PICMG Redfish Server Implementation using OpenAPI

This document is prepared as part of the Master's Capstone Project for ASU by Fall 2022 students.

The following students contributed to the project:

Manan Soni
Sambudha Nath
Govind Venugopal
Mayank Tewatia
Vishnu Preetham Reddy Dasari
Sudhanva Hanumanth Rao

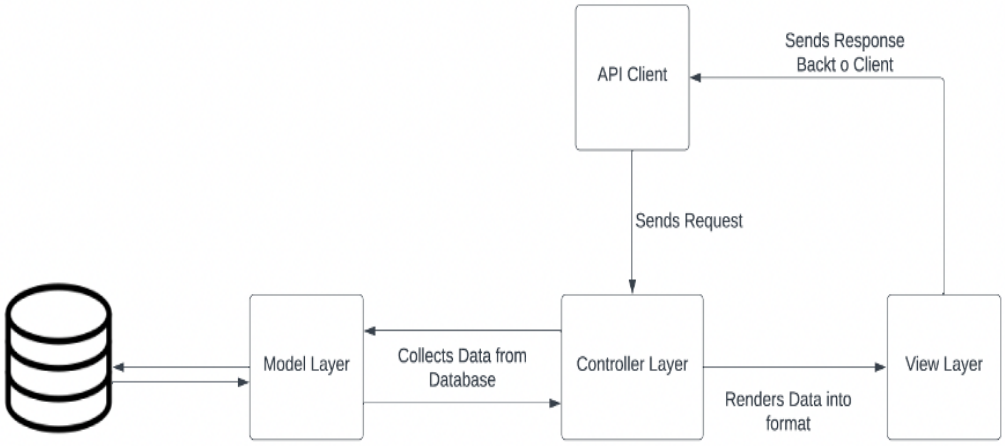
CONTENTS

1. Introduction

Redfish is a standard that uses RESTful interface semantics to access a schema-based data model to conduct management operations. It is suitable for a wide range of devices, from stand-alone servers to composable infrastructures, and to large-scale cloud environments. The problem statement for this system is to create an open-sourced version of a generic Redfish Server using OpenAPI code generation and Java. The resulting testbench would implement full role-based security, messaging, events, and dynamic as well as static data sources. The expected users for this system are the users that have access to connect to the device where this Redfish Server will be hosted and can authenticate themselves against the accounts present in the Redfish database which will be created as part of this implementation.

2. Theory of Operation

MVC Architecture



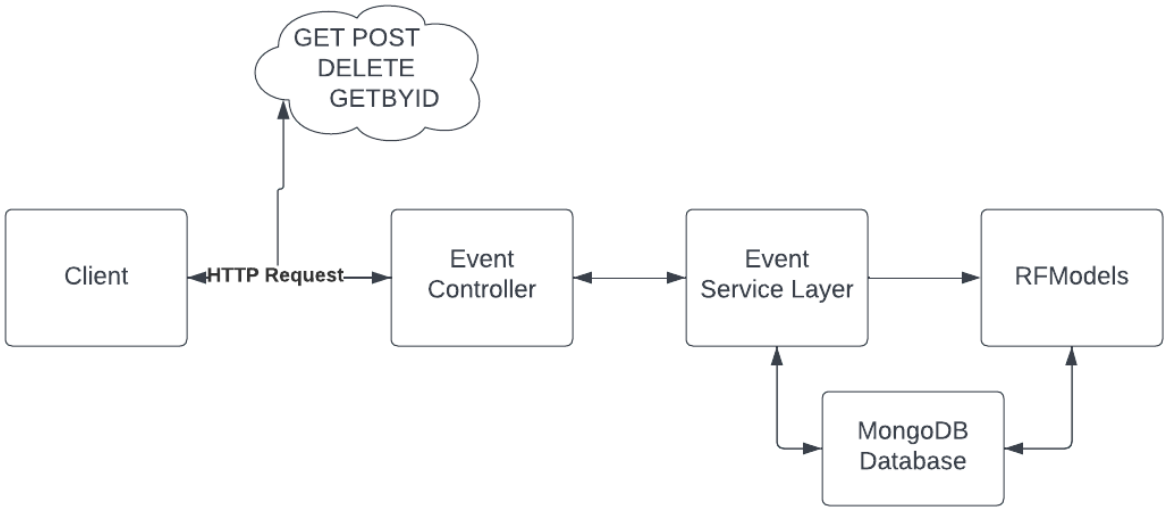
Our application follows the MVC architecture. The API client sends API requests with the JWT token to the controller. Web security Configurations make sure only authenticated requests are made. After successful authentication and authorization, the models request data/ make changes to the Mongo database (if it's POST, PUT, or PATCH) request. The Model sends back the result to the controller. The View Layer is responsible for formatting the data as specified and the final formatted result is sent back to the API client.

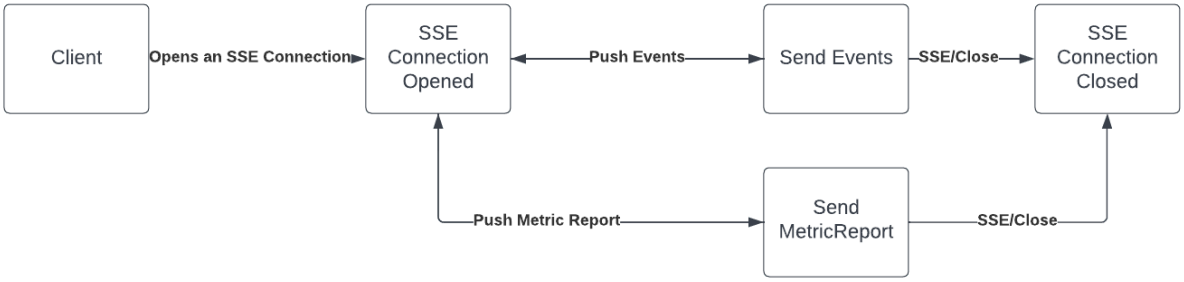
Hierarchical configuration data and logging information is stored in the application.yml file. Application.yml file helps in connecting with the MongoDB database, performing SMTP protocol, wait-time, and retry time to make the tasks async.

