Iotivity Programmer’s Guide for
– Control Manager
– Controlled Device/Controllee

– REST Framework for Control Manager

1 CONTENTS

2 Revision History 4
3 Control Manager (CM) 5
4 Control Manager Architecture and Components 5
4.1 Block Diagram 5
5 SDK API 6
5.1 ControlManager 6
5.1.1 Initializing and starting ControlManager 7
5.2 DeviceFinderListener 8
5.3 Notification Listener 8
5.4 Device Control 9
5.5 Stop Control Manager 10
6 Controlled Device/Controllee (CD) 11
7 Controlled Device Architecture and Components 11
7.1 Block Diagram 11
8 Controlled Device API 12
8.1 Key Classes 12
8.1.1 ControlleeManager 12
8.1.2 Configuration 12
8.1.3 MyDevice 13
8.1.4 SubscriptionManager & SubscriptionDatabase 13
8.2 Use of the API 13
8.2.1 Configuring the ControlleeManager & Associated Subscription Database 13
8.2.2 Populating MyDevice 13
8.2.3 Registering Resources 14
8.2.4 State Changes (start, stop) 14
8.2.5 Device Control 15
9 REST Framework For Control Manager 16

10 REST Framework For Control Manager’s Architecture and Components 16
10.1 Block Diagram 16
11 REST Framework For Control Manager API’s 18
11.1 Enumeration Data types 18
11.1.1 HttpStatusCode 18
11.1.2 HttpMethod 19
11.1.3 RestResponse 20
11.2 Structures 20
11.2.1 HttpHeaderData 20
11.2.2 HTTPReq 21
11.2.3 HTTPResponse 21
11.2.4 HttpAPIVersion 21
11.3 Classes 22
11.3.1 RestRequestHandler 22
11.4 Sample Usage 23
11.4.1 Process Flow 23
11.4.2 Initialize Rest Framework 23
11.4.3 ReceiveAndProcess REST Request 24
11.4.4 DeInit REST framework 25
11.4.5 Destroy REST framework 25
11.4.6 Combined Usage 25

2 Revision History

Revision	Date	Author(s)	Comments
v0.1	12/15/2014	Deepraj Patkar Mark Trayer Karthikeyani	Initial Release

3 Control Manager (CM)

This guide provides interaction details of ControlManager (CM) which runs on top of Iotivity base framework. The purpose of this document is to provide details for developers to understand how to use SDK APIs and how the CM works to support the APIs.

Section 4, CM Architecture and Components, describes how the main operations of the SDK API are operated in the CM. In this part, the architecture of CM is presented and the components in the architecture are described in details.

Section 5, SDK API and code snippets, describes how an application can use the CM for their purpose. We provide an Ubuntu linux based sample application which includes the functionality of discovering other controllee devices in the network, controling them and monitoring them.

4 Control Manager Architecture and Components

ControlManager runs both as Iotivity Client and Iotivity Server. ConntolManager provides SDK APIs for discovery of controlee devices, and controlling them with RESTful resource operations. ControlManager also provides subscription/notification functionality for monitoring the device operations or state changes.

