HowTo – Develop Redfish Interface Emulator

August 8, 2016

This document should be read by programmer wanting to:

- Run the emulator with a mockup
- Create dynamic resources
- Add new resource types

Contents

1 Common Knowledge

1.1 Installing the Python Locally

1. Install Python 2.7 (https://www.python.org/downloads/release/python-279/).
2. Install required Python packages by executing the command in command window.

```
$ python install.py
```

The file install.py installs the packages found in the 'dependencies' folder

1.2 Emulator Configuration File

Emulator.py accepts several command line parameters.

```
usage: emulator.py [-h] [-v] [-port PORT] [-debug]

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit
  -port PORT     Port to run the emulator on. Default is 5000
  -debug         Run the emulator in debug mode. Note that if you run in debug
                 mode, then the emulator will only be run locally.
```

Emulator.py reads the configuration file, emulator-config.json, to determine the port number to monitor for requests.

- The MODE property specifies port to use. If the value is 'Cloud', the port is assigned by the cloud foundry. If the value is ‘Local’, the port is assigned the value of 5000
- The SPEC property specifies whether the node is represented as a Redfish ComputerSystem (SPEC = “Redfish”) or another structure.
- The TRAYS property specifies the resources that constitutes the resource pool of the pod when it first starts up. These resource will determine the size of the processor and memory resource pools. Multiple trays can be specified.

```
{
    "SPEC": "Redfish",
    “MODE”: “Cloud”,
    "TRAYS": [
        "./Resources/Systems/1/index.json"
    ]
}
```

2 Running the Emulator

2.1 Running the Emulator, locally on Windows

To run the emulator entirely local to Windows.

1. Check emulator-config.py file has the "MODE" property set to 'Local'
2. Start the emulator
    1. Start a Command Window
    2. Change directory to the top of the emulator source tree
    3. Execute the following command

```
$ python emulator.py
```

3. Access via browser

    1. Start Chrome-Postman
    2. Enter the following path into the URI field

    ```
    http://localhost:5000/redfish/v1
    ```

    3. The HTTP command should be set to GET
    4. Click the 'Send' button

2.2 Running Emulator in the Intel Cloud Foundry

Change the MODE in Emulator-config.json to “Cloud” and perform a "cf push" onto the Intel Cloud Foundry.

The MODE switch changes how the port is assigned by the Flask package. For a cloud foundry, the application needs to know how to determine the port for the cloud foundry instance to connect to the HTTP interface. Cloud foundries have documentation on what variable is used to convey this information.

2.3 SSL and HTTP Basic Authentication

The default emulator (emulator.py) does not support SSL or HTTP authentication. If you want to requests to 'https', use emulator_ssl.py.

1. Run emulator_ssl.py.
2. Access emulator from website or postman (in this case you have to authorization type to basic auth) using https protocol.
3. Login with credentials ‘Administrator’:’Password’ for admin role or ‘User’:’Password’ for ReadOnlyUser role.

3 Testing the Emulator

The command to test the emulator can executed against the emulator running locally or host in the cloud.

The following is the general command for running unit test. The value of <SPEC> is Redfish and should correspond to the value in emulator-config.json used to start the emulator. The value of <PATH> is the path to use for testing. <PATH> should be enclosed in double-quotes.

```
$ python unittests.py <SPEC> "<PATH>"
```

3.1 Running the Unit Test, locally

An instance of the emulator running locally can be tested, via a loopback connection.

1. Check emulator-config.py file has the "MODE" property set to 'Local'
2. Start a Command Line Window
3. Change current directory to the top of the emulator source tree
4. Execute the following command

```
$ python unittests.py Redfish "localhost:5000"
```

5. Inspect the log file produced, “test-rsa-emulator.log”

The program unittest.py reads the configuration file to determine the port number to use to communicate with the emulator.

3.2 Running the Unit Test on a remotely hosted

An instance of the emulator running in the cloud can be test.

1. Check emulator-config.py file has the "MODE" property set to 'Cloud'
2. Start a Command Line Window
3. Change current directory to the top of the emulator source tree
4. Execute the following command. It the example, the URL to the cloud instance is in quotes.

```
$ python unittests.py Redfish "rsa-emulator.apps1-fm-int-icloud.intel.com"
```

5. Inspect the log file produced, "test-rsa-emulator.log"

FYI, unittests.py reads the configuration file to determine the port number to use to communicate with the emulator.