The system could go wrong if any service or collection data is not populated in the database on initialization or if any service or collection table contains more than one record. Also, if the password field in any record of the ManagerAccount table is null, the system login functionality will not work and the system will fail. Also, if any action is implemented which requires a Request Body, and the ActionInfo for this action is not present in the database, the system will fail.

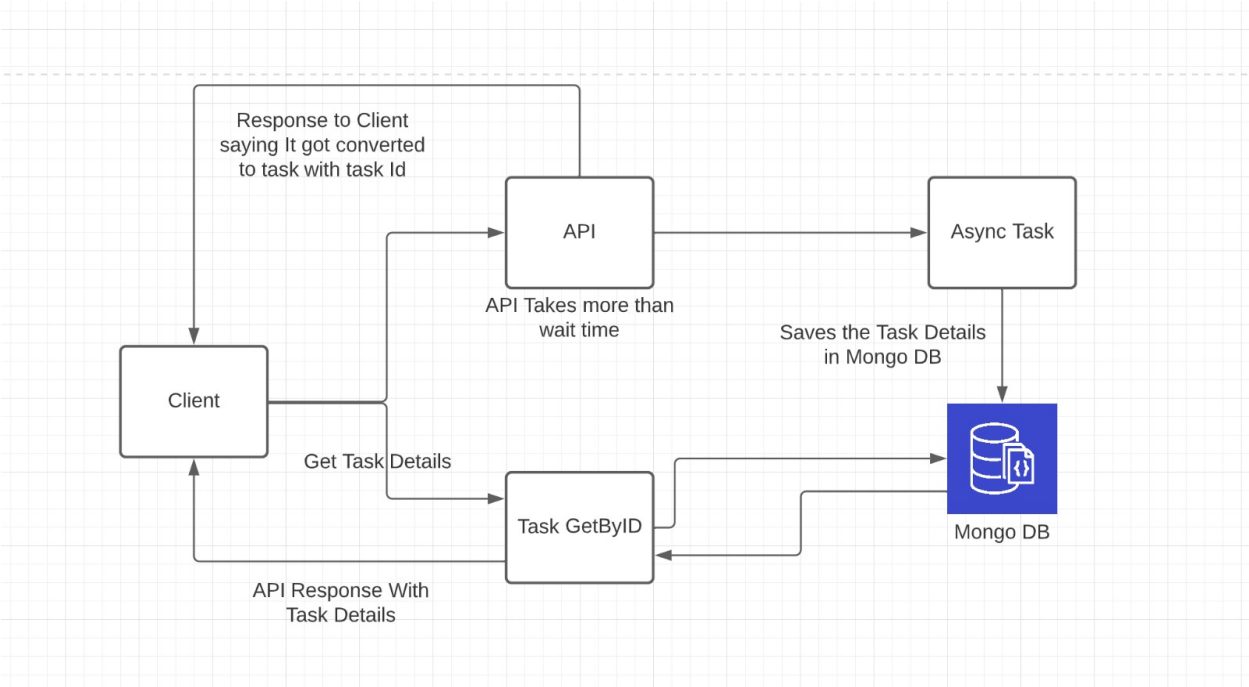
Component Diagram:

Event Service





Task Service



3. Installation Guide

3.1. Linux Installation

Follow the below steps to run the Redfish server on a Linux system.

1. Download and install Java 17 using https://docs.oracle.com/en/java/javase/17/install/installation-jdk-linux-platforms.html

2. Download and install Apache Maven 3.8.4 using https://maven.apache.org/download.cgi and https://maven.apache.org/install.html

3. Download and install MongoDB Community Server using https://www.mongodb.com/docs/manual/administration/install-on-linux/

4. Download and install Python 3.10.6 using https://www.python.org/downloads/
5. Install git by following the Linux installation steps on https://github.com/git-guides/install-git
6. Install the following Python dependencies required for this application using the below commands.
    a) pip3 install wget
    b) pip3 install pymongo
    c) pip3 install pyyaml

7. Clone the repository using

git clone https://github.com/prosm056/SER517-group11-RedfishServer.git

8. If a database named RedfishDB already exists in MongoDB, drop that database.

9. Specify the path to JSON files to be loaded into the database in the json_file_path key in the config.json file in the Initialization Scripts folder. Note that if the value of this key is empty, mockup data will be downloaded from the public-rackmount1 folder of the redfish mockups. Also, this particular mockup data has the password as null for the user Administrator in the ManagerAccount table, so the password has to be modified in the below format after running the Initialization script-

```
"Password": {
   "value": "<enter the password here>",
   "isPresent": true
 },
```

Also, there is no record present in the ActionInfo table for BiosChangePassword action. So to test the BioschangePassword action below record should be inserted in the ActionInfo table after running the Initialization script

```
{
   "@odata.type": "#ActionInfo.v1_2_0.ActionInfo",
   "Id": "BiosChangePasswordActionInfo",
   "Name": "BiosChangePassword Action Info",
   "Parameters": [
      {
         "Name": "PasswordName",
         "Required": true,
         "DataType": "String",
         "AllowableValues": [
            "AdminPassword",
            "UserPassword"
         ]
```

```
        },
        {
            "Name": "OldPassword",
            "Required": true,
            "DataType": "String"
        },
        {
            "Name": "NewPassword",
            "Required": true,
            "DataType": "String"
        }
    ],
    "Oem": {},
    "@odata.id": "/redfish/v1/Systems/437XR1138R2/Bios/BiosChangePasswordActionInfo"
}
```


10. Open the command prompt and change the directory to the Initialization Scripts folder. Run the initializeRedfishServer.py file using the below command

    python initializeRedfishServer.py

3.2. Windows Installation

Follow the below steps to run the Redfish server on a windows system.