4.1 Block Diagram

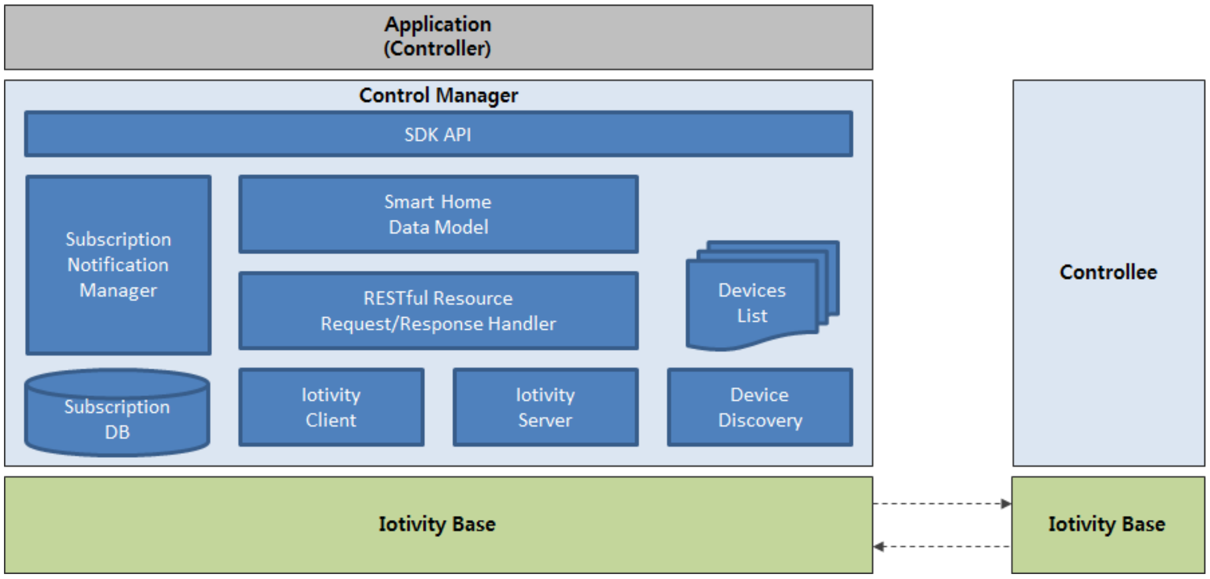


Figure 1. ControlManager Architecture

Smart Home Data Model is based on Samsung Smart Home Profile. Smart Home profile defines resource model for all the available home devices and appliances, defines hierarchical resource model and their attributes. There are common set of resources and there are function specific resources. Common set of resources provides information related to device capabilities, device configuration and supported resources. Function specific resources provides resources specific to device function like Thermostat, Light, Door, etc. With the help of data model, application developers can easily know about device information, state and control the device.

RESTful Resource Request/Response handler provides the functionality of sending the requests to controlee device by serializing it from data model to message format. And desterilizes received response to smart home data model. It uses Client module for sending requests and receiving responses.

Iotivity Client implements the client using Iotivity base framework for performing messaging with other Iotivity device as per Iotivity protocol. It supports sending request to other Iotivity device (e.g. controlee device) and receiving response from them.

Iotivity Server implements the server using Iotivity base framework for responding to requests from other Iotivity devices. ControlManager acts like a server for responding to the discovery requests from other Iotivity devices and for receiving notifications sent from other Iotivity devices.

Device Discovery uses Iotivity Discovery mechanism of the base framework for discovering other Iotivity devices. Apart from initial device discovery, ControlManager discovery mechanism retrieves device specific information and capability and maintains the discovered device's information in the devices list.

Subscription/Notification Manager provides functionality of subscribing to other devices and receiving notifications from other devices as defined in Samsung Smart Home Profile. This is RESTful subscription/notification mechanism using which ControlManager subscribes to resources of other Iotivity Devices. Notifying device, notifies to ControlManager server with the REST URI specified by ControlManager during subscription request. ControlManager also maintains the subscription information for the devices and resources it has already subscribed.

All functionality of ContolManager is accessible to developers through SDK APIs.

5 SDK API

5.1 CONTROLMANAGER

This class is starting point for developers. With this class developers can start and stop ControlManager. By invoking start function of ControlManager, it initializes the internal state and starts device discovery procedure.

