of the resources, but keep in mind:

- not every property in the schema is implemented in every implementation.
- some properties are schemed to allow both null and another type like string or integer.

Robust client code should check both the existence and type of interesting properties and fail gracefully if expectations are not met.

Additional client information

Clients should always be prepared for:

- Unimplemented properties (for example, a property doesn't apply in a particular case)
- Null values in some cases if the value of a property is not currently known due to system conditions
- HTTP status codes other than 200 OK. Can your code handle an HTTP 500 Internal Server Error with no other info?
- URIs are case insensitive
- HTTP header names are case insensitive
- JSON Properties and Enum values are case sensitive
- A client should be tolerant of any set of HTTP headers the service returns

8 RESTful Events and the Event Service

Starting with iLO 4 2.30, iLO now features a new event subscription service that enables you to subscribe to receive notifications when the REST data changes or when certain alerts occur. These notifications are in the form of HTTPS POST operations to a URI of your choice.

The event service is located in the data model at /rest/v1/EventService. This resource includes a link to a collection of subscriptions (called `EventSubscriptions` located at `/rest/v1/EventService/EventSubscriptions`).

Subscribing for Events examples

In order to receive events, you must provide an HTTPS server accessible to iLO's network with a URI you designate as the target for iLO-initiated HTTPS POST operations.

Construct a JSON object conforming to the type `ListenerDestination` (see example) and POST this to the collection indicated by the `EventSubscriptions` link at `/rest/v1/EventService/EventSubscriptions`. If you receive an HTTP 201 Created response, a new subscription has been added. Note that iLO does not test the destination URI during this phase, so if the indicated URI is not valid, this will not be flagged until events are emitted and the connection to the destination fails.

Example 1 Example POST payload to EventSubscriptions collection (`ListenerDestination`)

```
{
  "Destination": "https://myeventreciever/eventreceiver",
  "EventTypes": [
    "ResourceAdded",
    "ResourceRemoved",
    "ResourceUpdated",
    "StatusChange",
    "Alert"
  ],
  "HttpHeaders": {
    "Header”:”HeaderValue”
  },
  "TTLCount": 1440,
  "TTLUnits": "minutes",
  "Context": "context string",
  "Oem": {
    "Hp": {
      "DeliveryRetryIntervalInSeconds": 30,
      "RequestedMaxEventsToQueue": 20,
      "DeliveryRetryAttempts": 5,
      "RetireOldEventInMinutes": 10
    }
  }
}
```

Much of the above content depends entirely upon your needs and setup:

- **`Destination`** must be an HTTPS URI accessible to iLO's network.
- **`EventTypes`** in the example is everything but you could remove types from the array.
- **`HttpHeaders`** gives you an opportunity to specify any arbitrary HTTP headers you need for the event POST operation. Note that the subscription is readable via GET to an authorized iLO user.
- **`Context`** may be any string.

Consult the **`ListenerDestination`** schema for more details on each property. The subscription will automatically expire after the TTL information specified and must be renewed.

9 Troubleshooting

Resetting the RESTful API using a 3rd party REST web client

Symptom

ProLiant Gen9 servers could possibly experience a RESTful API error during system boot that results in inability to configure the BIOS settings using the RESTful API. In addition, the following persistent error message might display during system boot (POST) and is logged to the Integrated Management Log:

```
335 RESTful API Error- RESTful API PUT request failed (HTTP: Status
Code = 404)
```

With iLO firmware v2.20 or later, you can reset the REST API. You do this through the RESTful API using any 3rd party REST web client, the RESTful Interface Tool, or from the HPE Embedded UEFI shell `restclient` command.

Action

1. Execute a POST operation to the resource at URI `<ilo-ip>/rest/v1/managers/1` with the following JSON in the request body.

```
--------------- Copy Start ----------------
{
    "Action":"ClearRestApiState",
    "Target":"/Oem/Hp"
}
---------------- Copy End -----------------
```

2. Restart the server.

Resetting the iLO RESTful API using the RESTful Interface Tool

Symptom

ProLiant Gen9 servers could possibly experience a RESTful API error during system boot that results in inability to configure the BIOS settings using the RESTful API. In addition, the following persistent error message might display during system boot (POST) and is logged to the Integrated Management Log:

```
335 RESTful API Error- RESTful API PUT request failed (HTTP: Status
Code = 404)
```

With iLO firmware v2.20 or later, you can reset the REST API. You do this through the RESTful API using any 3rd party REST web client, the RESTful Interface Tool, or from the HPE Embedded UEFI shell `restclient` command.