1. Download and install Java 17 using https://www.oracle.com/java/technologies/downloads/#java17
2. Download and install Apache Maven 3.8.4 using https://maven.apache.org/download.cgi
3. Download and install MongoDB Community Server using https://www.mongodb.com/try/download/community
4. Download and install Python 3.10.6 using https://www.python.org/downloads/
5. Install the following Python dependencies required for this application using the below commands.
    a. pip3 install wget
    b. pip3 install pymongo
    c. pip3 install pyyaml
6. Download OpenSSL using https://www.openssl.org/source/ and extract the files from the zip. Specify the path of the extracted files in the Path variable of windows system environment variables.
7. Clone the repository using
   git clone https://github.com/prosm056/SER517-group11-RedfishServer.git
8. If a database named RedfishDB already exists in MongoDB, drop that database.

9. Specify the path to JSON files to be loaded into the database in the json_file_path key in the config.json file in the Initialization Scripts folder. Note that if the value of this key is empty, mockup data will be downloaded from the public-rackmount1 folder of the redfish mockups. Also, this particular mockup data has the password as null for the user Administrator in the ManagerAccount table, so the password has to be modified in the below format after running the Initialization script-

```
"Password": {
   "value": "<enter the password here>",
   "isPresent": true
 },
```

Also, there is no record present in the ActionInfo table for BiosChangePassword action. So to test the BioschangePassword action below record should be inserted in the ActionInfo table after running the Initialization script

```
{
   "@odata.type": "#ActionInfo.v1_2_0.ActionInfo",
   "Id": "BiosChangePasswordActionInfo",
   "Name": "BiosChangePassword Action Info",
   "Parameters": [
      {
         "Name": "PasswordName",
         "Required": true,
         "DataType": "String",
         "AllowableValues": [
            "AdminPassword",
            "UserPassword"
         ]
      },
      {
         "Name": "OldPassword",
         "Required": true,
         "DataType": "String"
      },
      {
         "Name": "NewPassword",
         "Required": true,
         "DataType": "String"
      }
   ],
   "Oem": {},
   "@odata.id": "/redfish/v1/Systems/437XR1138R2/Bios/BiosChangePasswordActionInfo"
}
```

10. Open the command prompt and change the directory to the Initialization Scripts folder. Run the initializeRedfishServer.py file using the below command

python initializeRedfishServer.py

3.3. Mac Installation

3.3.1. Homebrew

Make sure the brew is installed correctly on your system. You can install homebrew from **https://brew.sh/**

3.3.2.Python3

We are using python 3.9 to run our python based scripts. One can download the latest Python on their system from [1]**https://www.python.org/downloads/macos/**

3.3.3.Python3 Modules

We are going to use the following python modules to run our script which can be downloaded onto your system after installing python3 as follows

3.3.3.1.wget

pip3 install wget (using to download Redfish mock files from the redfish standards page)

3.3.3.2.pymongo

pip3 install pymongo (used to connect to MongoDB server)

3.3.3.3. pyyaml

pip3 install pyyaml (used to access yaml files)

3.3.4. Mongo Community Service

You need to install mongo community service for your database. You can download the latest mongo community service from here: [4]**https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-os-x/.** You can start your mongo service using the command: **brew services start mongodb-community@{version}.**

“”

For example, I have MongoDB-community version 5.0 installed on my system. So to start my mongo server I use the command:
brew services start mongodb-community@5.0
”

3.3.5.Java

We are going to use **Java 1.8** for our spring boot server. One can download the java onto your system from [2]**https://www.java.com/en/download/apple.jsp**
“”
The current java configuration on my system for running the application is mentioned below:
OpenJDK 17.0.4.1 2022-08-12 LTS
OpenJDK Runtime Environment (build 17.0.4.1+1-LTS)
OpenJDK 64-Bit Server VM (build 17.0.4.1+1-LTS, mixed mode, sharing)
“”

3.4.Steps to Setup Redfish Application Server

3.4.1. Downloading project from GitHub Repository

The project is currently in the GitHub Repository:
https://github.com/prosm056/SER517-group11, To clone the repository into your machine, run the following in your command line:
git clone https://github.com/prosm056/SER517-group11.git
This will clone the project in your machine.

3.4.2.Running the python scripts

1. Please follow Step 9 from Windows Installation Guide and follow the same steps.
2. In your command line, go Inside the project directory then go inside the directory Initialization_Scripts and execute the following command: **python3 initializeRedfishServer.py**

The initializeRedfishServer.py file will be responsible for the initialization of the database, generating the Redfish Standard Models, and Updating these classes to integrate with spring-boot. A detailed View of the initializeRedfishServer.py file is stated in 4.2.2.

4. Developer Guide

4.1. Services

4.1.1. Account Service

4.1.2. Account Controller

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

- To list the Account services we use the following URI

GET request: “redfish/v1/AccountService”.

  - If the GET request succeeds the client will receive a 200 OK response code.
  - If the authorization is not successful, the client will receive a message “Service recognized the credentials in the request but those credentials do not possess authorization to complete this request.”

- To list all the accounts we use the following URI
  - GET request: “redfish/v1/AccountService/Accounts”
  - If the GET request succeeds the client will receive a 200 OK response code.
  - If the authorization does not succeed the client will get a message “Service recognized the credentials in the request but those credentials do not possess authorization to complete this request.”