5.1.1 Initializing and starting ControlManager

```
bool
TestApp::initializeAndStartControlManager ()
{
	/* Initializing ControlManager */
	sp_shp = OC::Cm::ControlManager::getInstance();
	sp_shp->setLogLevel(OC::Cm::CMLogType::CM_LOG_TYPE_DEBUG);

	sp_ControlManagerListener = new ControlManagerListener();
	sp_shp->setListener(*sp_ControlManagerListener);

	/* Configuring subscription DB path */
	OC::Cm::Configuration* p_config = sp_shp->getConfiguration();
	p_config->setSubscriptionDbPath("CMSubscriptionDB"); // Developer needs to give
actual DB Path
	sp_shp->setConfiguration(p_config);

	/* Intialize and configure self device */
	sp_myDevice = ::MyDevice::getInstance();
	/* Configure Device Details */
	sp_myDevice->setAddress(ipAddress);
	sp_myDevice->setDeviceType(OC::Cm::DeviceType_Unknown);
	sp_myDevice->setUUID(uuid.c_str()); //UUID: "E8113233-9A97-0000-0000-
000000000000"
	sp_myDevice->setDescription("Description");
	sp_myDevice->setManufacturer("Manufacturer");
	sp_myDevice->setModelID("Model ID");
	sp_myDevice->setSerialNumber("Serial Number");

	/* Configure Supported Resources */
	sp_myDevice->setSupportedResourceType("Capability");
	sp_myDevice->setSupportedResourceType("Device");
	sp_myDevice->setSupportedResourceType("Devices");
	sp_myDevice->setSupportedResourceType("Information");

	/* Set Device Discovery Listener */
	DeviceFinderLister* pDeviceFinderLister new DeviceFinderLister();
	OC::Cm::DeviceType type(OC::Cm::DeviceType::DeviceType_All);
	OC::Cm::DeviceDomain domain(OC::Cm::DeviceDomain::DeviceDomain_All);
	sp_shp->getDeviceFinder()-
>setDeviceFinderListener(domain,type,pDeviceFinderLister);

	/* Set Notification Listener */
	NotificationResponseListener* pNotificationListener = new
NotificationResponseListener();
	sp_shp->addNotificationListener(pNotificationListener);

	if (false == sp_shp->start(*sp_myDevice))
	{
		return false;
	}
```

5.2 DeviceFinderListener

This class should be implemented for receiving the notifications of the devices which are joining newly into the network or leaving from the already joined network.

```
class DeviceFinderListener : public OC::Cm::DeviceFinder::IDeviceFinderListener
{
	virtual void OnDeviceAdded( OC::Cm::Device& device )
	{
		/* This method will be invoked when a new device is discovered */
	}

	virtual void OnDeviceRemoved( OC::Cm::Device& device)
	{
		/*This method will be invoked when a device leaves the network */
	}

	virtual void OnDeviceUpdated( OC::Cm::Device& device)
	{
		/*This method will be invoked when a device details are updated */
	}

	virtual void OnDeviceError( OC::Cm::Device& device )
	{
		/*This method will be invoked when some error in device discovery */
	}
};
```

5.3 Notification Listener

This class should be implemented for receiving the notifications from the devices for which you are already subscribed.

```
class NotificationResponseListener : public OC::Cm::Notification::INotificationListener
{

	virtual void onNotificationReceived( std::string& uuid,
				      std::string& resource,
					      std::string& eventType,
				       OC::Cm::Serialization::ISerializable* notification,
					       std::string& subscriptionURI,
				      OC::Cm::Serialization::Xsd::DateTimeType* eventTime )
	{
	/* This method will be invoked on receiving a notification message from other
device */
	}

	virtual void onMulticastedNotifcationReceived( const OC::Cm::Device& device,
					       const std::string& elementType,
			       const OC::Cm::Serialization::ISerializable* notification)
	{
	/* This method will be invoked on receiving a broadcastred message from other
device */
	}
};
```

5.4 Device Control

After the device is discovered by Control Manager, the device can be controlled easily by knowing the device type and its supported resources. For controlling the device, application should set proper and acceptable attribute values according to resource data model classes.

Here is an example code for controlling Light device with LightResource.

```
LightResource* pLightResource = NULL;

/* PowerOff the Light device */
void powerOffLight()
{
	pLightResource = device->createResource(RT_LIGHT);
	if ( NULL != pLightResource )
	{
		int requestId;
		pLightResource->addResponseListener(*(new
LightResourceResponseListener()));
		pLightResource->getLight(requestId);
	}
}



/* Handle response of actions performed on Light device*/
class LightResourceResponseListener : public ILightResourceResponseListener
{
	public:
	bool onGetLight(int& requestId, int status, ::Light* pRespData)
	{
		std::string power = pRespData->mpLightPower->value;
		// Check if light is powered on
		if (power.compare("on") == 0)
		{
			int requestId;
			Light *pLight = new Light();
			pLight->mpLightPower = new OnType();
			pLight->mpLightPower->value = "Off";
			// Power off the light
			pLightResource->putLight( requestId , *pLight);
		}

		return true;
	}

	bool onPutLight(int& requestId, int status)
	{
		if (status == 204)
		{
			// Light powered off successfully
		}
	}
};
```