Cause

Action

1. Download and install the RESTful interface Tool. For more information on using this tool, refer to **http://www.hpe.com/info/resttool**.
2. Copy and paste the following JSON into a text file and save it as `hprest_tool_clear_api.json`.

```
--------------- Copy Start ----------------
{
    "path": "/rest/v1/managers/1",
    "body": {
        "Action":"ClearRestApiState",
        "Target":"/Oem/Hp"
```

```
    }
}
--------------- Copy End -----------------
```

3. Start the hprest tool.

   ```
   hprest
   ```

4. Log in to iLO.

   ```
   hprest> login <ilo-ip>
   ```

5. Run the following command, pointing to the `hprest_tool_clear_api.json` file.

   ```
   hprest> rawpost hprest_tool_clear_api.json
   ```

6. Restart the server.

Resetting the iLO RESTful API using the Embedded UEFI Shell `restclient` command

Symptom

ProLiant Gen9 servers could possibly experience a RESTful API error during system boot that results in inability to configure the BIOS settings using the RESTful API. In addition, the following persistent error message might display during system boot (POST) and is logged to the Integrated Management Log:

```
335 RESTful API Error- RESTful API PUT request failed (HTTP: Status
Code = 404)
```

With iLO firmware v2.20 or later, you can reset the REST API. You do this through the RESTful API using any 3rd party REST web client, the RESTful Interface Tool, or from the HPE Embedded UEFI shell `restclient` command.

Cause

Action

1. Enter the Embedded UEFI Shell. For more information, refer to the *UEFI Shell User Guide* at **http://www.hpe.com/servers/proliant/uefi**.
2. Copy and paste the following JSON into an ASCII text file save it as `clear_api.json` on FAT formatted USB media.

   ```
   --------------- Copy Start ----------------
   {
     "Action":"ClearRestApiState",
     "Target":"/Oem/Hp"
   }
   --------------- Copy End -----------------
   ```

3. Attach the USB media to the server.
4. Turn on the server and boot to the Embedded UEFI Shell.
5. At the UEFI shell prompt, use the `partitions` command to find the file system that corresponds to the USB media. For example, `FS0`, `FS1`, and so on.
6. To switch to the file system, type the file system name (for example, `shell>FS0:`).
7. Execute the following command:

   ```
   Fs0:> restclient –m POST –uri "/rest/v1/managers/1" –i clear_api.json
   ```

8. Restart the server.

Resetting the iLO RESTful API using the iLO SSH CLI

Symptom

ProLiant Gen9 servers could possibly experience a RESTful API error during system boot that results in inability to configure the BIOS settings using the RESTful API. In addition, the following persistent error message might display during system boot (POST) and is logged to the Integrated Management Log:

```
335 RESTful API Error- RESTful API PUT request failed (HTTP: Status
Code = 404)
```

With iLO firmware v2.20 or later, you can reset the REST API. You do this through the RESTful API using any 3rd party REST web client, the RESTful Interface Tool, or from the HPE Embedded UEFI shell `restclient` command.

Cause

Action

1. Open an SSH connection with iLO, log in using an account with administrator privileges. For more information, see the *HPE iLO 4 Scripting and Command Line Guide* at **http://www.hpe.com/info/iLO**.
2. At the CLI prompt, execute the command `oemhp_clearRESTAPIstate`. Note that this command might take a few seconds to complete.
3. Restart the server.

Unusual characters in JSON when Subscribing to an Alert EventType

Symptom

When subscribing to an Alert EventType, unusual or extra characters are included at the end of the JSON file.

Cause

The event listener is parsing beyond the `Content-Length` value in the header.

Action

10 iLO RESTful API functionality for Gen9 servers

The following table lists the additional REST API functionality available for Gen9 servers running iLO 4 2.30 as compared to Gen8 servers running iLO 4 2.30.