- To delete an account from the database
  - DELETE request: “redfish/v1/AccountService/Account”
  - If the DELETE request succeeds the client will receive a 200 OK response code.
  - If the account does not exist, the client will get a message “the requested resource was not found”.
- To add a new account to the database,
  - POST request: “redfish/v1/AccountService/Account”
  - If the account already exists, the client will get a message “Creation or update request could not be completed because it would cause a conflict in the current state of the resources that the platform supports”
- To get an account by account Id,
  - GET request: “redfish/v1/AccountService/Account/{Id}”
  - If the account does not exist, the client will get the following message, Request could not be processed because it contains invalid information.
- To get a list of all the available Roles,
  - GET request: “redfish/v1/AccountService/Roles”
  - If the authorization fails the client will get the following message, “Service recognized the credentials in the request but those credentials do not possess authorization to complete this request.”

- To get a role by role Id,
  - GET request: “redfish/v1/AccountService/Roles/{Id}”
  - If the role does not exist, the client will get the following message, Request could not be processed because it contains invalid information.
- To list all the ExternalAccountProviders,
  - GET request “redfish/v1/AccountService/ExternalAccountProviders”
- To get an ExternalAccountProvider by Id,
  - GET request “redfish/v1/AccountService/ExternalAccountProviders/{Id}”

4.1.3.Event Service

4.1.3.1. Eventing Overview

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

To create a subscription, we used the following two methods:

1. Directly HTTP POST to the subscription collection.
2. Indirectly open a server-sent events (SSE) connection for the Event Service.

4.1.3.2.Event Controller

This clause describes the API’s implemented for Event Service in EventController.java.

The client may perform an HTTP POST on a resource collection URI for subscriptions in the event service to subscribe to events. For subscription body syntax, refer to the EventDestination schema.

POST request: “localhost:8080/redfish/v1/EventService/Subscriptions”

If the subscription request succeeds, the service shall return,

1. An HTTP 201 Created status code
2. The Location header contains a URI of the newly created subscription resource.

If the format of the body in the POST request is conflicting or is not in the right format, the service shall return the HTTP 400 Bad Request.

The client may get the EventDestinationCollection upon doing a GET request on “localhost:8080/redfish/v1/EventService/Subscriptions” with a bearer token in the Authorization header.

The client shall get details of a specific subscription upon performing a GET request on "localhost:8080/redfish/v1/EventService/Subscriptions/ID" by giving the ID number at the end of the API.

The Client shall perform an HTTP DELETE request to the subscription's resource URI to unsubscribe from the events associated with this subscription.

DELETE request:
"localhost:8080/redfish/v1/EventService/Subscriptions/ID"

The Client shall receive an HTTP 404 Not Found status code upon subsequent requests to the subscription resources that have been terminated.

Indirectly open an SSE connection for the Event Service:

To open an SSE connection, the client shall do a GET request on the ServerSentEventUri property in the EventService resource that is "localhost:8080/redfish/v1/EventService/SSE". This will start the SSE connection and the client shall continuously listen to the events.

If the SSE connection is successful, the service shall:
1. Return HTTP 200 Ok status code
2. Have a Content-Type header set as text/event-stream or text/event-stream;charset=utf-8

Upon a successful SSE connection, the service shall create an EventDestination resource in the subscriptions collection for the event service to represent the connection.

If the SSE connection is unsuccessful, the service shall:
1. Return an HTTP 400 Bad Request status code.
2. Have a Content-Type header set as application/json or application/json;charset=utf-8

The Client shall use the POST API: "localhost:8080/redfish/v1/EventService/PushEvents" to send a new Event to the subscribers. Upon sending the new Event, subscribers who subscribed to the same EventType will get the message that a new event is added.

The Service shall not push an event payload greater than 1 MB.

If the subscriber's Protocol is REDFISH, then the service shall discover, connects to, and intercommunicate with a Redfish Service.

If the subscriber's Protocol is SMTP, then the service shall email the Event Message to the subscriber. The recipient address is the address that is in the Destination column in the EventDestination resource.

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

The Client shall close the connection using API
"localhost:8080/redfish/v1/EventService/SSE/close"

4.1.4. Root Service

Root Service consists of Chassis Service, Systems Service, and Managers Service.

4.1.4.1 Chassis Controller

This clause describes how to use the REST-based mechanism to view Chassis information within the Root Service.

API's:

/redfish/v1/Chassis

The Client shall get the collection of Chassis resource instances

/redfish/v1/Chassis/{chassisObjectId}

The Client shall get the physical components of a system. This resource represents the sheet-metal confined spaces and logical zones such as racks, enclosures, chassis, and all other containers. Subsystems, such as sensors, that operate outside of a system's data plane are linked either directly or indirectly through this resource. A subsystem that operates outside of a system's data plane is not accessible to software that runs on the system.

/redfish/v1/Chassis/{chassisObjectId}/Controls

The Client shall receive the collection of control resource instances

/redfish/v1/Chassis/{Id}/Controls/{controlObjectId}

The Control schema describes a control point and its properties

/redfish/v1/Chassis/{chassisObjectId}/Sensors

The Client shall receive the collection of sensor resource instances

/redfish/v1/Chassis/{chassisObjectId}/Sensors/{sensorObjectId}

The sensor schema describes a sensor and its properties

/redfish/v1/Chassis/{chassisObjectId}/EnvironmentMetrics

The Environment metrics represents the environmental metrics of a device

/redfish/v1/Chassis/{chassisObjectId}/Power

The Power schema describes power metrics and represents the properties for power consumption and power listening

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem

This PowerSubsystem schema contains the definition for the power subsystem of a chassis.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/PowerSupplies

The PowerSupplies schema contains the collection of PowerSupply resource instances.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/Batteries

The Batteries schema contains the collection of Battery resource instances.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/PowerSupplies/{powerSupplyObjectId}

The PowerSupply schema describes a power supply unit.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/PowerSupplies/{powerSupplyObjectId}/Assembly

The Assembly schema defines an assembly. Assembly information contains details about a device, such as part number, serial number, manufacturer, and production date. It also provides access to the original data for the assembly.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/PowerSupplies/{powerSupplyObjectId}/Metrics

The PowerSupplyMetrics schema contains definitions for the metrics of a power supply.