5.5 Stop Control Manager

Developers should stop the framework when they want to exit the application. By stopping the framework it stops the internal server and cleans up the internal components and allocations. Refer to below

```
bool
TestApp::stopFramework()
{
    /* Stop Control Manager */
    pControlManager->stop();

    /** Confirm SHP-Stop */

    OC::Cm::CMStates cmState = pControlManager->getState() ;

    if (OC::Cm::CM_STOPPED == cmState)
    {
        /* Control Manager Stopped Completely" */
    }

    /* Un-subscribe SHP Listener */
    pControlManager->removeListener(pControlManagerListener);
    if (pControlManagerListener) { delete pControlManagerListener; }

    /** Reset SHP Configuration */
    OC::Cm::Configuration *config = pControlManager->getConfiguration();

    if (NULL != config) {
        config->reset();
    }

    if (pControlManager) { delete pControlManager; }

    return true;
}
```

6 CONTROLLED DEVICE/CONTROLLEE (CD)

This guide provides details of the APIs offered by the Controlled Device/Controllee (CD) which runs on top of the Iotivity base framework. The purpose of this document is to provide details for developers to understand how to use these APIs and how the CD works to support the APIs.

Section 7, CD Architecture and Components, describes how the main operations of the API are realized in the CD. In this part, the architecture of the CD is presented and the components of the architecture are described in detail.

Section 8, Controlled Device API, and code snippets, describes how an application can use the CD for its purpose. Also include are example code segments. The CD code also provides an Ubuntu Linux based sample application which starts up an instance of a Controlled Device, registers resources, advertises presence and responds to RESTful network stimulus.

7 CONTROLLED DEVICE ARCHITECTURE AND COMPONENTS

A Controlled Device runs as an Iotivity Server. It provides APIs that allow control of device state, registration of resources with the base, and provides an application the ability to respond to received network stimulus (Create, Read, Update and Delete) as appropriate. It also provides the ability to manage received subscription requests and generate notifications in response based on local device events and state changes.