One can execute, the "cf push" and execute unittest.py sequentially from the same Git Shell. Just use the URL from the "cf push" output.

```
$ cf push rsa-wip
. . .
instances: 1/1
usage: 256M x 1 instances
urls: rsa-wip.apps1-fm-int.icloud.intel.com
last uploaded: Thu Mar 3 19:25:24 UTC 2016

$ python unittests.py Redfish "rsa-wip.apps1-fm-int-icloud.intel.com"
```

4 Code Overview

The emulator uses Flask-RESTful, which allows easier RESTful code than Flask.

4.1 Flask

With Flask, the code associates each resource path and method to a Python method. The problem with this code is our resource is split up over multiple methods. There's no encapsulation. Below is an example of the code generated.

```
app = Flask(__name__)

@app.route('/tasks', methods=['GET'])
def get_all_tasks():
    return task_db.fetch_all_tasks()

@app.route('/tasks', methods=['POST'])
def create_task():
    task_string = request.form['task']
    task_db.create_task(task_string)
    return task_string

@app.route('/tasks/<int:id>', methods=['GET'])
def get_task(id)
    return task_db.fetch_task(id)
```

4.2 Flask-RESTful

With Flask-RESTful, routes map directly to objects and the class's methods are the same as their HTTP counterparts (e.g. get, post, etc.). In other words, the RESTful resource are implemented as classes. Below is an example of the code generated with does the same as above.

```
app = Flask(__name__)
api = restful.Api(app)

class Users(restful.Resource):
    def get(self):
```

```
        return task_db.fetch_all_tasks()

    def post(self):
        task_string = request.form['task']
        task_db.create_task(task_string)
        return task_string

class User(restful.Resource)
    def get(self, id):
        return task_db.fetch_task(id)

api.add_resource(Users, '/tasks')
api.add_resource(User, '/tasks/<int:id>')
```

4.3 Program Flow

The Emulator starts by reading the configuration file (emulator-config.json) and obtain the configuration parameters (SPEC, MODE and TRAYS).

The emulator.py file loads the configuration. The code handles the non-Redfish paths (e.g. / and /redfish/v1/reset. The code for 'RedfishAPI' class is defined as a Flask-RESTful class. Otherwise, control is passes the Resource Manager (resource_manager.py).

The resource manager loads the static resources (mockup) for Chassis, Managers, Tasks, Sessions, Account Services and Registries.

The resource manager then loads the dynamic resources. Dynamic resources are resources that: 1) can have properties that change, 2) actions that get invoked (via POST) or 3) get created and deleted.

Finally, the Emulator waits to receive HTTP requests. As requests are received, additional requests are made to the resource manager support the request.

5 Modifying the Source

5.1 Change the base URI

The base URI is defined in the field REST BASE in emulator.py. This is the single point from where the entire emulator derives its base URI. To change the base URI, edit this value.

```
# Base URL of the RESTful interface
REST_BASE = '/redfish/v1/'
```

5.2 Add Static mockups

Static mockups are added by:

1. Copying the mockup directory tree into the ./static folder
2. Edit resource_manager.py to fix up ServiceRoot for new entries and indicating what directory to load for a new ServiceRoot entry.

In the details below, the examples will use PCIeSwitches.

5.2.1 Copying to static folder

Copy the mockup files into ./api_emulator/static/redfish.

5.2.2 Editing resource_manager.py

The structure of the ServiceRoot is statically defined in resource_manager.py. The ServiceRoot in the code has the resources which Redfish 1.0.0 defines as being in ServiceRoot.

If the mockup copied in has new folders that need to appear in the ServiceRoot, then the resource_manager.py needs to be edited, as follows:

1. In the 'configuration' function, add a line to the 'Links' declaration in the definition of the Service Root. The following is the fragment for PCIeSwitches.

```
        'Links': {
              'Chassis': {'@odata.id': self.rest_base + 'Chassis'},
              'Hierarchy':{'@odata.id': self.rest_base + 'Hierarchy'},
              'Managers':{'@odata.id': self.rest_base + 'Managers'},
              'PCIeSwitches':{'@odata.id': self.rest_base + 'PCIeSwitches'},
              . . .
        }
```

2. In the '__init__' function, add a line to load the mockup folder. This line will recursive load the index.json files from the specified folder. The load_static arguments are <folder name>, <profile name>, rest_base. The path to the folder is constructed as

```
./static/<profile name>/<folder name>
```