/redfish/v1/Chassis/{chassisObjectId}/ThermalSubsystem

This ThermalSubsystem schema contains the definition for the thermal subsystem of a chassis.

/redfish/v1/Chassis/{chassisObjectId}/ThermalSubsystem/ThermalMetrics

The ThermalMetrics schema represents the thermal metrics of a chassis.

/redfish/v1/Chassis/{chassisObjectId}/ThermalSubsystem/Fans

The Fans schema contains the collection of Fan resource instances.

/redfish/v1/Chassis/{chassisObjectId}/ThermalSubsystem/Fans/{fanObjectId}

The Fan schema describes a cooling fan unit for a computer system or similar devices contained within a chassis.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/Batteries/{batteryObjectId}

The Battery schema describes a battery unit, such as those used to provide systems with power during a power loss event.

/redfish/v1/Chassis/{chassisObjectId}/PowerSubsystem/Batteries/{batteryObjectId}/Metrics

The BatteryMetrics schema contains definitions for the metrics of a battery unit.

/redfish/v1/Chassis/{chassisObjectId}/TrustedComponents

The collection of TrustedComponent resource instances.

/redfish/v1/Chassis/{chassisObjectId}/TrustedComponents/{trustedComponentObjectId}/Certificates

The collection of Certificate resource instances.

/redfish/v1/Chassis/{chassisObjectId}/TrustedComponents/{trustedComponentObjectId}/Certificates/{certificateObjectId}

The Certificate schema describes a certificate that proves the identity of a component, account, or service.

/redfish/v1/Chassis/{chassisObjectId}/TrustedComponents/{trustedComponentObjectId}

The TrustedComponent resource represents a trusted device, such as a TPM.

4.1.4.2 Systems Controller

This clause describes how to use the REST-based mechanism to view Systems information within the Root Service.

API's:

/redfish/v1/Systems

The Systems schema has the collection of ComputerSystem resource instances.

/redfish/v1/Systems/{systemObjectId}

The ComputerSystem schema represents a computer or system instance and the software-visible resources, or items within the data plane, such as memory, CPU, and other devices that it can access. Details of those resources or subsystems are also linked through this resource.

/redfish/v1/Systems/{systemCollectionId}/Bios

The Bios schema contains properties related to the BIOS attribute registry. The attribute registry describes the system-specific BIOS attributes and actions for changing to BIOS settings. Changes to the BIOS typically require a system reset before they take effect. It is likely that a client finds the `@Redfish.Settings` term in this resource, and if it is found, the client makes

requests to change BIOS settings by modifying the resource identified by the `@Redfish.Settings` term.

/redfish/v1/Systems/{systemCollectionId}/Bios/Settings

The Bios schema contains properties related to the BIOS attribute registry. The attribute registry describes the system-specific BIOS attributes and actions for changing to BIOS settings. Changes to the BIOS typically require a system reset before they take effect. It is likely that a client finds the `@Redfish.Settings` term in this resource, and if it is found, the client makes requests to change BIOS settings by modifying the resource identified by the `@Redfish.Settings` term.

/redfish/v1/Systems/{systemCollectionId}/SecureBoot

The SecureBoot schema contains UEFI Secure Boot information and represents properties for managing the UEFI Secure Boot functionality of a system.

/redfish/v1/Systems/{systemCollectionId}/SecureBoot/SecureBootDatabases

The SecureBootDatabase schema contains the collection of SecureBootDatabase resource instances.

/redfish/v1/Systems/{systemCollectionId}/SecureBoot/SecureBootDatabases/{secureBootDatabaseObjectId}

The SecureBootDatabase schema describes a UEFI Secure Boot database used to store certificates or hashes.

/redfish/v1/Systems/{systemCollectionId}/Processors

The Processors schema contains the collection of Processor resource instances.

/redfish/v1/Systems/{systemCollectionId}/Processors/{processorObjectId}

The Processor schema describes the information about a single processor that a system contains. A processor includes both performance characteristics, clock speed, architecture, core count, and so on, and compatibility, such as the CPU ID instruction results.

/redfish/v1/Systems/{systemCollectionId}/Processors/{processorObjectId}/EnvironmentMetrics

The EnvironmentMetrics schema represents the environmental metrics of a device.

/redfish/v1/Systems/{systemCollectionId}/Processors/{processorObjectId}/ProcessorMetrics

The ProcessorMetrics schema contains usage and health statistics for a processor.

/redfish/v1/Systems/{systemCollectionId}/Memory

The Memory schema contains the collection of Memory resource instances

/redfish/v1/Systems/{systemCollectionId}/Memory/{memoryObjectId}

The Memory schema represents a memory device, such as a DIMM, and its configuration.

/redfish/v1/Systems/{systemCollectionId}/EthernetInterfaces

The EthernetInterfaces schema contains the collection of EthernetInterface resource instances.

/redfish/v1/Systems/{systemCollectionId}/EthernetInterfaces/{ethernetInterfaceObjectId}

The EthernetInterface schema represents a single, logical Ethernet interface or network interface, controller.

/redfish/v1/Systems/{systemCollectionId}/SimpleStorage
The SimpleStorageCollection schema contains a collection of simple storage instances.

/redfish/v1/Systems/{systemCollectionId}/SimpleStorage/{simpleStorageObjectId}

The SimpleStorage schema represents the properties of a storage controller and its directly-attached devices.