7.1 BLOCK DIAGRAM

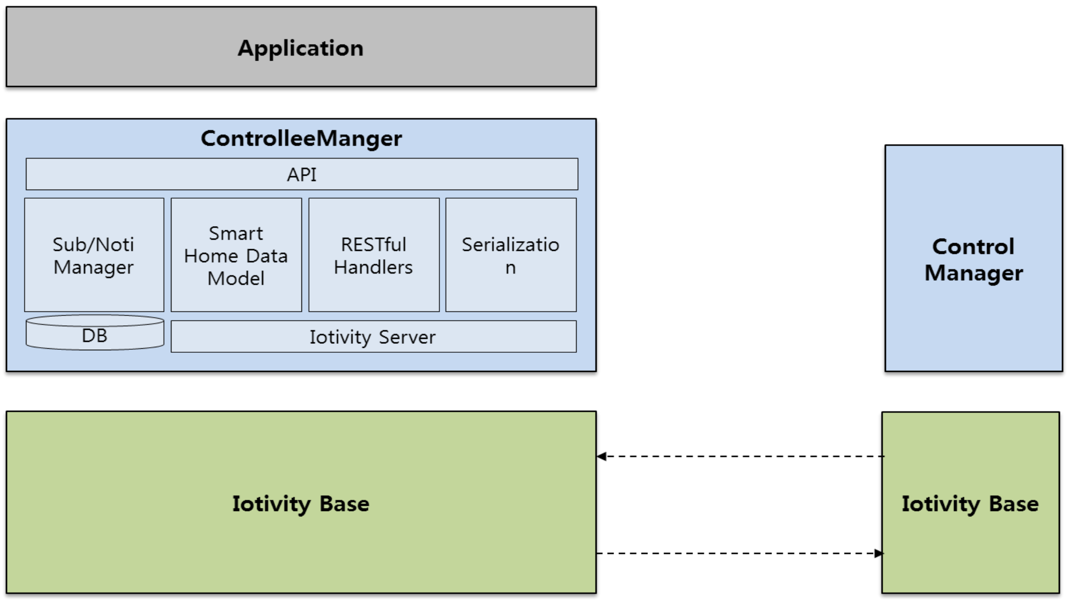


Figure 2. ControlleeManager Architecture

Smart Home Data Model is based on Samsung Smart Home Profile. Smart Home profile defines resource model for all the available home devices and appliances, defines hierarchical resource model and their attributes. There are common set of resources and there are function specific resources. Common set of resources provides information related to device capabilities, device configuration and supported resources. Function specific resources provides resources specific to device function like Thermostat, Light, Door, etc.

RESTful Resource Request/Response handler provides the application the ability to provide custom functionality in handling received resource targeted requests and the creation of responses to those requests. All resources are as identified in the Data Model.

Serialization provides the functionality of sending the responses to the Controlling device by serializing the data from data model to message format. The element also de-serializes received requests to smart home data model format.

Iotivity Server implements the server using Iotivity base framework for responding to requests from other Iotivity devices. Controllee acts as a server for responding to the resource requests from other Iotivity devices and for receiving subscriptions sent from other Iotivity devices. The Server also provides Presence indications to the network.

Subscription/Notification Manager provides functionality of handling subscriptions from other devices and generating notifications to other devices for resources of interest. This is a RESTful subscription/notification mechanism in which the ControlleeManager handles subscribe requests to its owned resources from other Iotivity Devices.

All of the functionality of the ControlleeManager is accessible to developers through APIs defined in Section 8.

8 Controlled Device API

8.1 Key Classes

8.1.1 ControlleeManager

This class is starting point for developers. With this class developers can start and stop the Controlled Device (which also starts and stops Presence) and register resources that are known by the application. By invoking the start() method of the ControlleeManager, the stack initializes internal state and starts advertising presence to external Iotivity devices.

8.1.2 Configuration

The ControlleeManager makes use of a default instance of the Configuration class to instantiate the internal data structures and dependencies. Developers can tailor these values depending on their own environment as part of the initialization of the stack.

8.1.3 MyDevice

MyDevice is a singleton that encapsulates the properties of the Controlled Device. Developers should obtain an instance of MyDevice and use the provided accessor and mutator methods to customize the fundamental device information.

8.1.4 SubscriptionManager & SubscriptionDatabase

If the application supports the ability for a client to subscribe to exposed resources and receive notifications that may be generated then the application shall create an instance of the SubscriptionManager and its associated Database and register these with the ControlleeManager. The database wrapper class shall be a concrete instantiation of the ISubscriptionDB virtual class. The framework provides an instance of such as class (SqliteSubscriptionDatabase) should the developer not want nor need to provide their own.

8.2 USE OF THE API

8.2.1 Configuring the ControlleeManager & Associated Subscription Database

The application invokes the constructor for the ControlleeManager, providing to the framework the IP address associated with the stack instance. The application can then obtain an instance of the Configuration and set or change details as appropriate to it. Should the application support subscriptions it shall also provide an instance of the SubscriptionManager.

See the below code segment for an example:

```
/* Initializing ControlleeManager */
OC::ControlleeManager *cmgr = new OC::ControlleeManager(MY_IP_ADDRESS):

/* Set configuration information */
OC::Cm::Configuration *config = cmgr->getConfiguration();
config->setAppType(OC::Cm::ApplicationType_Controllable);

/* Configuring subscription DB path */
config->setSubscriptionDbPath("MY_DB_PATH");
OC::Cm::Notification::ISubscriptionDB subDBStore =
       new OC::Cm::Notification::SqliteSubscriptionDatabase();
OC::Cm::Notification::SubscriptionManager pSub =
       new OC::Cm::Notification::SubscriptionManager(subDBStore);
cmgr->setSubscriptionManager(*pSub);
```

8.2.2 Populating MyDevice

The application should obtain an instance of MyDevice and populate the attributes as appropriate for the device under its control.