# UEFI enhanced memory detail	
/rest/v1/Systems/{item}/Memory/{item}	HpMemory.1.0.0 #/Manufacturer
# UEFI enhanced PCI Device detail	
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/ClassCode
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/DeviceID
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/DeviceInstance
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/DeviceSubInstance
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/SegmentNumber
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/StructuredName
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/SubclassCode
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/SubsystemDeviceID
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/SubsystemVendorID
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/UEFIDevicePath
/rest/v1/Systems/{item}/PCIDevices/{item}	HpServerPciDevice.1.0.0 #/VendorID
# UEFI PCI Slot Detail	
/rest/v1/Systems/{item}/PCISlots/{item}	HpServerPCISlot.1.0.0 #/UEFIDevicePath
# BIOS-related resources	
/rest/v1/Systems/{item}/SecureBoot	HpSecureBoot.1.0.0
/rest/v1/Systems/{item}/bios	HpBios.1.2.0
/rest/v1/Systems/{item}/bios/Settings	HpBios.1.2.0
/rest/v1/Systems/{item}/bios/iScsi/Settings	HpiSCSISoftwareInitiator.1.1.0
/rest/v1/Systems/{item}/bios/Mappings	HpBiosMapping.1.2.0
/rest/v1/Systems/{item}/bios/iScsi/BaseConfigs	HpBaseConfigs.0.10.0
/rest/v1/Systems/{item}/bios/iScsi	HpiSCSISoftwareInitiator.1.1.0
/rest/v1/Systems/{item}/bios/Boot/Settings	HpServerBootSettings.1.2.0
/rest/v1/Systems/{item}/bios/BaseConfigs	HpBaseConfigs.0.10.0
# PCI Device firmware versions	
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/{PCIDeviceID}[]/ImageSizeBytes
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/{PCIDeviceID}[]/Key
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/{PCIDeviceID}[]/Location
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/{PCIDeviceID}[]/Updateable
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/{PCIDeviceID}[]/VersionString
# Gen9 new firmware components	

/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/SASProgrammableLogicDevice[]/Location
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/SASProgrammableLogicDevice[]/VersionString
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/StorageBattery[]/Location
/rest/v1/Systems/{item}/FirmwareInventory	FwSwVersionInventory.1.0.0 #/Current/StorageBattery[]/VersionString
/rest/v1/Chassis/{item}	HpServerChassis.1.0.0 #/Oem/Hp/Firmware/SASProgrammableLogicDevice/Current/VersionString

11 Support and other resources

Accessing Hewlett Packard Enterprise Support

- For live assistance, go to the Contact Hewlett Packard Enterprise Worldwide website:

  www.hpe.com/assistance

- To access documentation and support services, go to the Hewlett Packard Enterprise Support Center website:

  www.hpe.com/support/hpesc

Information to collect

- Technical support registration number (if applicable)
- Product name, model or version, and serial number
- Operating system name and version
- Firmware version
- Error messages
- Product-specific reports and logs
- Add-on products or components
- Third-party products or components

Accessing updates

- Some software products provide a mechanism for accessing software updates through the product interface. Review your product documentation to identify the recommended software update method.
- To download product updates, go to either of the following:
  - Hewlett Packard Enterprise Support Center **Get connected with updates** page:

    www.hpe.com/support/e-updates

  - Software Depot website:

    www.hpe.com/support/softwaredepot

- To view and update your entitlements, and to link your contracts and warranties with your profile, go to the Hewlett Packard Enterprise Support Center **More Information on Access to Support Materials** page:

  www.hpe.com/support/AccessToSupportMaterials

IMPORTANT: Access to some updates might require product entitlement when accessed through the Hewlett Packard Enterprise Support Center. You must have an HP Passport set up with relevant entitlements.

Websites and documents

Table 3 Websites

Website	Link
Hewlett Packard Enterprise Information Library	**http://www.hpe.com/info/enterprise/docs**
UEFI	**http://www.hpe.com/info/ProLiantUEFI/docs**
HPE Service Pack for ProLiant	**http://www.hpe.com/servers/spp/documentation**
HPE iLO 4	**http://www.hpe.com/info/ilo/docs**
HPE iLO University videos	**http://www.hpe.com/info/ilo/videos**
HPE Systems Insight Manager	**http://www.hpe.com/info/hpsim**
HPE Onboard Administrator	**http://www.hpe.com/info/oa**
HPE VMware Vibs Depot	**http://vibsdepot.hpe.com/**
Hewlett Packard Enterprise Information Library	**www.hpe.com/info/enterprise/docs**
Hewlett Packard Enterprise Support Center	**www.hpe.com/support/hpesc**
Contact Hewlett Packard Enterprise Worldwide	**www.hpe.com/assistance**
Subscription Service/Support Alerts	**www.hpe.com/support/e-updates**
Software Depot	**www.hpe.com/support/softwaredepot**
Customer Self Repair	**www.hpe.com/support/selfrepair**
Insight Remote Support	**www.hpe.com/info/insightremotesupport/docs**
Serviceguard Solutions for HP-UX	**www.hpe.com/info/hpux-serviceguard-docs**
Single Point of Connectivity Knowledge (SPOCK) Storage compatibility matrix	**www.hpe.com/storage/spock**
Storage white papers and analyst reports	**www.hpe.com/storage/whitepapers**

Table 4 Documents

Document
iLO RESTful API Data Model Reference
HPE iLO 4 Scripting and Command Line Guide
HPE iLO 4 Release Notes
UEFI System Utilities User Guide
HPE ProLiant Gen9 Troubleshooting Guide, Volume I: Troubleshooting
HPE iLO Federation User Guide
HPE Service Pack for ProLiant Quick Start Guide

Customer self repair

Hewlett Packard Enterprise customer self repair (CSR) programs allow you to repair your product. If a CSR part needs to be replaced, it will be shipped directly to you so that you can install it at your convenience. Some parts do not qualify for CSR. Your Hewlett Packard Enterprise authorized service provider will determine whether a repair can be accomplished by CSR.