/redfish/v1/Systems/{systemCollectionId}/LogServices

The LogServiceCollection schema describes a Resource Collection of LogService instances.

/redfish/v1/Systems/{systemCollectionId}/LogServices/{logServiceObjectId}

The LogService schema contains properties for monitoring and configuring a log service. When the Id property contains `DeviceLog`, the log contains device-resident log entries that follow the

physical device when moved from system to system, and not a replication or subset of a system event log.

/redfish/v1/Systems/{systemCollectionId}/GraphicsControllers

The GraphicsControllers schema contains the collection of GraphicsController resource instances.

/redfish/v1/Systems/{systemCollectionId}/GraphicsControllers/{graphicsControllerObjectId}

The GraphicsController schema defines a graphics controller that can be used to drive one or more display devices.

/redfish/v1/Systems/{systemCollectionId}/USBControllers

The collection of USB controller resource instances.

/redfish/v1/Systems/{systemCollectionId}/USBControllers/{uSBControllerObjectId}

The USB controller schema defines a Universal Serial Bus controller.

/redfish/v1/Systems/{systemCollectionId}/Certificates

The collection of Certificate resource instances.

/redfish/v1/Systems/{systemCollectionId}/Certificates/{certificateObjectId}

The Certificate schema describes a certificate that proves the identity of a component, account, or service.

/redfish/v1/Systems/{systemCollectionId}/VirtualMedia

The VirtualMediaCollection schema describes a collection of virtual media instances.

/redfish/v1/Systems/{systemCollectionId}/VirtualMedia/{virtualMediaObjectId}

The VirtualMedia schema contains properties related to the monitoring and control of an instance of virtual media, such as a remote CD, DVD, or USB device. A manager for a system or device provides virtual media functionality.

/redfish/v1/Systems/{systemCollectionId}/VirtualMedia/{virtualMediaObjectId}/Certificates

The collection of Certificate resource instances.

/redfish/v1/Systems/{systemCollectionId}/VirtualMedia/{virtualMediaObjectId}/Certificates/{certificateObjectId}

The Certificate schema describes a certificate that proves the identity of a component, account, or service.

4.1.4.3 Managers Controller

This clause describes how to use the REST-based mechanism to view Managers' information within the Root Service.

API's:

/redfish/v1/Managers

The Mangers schema contains the Manager resource instances.

/redfish/v1/Managers/{managerObjectId}

In Redfish, a manager is a systems management entity that can implement or provide access to a Redfish service. Examples of managers are BMCs, enclosure managers, management controllers, and other subsystems that are assigned manageability functions. An implementation can have multiple managers, which might be directly accessible through a Redfish-defined interface.

/redfish/v1/Managers/{managerObjectId}/NetworkProtocol

The NetworkProtocol schema contains the network service settings for the manager

/redfish/v1/Managers/{managerObjectId}/NetworkProtocol/HTTPS/Certificates

The Certificates schema contains the collection of the certificate resource instance.

/redfish/v1/Managers/{managerObjectId}/NetworkProtocol/HTTPS/Certificates/{certificateObjectId}

The Certificate schema describes a certificate that proves the identity of a component, account, or service.

/redfish/v1/Managers/{managerObjectId}/EthernetInterfaces

The collection of EthernetInterface resource instances.

/redfish/v1/Managers/{managerObjectId}/EthernetInterfaces/{ethernetInterfaceObjectId}

The EthernetInterface schema represents a single, logical Ethernet interface or network interface controller (NIC).

/redfish/v1/Managers/{managerObjectId}/DedicatedNetworkPorts

The collection of Port resource instances.

/redfish/v1/Managers/{managerObjectId}/DedicatedNetworkPorts/{portObjectId}

The Port schema contains properties that describe a port of a switch, controller, chassis, or any other device that could be connected to another entity.

/redfish/v1/Managers/{managerObjectId}/HostInterfaces
The collection of HostInterface Resource instances.

/redfish/v1/Managers/{managerObjectId}/HostInterfaces/{hostInterfaceObjectId}
The properties associated with a Host Interface. A Host Interface is a connection between host software and a Redfish Service.

/redfish/v1/Managers/{managerObjectId}/SerialInterfaces
The collection of SerialInterface resource instances.

/redfish/v1/Managers/{managerObjectId}/SerialInterfaces/{serialInterfaceObjectId}
The SerialInterface schema describes an asynchronous serial interface, such as an RS-232 interface, available to a system or device.

/redfish/v1/Managers/{managerObjectId}/LogServices
The LogServiceCollection schema describes a Resource Collection of LogService instances.

/redfish/v1/Managers/{managerObjectId}/LogServices/Log
The LogService schema contains properties for monitoring and configuring a log service. When the Id property contains `DeviceLog`, the log contains device-resident log entries that follow the physical device when moved from system to system, and not a replication or subset of a system event log.

/redfish/v1/Managers/{managerObjectId}/SecurityPolicy
The SecurityPolicy resource provides a central point to configure the security policy of a manager.

/redfish/v1/Managers/{managerObjectId}/SecurityPolicy/SPDM/{certType}
The collection of Certificate resource instances.

/redfish/v1/Managers/{managerObjectId}/SecurityPolicy/SPDM/{certType}/{cert}
The Certificate schema describes a certificate that proves the identity of a component, account, or service.

/redfish/v1/Managers/{managerObjectId}/SecurityPolicy/TLS/{obj1}/{certType}
The collection of Certificate resource instances.

/redfish/v1/Managers/{managerObjectId}/SecurityPolicy/TLS/{obj1}/{certType}/{cert}

The Certificate schema describes a certificate that proves the identity of a component, account, or service.

4.1.5.Session Service

Session Service is used to authenticate the user and return a JWT token to the user to use as authentication for other APIs. The APIs for all session-related endpoints are written in SessionController.java.