The following is the fragment for PCIeSwitches.

```
      # Static Redfish Modules

       self.Tasks = load_static('Tasks', 'redfish', rest_base)
       self.Sessions = load_static('Sessions', 'redfish', rest_base)
       self.AccountService = load_static('AccountService', 'redfish', rest_base)
       self.EventService = load_static('EventService', 'redfish', rest_base)
       self.Registries = load_static('Registries', 'redfish', rest_base)
       self.PCIeSwitches = load_static('PCIeSwitches','redfish',rest_base)
```

5.3 Implement dynamic resources

While static resources are just a simple copy, creating dynamic resources requires a little programming. The following show the steps:

1. Write a template file and an API file in Python
2. Edit the appropriate Python file to add the API to the RESTful interface.

A template file is needed for singletons, but not for collections. Since collections resource are simple and similar, their template are incorporated in the code in the API file. The API file includes HTTP function definitions for the singleton and its associated collection (if the singleton is a member of a collection).

Below following sections have example using the chassis collection resource and chassis resource.

5.3.1 Write a Template file

The dynamic resource constructor loads a template which instantiates the resource and its properties. The template file has two sections, a template declaration and a single function.

The template declaration is derived from the mockup file. The template declaration is defined as a Python dictionary (which is very similar to JSON).

The template declaration also contains substitution fields. In the following example, the substitution fields are {rb} and {id}, which are replaced by the root_base (/redfish/v1) and the instance ID, respectively.

```
_CHASSIS_TEMPLATE = \
    {
        "@odata.context": "{rb}$metadata#Chassis/Links/Members/$entity",
        "@odata.id": "{rb}Chassis/{id}",
        "@odata.type": "#Chassis.1.0.0.Chassis",
```

```
        "Id": "{id},
        "Name": "Computer System Chassis",
        "ChassisType": "RackMount",
        "Manufacturer": "Redfish Computers",
        "Model": "3500RX",
        "SKU": "8675309",
        "SerialNumber": "437XR1138R2",
        "Version": "1.02",
        "PartNumber": "224071-J23",
        "AssetTag": "Chicago-45Z-2381",
        "Status": {
            "State": "Enabled",
            "Health": "OK"
        },
        "Links": {
            "ComputerSystems": [{ "@odata.id": "{rb}Systems/" }],
            "ManagedBy": [{ "@odata.id": "{rb}Managers/1" }],
            "ThermalMetrics": { "@odata.id": "{rb}Chassis/{id}/ThermalMetrics" },
            "PowerMetrics": { "@odata.id": "{rb}Chassis/{id}/PowerMetrics" },
            "MiscMetrics": { "@odata.id": "{rb}Chassis/{id}/MiscMetrics" },
        },
    }
```

The second section of the template file is the function definition. The function is called to instantiate an instance of the resource.

First, the function makes a copy of the template declaration (using deepcopy).

The function replaces the substitution fields. The 'rest_base' variable is set by the system to "redfish/v1". The 'ident' variable is decoded from the id of the member.

The function may set the value of any property. In the example, the default SerialNumber is overwritten by a randomly create one.

```
def get_chassis_template(rest_base, ident):

    c = copy.deepcopy(_CHASSIS_TEMPLATE)

    c['@odata.context'] = c['@odata.context'].format(rb=rest_base)
    c['@odata.id'] = c['@odata.id'].format(rb=rest_base, id=ident)
    c['Id'] = c['Id'].format(id=ident)
    c['Thermal']['@odata.id'] = c['Thermal']['@odata.id'].format(rb=rest_base,
id=ident)
    c['Power']['@odata.id'] = c['Power']['@odata.id'].format(rb=rest_base, id=ident)
    c['Links']['ManagedBy'][0]['@odata.id'] =
c['Links']['ManagedBy'][0]['@odata.id'].format(rb=rest_base, id=ident)
c['Links']['ComputerSystems'][0]['@odata.id']=c['Links']['ComputerSystems'][0]['@odata
.id'].format(rb=rest_base)

    c['SerialNumber'] = strgen.StringGenerator('[A-Z]{3}[0-9]{10}').render()

    return c
```

5.3.2 Write an API file.

The API file specifies the behavior for each HTTP request. The code below is pretty generic and can be copied for other collection resource and member resources.