For more information about CSR, contact your local service provider or go to the CSR website:

www.hpe.com/support/selfrepair

Remote support

Remote support is available with supported devices as part of your warranty or contractual support agreement. It provides intelligent event diagnosis, and automatic, secure submission of hardware event notifications to Hewlett Packard Enterprise, which will initiate a fast and accurate resolution based on your product's service level. Hewlett Packard Enterprise strongly recommends that you register your device for remote support.

For more information and device support details, go to the following website:

www.hpe.com/info/insightremotesupport/docs

Documentation feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (**docsfeedback@hpe.com**). When submitting your feedback, include the document title, part number, edition, and publication date located on the front cover of the document. For online help content, include the product name, product version, help edition, and publication date located on the legal notices page.

A Warranty and regulatory information

For important safety, environmental, and regulatory information, see *Safety and Compliance Information for Server, Storage, Power, Networking, and Rack Products*, available at **www.hpe.com/support/Safety-Compliance-EnterpriseProducts**.

Warranty information

HPE ProLiant and x86 Servers and Options

www.hpe.com/support/ProLiantServers-Warranties

HPE Enterprise Servers

www.hpe.com/support/EnterpriseServers-Warranties

HPE Storage Products

www.hpe.com/support/Storage-Warranties

HPE Networking Products

www.hpe.com/support/Networking-Warranties

Regulatory information

Belarus Kazakhstan Russia marking

EAC

Manufacturer and Local Representative Information

Manufacturer information:

- Hewlett Packard Enterprise Company, 3000 Hanover Street, Palo Alto, CA 94304 U.S.

Local representative information Russian:

- **Russia**:

  ООО «Хьюлетт Паккард Энтерпрайз», Российская Федерация, 125171, г. Москва, Ленинградское шоссе, 16А, стр.3, Телефон/факс: +7 495 797 35 00

- **Belarus**:

  ИООО «Хьюлетт-Паккард Бел», Республика Беларусь, 220030, г. Минск, ул. Интернациональная, 36-1, Телефон/факс: +375 17 392 28 18

- **Kazakhstan**:

  ТОО «Хьюлетт-Паккард (К)», Республика Казахстан, 050040, г. Алматы, Бостандыкский район, проспект Аль-Фараби, 77/7, Телефон/факс: + 7 727 355 35 50

Local representative information Kazakh:

- **Russia**:

  ЖШС "Хьюлетт Паккард Энтерпрайз" Ресей Федерациясы, 125171, Мәскеу, Ленинград тас жолы, 16А блок 3, Телефон/факс: +7 495 797 35 00

- **Belarus**:

  «HEWLETT-PACKARD Bel» ЖШС, Беларусь Республикасы, 220030, Минск қ., Интернациональная көшесі, 36/1, Телефон/факс: +375 17 392 28 18

- **Kazakhstan**:

  ЖШС «Хьюлетт-Паккард (К)», Қазақстан Республикасы, 050040, Алматы к., Бостандык ауданы, Әл-Фараби даңғ ылы, 77/7, Телефон/факс: +7 727 355 35 50

Manufacturing date:

The manufacturing date is defined by the serial number.

CCSYWWZZZZ (serial number format for this product)

Valid date formats include:

- YWW, where Y indicates the year counting from within each new decade, with 2000 as the starting point; for example, 238: 2 for 2002 and 38 for the week of September 9. In addition, 2010 is indicated by 0, 2011 by 1, 2012 by 2, 2013 by 3, and so forth.
- YYWW, where YY indicates the year, using a base year of 2000; for example, 0238: 02 for 2002 and 38 for the week of September 9.

Turkey RoHS material content declaration

Türkiye Cumhuriyeti: EEE Yönetmeliğine Uygundur

Ukraine RoHS material content declaration

Обладнання відповідає вимогам Технічного регламенту щодо обмеження використання деяких небезпечних речовин в електричному та електронному обладнанні, затвердженого постановою Кабінету Міністрів України від 3 грудня 2008 № 1057