4.1.5.1 Login

Users can log in to the Redfish Server by providing a username and password as a request body to the Sessions POST API. The API checks if the username and password are valid and then creates a record in the session table in the database. The session Id is a unique identifier generated using the UUID class. It also generates a JWT token using JWTService.java. JWT is signed using a 2048-bit private key and RS256 Algorithm.

4.1.5.2 Logout

Users can log out by sending a DELETE request to the Sessions endpoint with session id as a path parameter. This will delete the session record in the database.

4.1.5.3 Session Timeout

The JWT tokens for Redfish Server use a Session Timeout instead of token expiry to make the token invalid. The SessionTimeout field in the SessionService table in the database determines the time in minutes before the session times out. If the user sends a request using a particular JWT token, the Created time field of that session gets modified to indicate the session is being used. If the user sends a request after the session timeout expires, the session will get deleted.

4.1.5.4 Authentication

Authentication for each of the APIs takes place by calling the isUserAuthenticated method present in the APIAuthService.java. The jwt token from the request header is sent as a parameter to this method. This method calls the isTokenValid method in jwtService.java and checks if the token is valid and the session corresponding to the token has not expired and returns a boolean response.

4.1.5.5 Authorization

Authorization for each API takes place using the isUserAuthorizedForOperationType method present in APIAuthService.java. This method gets the assigned privileges for the role to which the user belongs. Then it gets the required privileges for the operation based on Entity name and operation type and checks if the user has the required privileges for that operation and returns a boolean value.

4.1.6. Task Service Overview

The task service allows the user to make long-running APIs asynchronous. These asynchronous operations are called tasks. Users can call the APIs regularly. If the APIs take more than the threshold time (called async wait time), the APIs start running in the background and the users get returned the URL for the task, and users can monitor the task status in the task monitor.

4.1.7.Task Service

4.1.6.1 Task Controller

This clause describes how to use the REST-based mechanism to view, add, and update tasks.

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

1. **To list the Task Services we use the following URI:**
   GET - redfish/v1/TaskService/
   1.1 The client performs a GET request to list Task Services.
   1.2 If the GET request succeeds the client will receive a 200 OK response code.
   1.3 If the authorization is not successful, the client will receive a message "Service recognized the credentials in the request but those credentials do not possess authorization to complete this request."

2. **To check the status of a task we use the following URI:**
   GET - redfish/v1/TaskService/Tasks/{id}
   2.1 The client performs a GET request to query the status of the operation.
   2.2 The service continues to return the HTTP 202 Accepted status code as long as the operation is in process.
   2.3 The response body contains a representation of the Task resource.
   2.4 If the authorization is not successful, the client will receive a message "Service recognized the credentials in the request but those credentials do not possess authorization to complete this request."
   2.5 If there is no task for the entered {id}, the request should return the message, "Request succeeded, but no content is being returned in the body of the response."

3. **To cancel a task we use the following URI:**
   DELETE - redfish/v1/TaskService/Tasks/{id}
   3.1 The client shall perform a DELETE request to cancel a task on the Task Resource or Task Monitor URI.
   3.2 Deleting the Task resource shall invalidate the associated task monitor. A subsequent GET request on the task monitor URI shall return either the HTTP 410 Gone or 404 Not Found status code.
   3.3 If the authorization is not successful, the client will receive a message "Service recognized the credentials in the request but those credentials do not possess authorization to complete this request."

4. **To list all the tasks running we use the following URI:**
   GET - redfish/v1/TaskService/Tasks

4.1 The client shall perform a GET request to get a list of all tasks running.
4.2 If the authorization is not successful, the client will receive a message “Service recognized the credentials in the request but those credentials do not possess authorization to complete this request.”

5. Task Service enforces the privileges described in the privilege registry required to perform operations on the Task resource, irrespective of which user or privileged context starts a task.

4.1.8. Task Monitor

As per redfish specification, some services support asynchronous operations and async operations become a task. When the URI is called, after a certain wait time, the original API returns 202 Accepted with the task object and Location of the task monitor in the response header and the operations run asynchronously in the background.

After the task is completed, the response message of the original URI is stored in the task monitor.

Approaches

1. Invoking a new Thread.class inside the original thread to continue the operation.
- This won't work because when we return 202 Accepted from the API, the parent Thread gets killed and the Java thread pool manager automatically kills the child thread.

2. Creating a Message Queue like Apache Kafka / Spark with Retry-Mechanism.
   This method is architecturally complex and difficult to implement

3. Using **“@Async”** Annotation in Spring-Boot
   This annotation allows the server to create a new Thread, separate from to parent thread and allows the process to execute in the background.

Implementation

1. Hit Any ASYNC API
2. The API will try to complete it within a specified time. **The default wait time is 5 seconds.**

3. If the API is executed within the specified time, the user gets a response from the API as per the redfish standards.
4. If the API took longer than what was expected, the server itself converts it into a Task and sends this task as response to the API call. The response header contains the **Location** and **Retry-After** parameters
    i. **Location**: This contains the URI of the task monitor
    ii. **Retry-Afte**r: This parameter contains an estimated waiting time the API might take to complete the task. **The default retry time is 60 seconds.**
5. **Task Monitor:** The task monitor API can be used to monitor the Status of the running task.

a. **Running State**: When the task is in progress, the task status is shown as "Running" and **202 HTTP Accepted** responses are shown.
b. **Completed State**: When the task is completed, the task status is shown as "Completed" and **200 OK HTTP** Status with the result of the running task is also displayed inside the "response" section.