Since Chassis is member of a collection, the code below is for the ChassisCollectionAPI class. The GET and POST are defined. The post function adds the ChassisAPI to the interface. The code also has specific error return codes for PUT and DELETE. Note, if a HTTP command is not present, the default Flask error will be returned.

```
class ChassisCollectionAPI(Resource):
    def __init__(self):
        self.rb = g.rest_base
```

```
        self.config = {
            '@odata.context': self.rb + '$metadata#ChassisCollection',
            '@odata.id': self.rb + 'ChassisCollection',
            '@odata.type': '#ChassisCollection.1.0.0.ChassisCollection',
            'Name': 'Chassis Collection',
            'Links': {}
        }
        self.config['Links']['Member@odata.count'] = len(member_ids)
        self.config['Links']['Members'] = member_ids

    # HTTP GET
    def get(self):
        try:
            resp = self.config, 200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        return resp

    # HTTP POST
    def post(self):
        try:
            g.api.add_resource(ChassisAPI, '/redfish/v1/Chassis/<string:ident>')
            resp=self.config,200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        return resp

    # HTTP PUT
    def put(self,ch_id):
         return 'PUT is not a valid command', 202

    # HTTP DELETE
    def delete(self,ch_id):
         return 'DELETE is not a valid command', 202
```

The definition of the Chassis API class is shown below. All the Chassis functions interact with the member[] and member_id[] lists. Namely, the functions find the entry that corresponds to the member ID.

The GET, POST, PATCH and DELETE are defined. The patch() function will update any property in the JSON file. The post() function creates a chassis and its subordinate resources.

```
class ChassisAPI(Resource):
    def __init__(self):
        print ('ChassisAPI init called')

    # HTTP GET
    def get(self,ident):
        try:
            # Find the entry with the correct value for Id
            for cfg in members:
                if (ident == cfg["Id"]):
                    break
            config = cfg
            resp = config, 200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        return resp

    # HTTP POST
    def post(self,ident):
        print ('ChassisAPI put called')
        try:
            global config
```

```
            config=get_Chassis_template(g.rest_base,ident)
            members.append(config)
            member_ids.append({'@odata.id': config['@odata.id']})
            global foo
            # Attach URIs for subordiante resources
            if  (foo == 'false'):
                g.api.add_resource(ThermalAPI,
'/redfish/v1/Chassis/<string:ch_id>/Thermal')
                g.api.add_resource(PowerAPI,
'/redfish/v1/Chassis/<string:ch_id>/Power')
                foo = 'true'
            # Create instances of subordinate resources, then call put operation
            cfg = CreateThermal()
            out = cfg.put(ident)
            cfg = CreatePower()
            out = cfg.put(ident)
            resp = config, 200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        print ('ChassisAPI put exit')
        return resp

    # HTTP PATCH
    def patch(self, ident):
        print ('ChassisAPI patch called')
        raw_dict = request.get_json(force=True)
        print (raw_dict)
        try:
            # Find the entry with the correct value for Id
            for cfg in members:
                if (ident == cfg["Id"]):
                    break
            config = cfg
            print (config)
            for key, value in raw_dict.items():
                print ('Update ' + key + ' to ' + value)
                config[key] = value
            print (config)
            resp = config, 200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        return resp

    # HTTP DELETE
    def delete(self,ident):
        # print ('ChassisAPI delete called')
        try:
            idx = 0
            for cfg in members:
                if (ident == cfg["Id"]):
                    break
                idx += 1
            members.pop(idx)
            member_ids.pop(idx)
            resp = 200
        except Exception:
            traceback.print_exc()
            resp = INTERNAL_ERROR
        return resp
```

5.3.3 Add the resource to interface

With the API's defined, other code needs to be modified to add the API to the resource URIs. The code which is modified depends on when one wants the resource API to begin responding. For example:

1. The resource API is present when the RESTful interface starts. The resources that are present off the ServiceRoot are examples of these resources.
2. The resource API is present when a parent resource is created. Subordinate resources are examples of these type of resources. For example, ./chassis/{id}/Thermal should not be present, until ./Chassis/{id} has been created.
3. The resource API is a member resource and should be created when the member is created.

An example of #3 can be found in the collection resource API code, above. The POST function for the collection, adds the API for the member instance which has been created.

An example of #2 can be found in the Chassis resource API code, above. The POST function for the Chassis resource, contains add_resource() calls the subordinate Thermal and Power resource when a Chassis resource member is created.