External References for Task Monitor:

https://support.huawei.com/enterprise/en/doc/EDOC1100177343?section=j04z

4.2 Actions

For handling Redfish Actions, a generic implementation has been built in this system. The API methods for handling Actions are provided in RootController.java. There are 2 API methods defined, one for actions having a Request body and one for actions that do not require a request body. These 2 methods are being used for multiple URIs which contain the "Actions" keyword present at the end, along with the resourceType and actionName.
The classes for each action are present in the actions folder inside the services folder. The code here is implemented using the Factory design pattern. The ActionHandler interface contains the methods that will be implemented by the specific action handlers that implement the interface. There are 2 action handlers implemented as part of this system, Bios_ResetActionHandler, and Bios_ChangePasswordActionHandler. The Bios_ResetActionHandler does not require a request body, so the validateRequestBody method returns null and the setRequestBody method has no implementation. For the Bios_ChangePasswordActionHandler, the request body is validated against the ActionInfo data. If the request body is valid then it is set using the setRequestBody method and then the execute method is called from the RootController method. The code required to execute the actions should be provided in the execute method of the implementation of the ActionHandler.

While creating an implementation of actions in the future, a java class for that particular action should be created and should implement the ActionHandler interface. Post that getActionHandler method of the ActionHandlerFactory class should be modified so that RootController gets the appropriate implementation of ActionHandler.

4.2.Python Initialization Script

4.2.1. Detailed View of the Config.json File

The config.json file will contain the following

a. Credentials

1. **Json_file_path:**
   The path of JSON files used to initialize the database. If this is empty, the script will download the public-rackmount-1 mockup data and populate the database using this data.

2. **Repository_url:**
   This field contains the URL to the GitHub repo containing our redfish services

3. **Repo_name:**
   This field contains the name of our repository

4. **Mongo_Creds:**
   This Key contains the mongo credentials of the local system
   a. Mongo_Client_URL: This field contains the URL pointing to your local mongo server (Default: mongodb://localhost:27017/)
   b. Mongo_database: This Key contains the name of the database created in mongo
5. **Redfish_Creds:**
   This Key contains the credentials of the official redfish standards
   a. Mockup_url: This Key contains the URL where the REDFISH STANDARD Mockups will be available. (Default: https://www.dmtf.org/sites/default/files/standards/documents/)
   b. Mockup_File_Name: This key contains the file name where all the mockup data will be stored.
   c. Privilege_File_Name: This key contains the file name where the privileged data will be stored.
   d. Mockup_dir_Name: This key contains the name of the dir inside Mockup_File_Name where the mockup file will be stored.
6. **Schema_URL:**

This key contains the URL for the open API,yml file where all the models are located by redfish. (Default: https://redfish.dmtf.org/schemas/v1/openapi.yaml)

7. **Action_info_Schema_URL:**
   This key contains the URL for ActionInfo Schema

8. **Message_registry_Schema_URL**:
   This key contains the URL for MessageRegistry Schema

9. **Privilege_registry_Schema_URL**:
   This key contains the URL for PrivilegeRegistry Schema

10. **Generator_URL:**
    This key contains the URL where we can download the late open-API generator for converting the Redfish-provided YAML to Spring Boot Java classes.

    (Default: https://repo1.maven.org/maven2/org/openapitools/openapi-generator-cli/4.3.1/openapi-generator-cli-4.3.1.jar)

11. **Repo_all_models_path:**
    This key contains the location in your local machine where All the Redfish Standard Models will be downloaded

12. **Repo_config_path**:
    This key contains the path of the config folder where enum converters are to be created.

13. **Repo_server_pom_path:**
    This key contains the path in your local machine where your pom file will be present, where we can start the Redfish Server

14. **Repo_all_model_dir_name:**
    This key contains the name of the dir where all your Redfish models will be located

15. **Index_file_name:**
    This key contains the name of the index.json file which will be used to iterate through the mockup data and populate the DB with these data

4.2.2. Detailed View of the InitializeRedfishServer.py File

The initializeRedfishServer.py contains the following:

1. The __main__ method in the file calls the following methods
a. **loadConfigJsonFile**()
b. **cloneRepo**()
c. **downloadModels**()
d. **download_and_initialize_redfish_mockups**()
e. **start_Redish_Server**()

2. **loadConfigJsonFile**(): This method is responsible for loading the config.json file into the python file and making it available globally across all methods

3. **cloneRepo**(): This method is responsible for cloning the Repository locally. Inside it is the method **generateCertificates**()
a. **generateCertificates**(): This method is responsible for OpenSSL RSA private and public pem files used for creating the JWT token

4. **downloadModels**(): This method is responsible for downloading the YAML files for generating Redfish Models
5. **generateModels()**: This method is responsible for generating the Redfish models using YAML files and Open API generator
6. **updateRedfishModelswithMongoDBAnnotations():** This method is responsible for updating the Redfish models with MongoDB annotations and generating Enum converters.

7. **download_and_initialize_redfish_mockups**(): This method is responsible for initializing the database with data from JSON files. If the "json_file_path" value is empty in the config.json file, this function will download the mockup data from the DMTF website and populate the database with public-rackmount-1 data.

8. **start_Redish_Server**(): This method is responsible for starting the Redfish Server using maven

5. Database Guide

The database being used for this system is MongoDB. It is a NoSQL database, so there is no particular schema associated with the database. The data inserted during initialization gets distributed into tables based on the "odata. type" field in the JSON files. The MessageRegistry and PrivilegeRegistry tables are being populated using the JSON files downloaded from the Redfish website. The database name, host, and port values are specified in the application.yml file. The data from the database is pulled into the

system using Repositories for the Redfish models. The models contain the document and field names which help to connect with the database and get data from a particular table.

For future work, a separate hardware service application can be developed which will connect to this database and modify the data as per changes in the values of hardware components. If the application is using Java, the database can be connected using MongoDB packages present in the pom.xml file of this system.

6. Resources

1. https://www.python.org/downloads/macos/