An example of #1 is shown below. In this example, we replace the static Chassis resources with dynamic Chassis resources.

In the file emulator_resource.py, the load_static() function call is commented out and replaced with two api.add_resource calls. The first call adds the API for the Chassis Collection. The second call adds the API for any Chassis member.

Furthermore, in this example, we like the Chassis Collection to start with a member instance (instead of being empty). To do this, an instance of CreateChassis is created and then its put() function is called with the name of the instance.

```
   # self.Chassis = load_static('Chassis', 'redfish', rest_base,
self.resource_dictionary)

   g.api.add_resource(ChassisCollectionAPI, '/redfish/v1/Chassis/')
   g.api.add_resource(ChassisAPI, '/redfish/v1/Chassis/<string:ident>')

   config = CreateChassis()
   out = config.put('Test2')
```

6 Archive

The following section describes code changes for the Flask construction and may change for Flask-restful.

6.1 Define "Actions" for POST operations

Redfish allows POST to include an action to be perform on the resource.

To implement the code for an action, the code needs to fork a thread to perform the action, while the main thread returns a HTTP response (command accepted).

In the code for the dynamic resource define the actions in the constructor. The "action" on the POST request is the dictionary key. The example shows the "Reset" and "ApplySetting" actions.

6.1.1 Reset Action

```
self.actions = {
            'Reset':self.reset_system,
            'ApplySetting':self.apply_setting}
```

In the example, the reset_system() method performs the reset. It does some checking, then calls the update_method() to change the PowerState.

```
@staticmethod
    def reset_system(action):
```

```
        global resource_manager
        try:
            assert request.json is not None, 'No JSON configuration given'
            if(action =='Reset'):
                path_x='Systems/1'
                path = path_x.split('/')
                path[0]=path_x
                cs_puid = int(path[1])
                config = resource_manager.update_method(cs_puid,request.json)
```

6.1.2 ApplySetting

The update_method() routine, the computer system of interest is obtained (cs), then calls its update() routine.

```
def update_method(self,cs_puid,rs):
        """
            Updates the power metrics of Systems/1
        """
        cs=self.Systems[cs_puid]
        cs.update(rs)
        return cs.configuration
```

The update() routine (defined in computerssytem.py) changes the PowerState property from "On" to "Off".

```
def update(self,config):
        self.config['PowerState']=config['PowerState']
```

7 Emulator Source Tree

7.1 Root directory

This directory contain source for the root of the source tree.

The following subdirectories are described in greater detail below:

- api_emulator
- api_emulator/redfish (Redfish resources)

Files & Directories	Description of Contents
Emulator.py	The main Python file for the emulator. Reads the emulator-config.json file.
Emulator-config.json	Contains configuration properties for the emulator. See description above. the decision points for SPEC,MODE,TRAY and OVFs
Install.py	Installs all the dependencies for python packages which are present in 'Dependencies' folder. This file is used to prepare the local environment to run the emulator.
Requirements.txt	This file is used by the Cloud Foundry. The file lists the python packages which are required to be installed. The Cloud Foundry should install these packages automatically as part of the 'cf push'
ProcFile	This file is used by the Cloud Foundry. The file specifies the command to execute after a successfully 'cf push'

Unittests.py	Tests the emulator locally - few tests have been defined locally. Reads to emulator-config.json file.
Unittest_data	Used by unittest.py for testing purposes.
./Dependencies	Directory which contains Python packages that need to be installed to support the emulator. These packages are referenced by install.py, when executed.

7.2 api_emulator directory

This directory contain source for the emulator infrastructure.

Files & Directories	Description of Contents
Emulator_resource.py	Loads the resources that become the resource pool. Components of tray are defined. See Resource/Systems.
Exception.py	Exceptions thrown in the API Emulator
Resource_manager.py	Loads and constructs the resources that are emulated. Loads both the static and dynamic resources. Loads the resource pool. Creation, deletion and utility of resources
Static_loader.py	Called by resource_manager.py to recursive load a static resource from the ./redfish/static directory. The recursion is performed by following the subordinate odata.id URI's in each resource.
Utils.py	Supporting functions

7.3 api_emulator/redfish directory

This directory contain source for models for Redfish 1.0 specification.

Files & Directories	Description of Contents
./	Contains code for dynamic resources
./static	Contains the mockup code for static resources.
./templates	Contains the template files for the dynamic resources.