See the below code segment for an example:

```
        /* Intialize and configure device */
        sp_myDevice = ::MyDevice::getInstance();

        /* Configure Device Details */
        sp_myDevice->setAddress(MY_IP_ADDRESS);
        sp_myDevice->setDeviceType(OC::Cm::DeviceType_Unknown);
        sp_myDevice->setUUID(MY_UUID);
        sp_myDevice->setDescription("Description");
        sp_myDevice->setManufacturer("Manufacturer");
        sp_myDevice->setModelID("Model ID");
        sp_myDevice->setSerialNumber("Serial Number");
```

8.2.3 Registering Resources

The application shall register with the ControlleeManager the resources that it is exposing/supporting. All of the known resource types can be found in ResourceTypeEnum. The ControlleeManager will register the resource details with the Iotivity framework along with appropriate callback mechanisms. The application will realize code to handle reception of requests to the registered resources via realizations of the resource handlers detailed in section ZZ.

By default the stack registers Capability, Device and Devices resources with the Iotivity framework; there is no need for the application to do so. Also if subscriptions are supported the stack will additionally register Subscription(s) and Notification(s) resources with the Iotivity framework.

See the below code segment for an example:

```
        int error;
        std::list<ResourceTypeEnum> resourceType;

        resourceType.push_back(RT_Light);
        resourceType.push_back(RT_Humidity);

        OC::ControlleeStatus status = cmgr->addResources(resourceType, error);
        if (status != OC::ControlleeError)
        {
                std::cout << "Registered all resources ok" << std::endl;
        }
        else
        {
                std::cout << "Error in registering resources" << std::endl;
        }
```

8.2.4 State Changes (start, stop)

The ControlleeManager provides methods that allow the application to start or stop the device as is required. Note that starting the device also starts Presence indications on the network and stopping the device stops these indications from being sent.

See the below code segment for an example:

```
        if (false == cmgr->start())
        {
                cout << "ProgramUtils::startFramework() => "
                        << "ERROR: Failed to Start framework" << std::endl);
        }

        /* Application Logic */

        cmgr->stop();
```

8.2.5 Device Control

For every known resource that can be registered by an application the API provides a default version of a resource handler class that is itself a concrete instance of a SyncResourceHandler; methods on this class are invoked depending upon the nature of the request received by the stack that is targeted at the resource. These methods are onDELETE, onGET, onPOST and onPUT. The signatures for these methods provide status code, request data and the ability for the application to provide response data where needed. The request and response data classes are instances of a Resource class subclassed with attributes applicable to the resource being handled. This data is then serialized by the framework. The Application shall provide realizations of these methods as appropriate for the device on which the application is resident.

See the below code segment for an example showing the method signatures for a Humidity Resource:

```
         /**
          * This method will be invoked to handle a GET request.
          *
          * @param[out] statusCode   Http status code to be returned to the client
          * @param[out] respData Humidity object to be returned to the server through
the serializer. This object should not be re-initialized by a user.
          *
          * @return  @c True If the request is properly handled @n
          *          @c False In case of any error occurred
         */
         bool onGET(  int &statusCode, ::Humidity *respData);

         /**
          * This method will be invoked to handle a PUT request.
          *
          * @param[out] statusCode   Http status code to be returned to the client
          * @param[in] reqData   Received Humidity object through the deserializer. This
object should not be re-initialized by a user.
          *
          * @return  @c True If the request is properly handled @n
          *          @c False In case of any error occurred
         */
         bool onPUT(  int &statusCode, ::Humidity *reqData);
```

9 REST Framework For Control Manager

The purpose of this document is to provide details for developers to understand how to use REST framework in order to process REST requests using Control Manager. The Control Manager, on receiving the requests, passes it onto the Iotivity Base, which forwards the requests to the controllee device.

10 REST Framework For Control Manager's Architecture and Components

REST Framework for Control Manager enables the application to access and manage the RESTFul resources in a controllee device. REST framework instantiates the Control Manager module. Control Manager provides API's for controlling devices with RESTful resources. Information available in the REST request, like the request URI, the HTTP method (GET/PUT/POST/DELETE), DeviceID, etc., allows the REST Framework to use the appropriate API's in Control Manager in order to control the controllee device.

10.1 Block Diagram

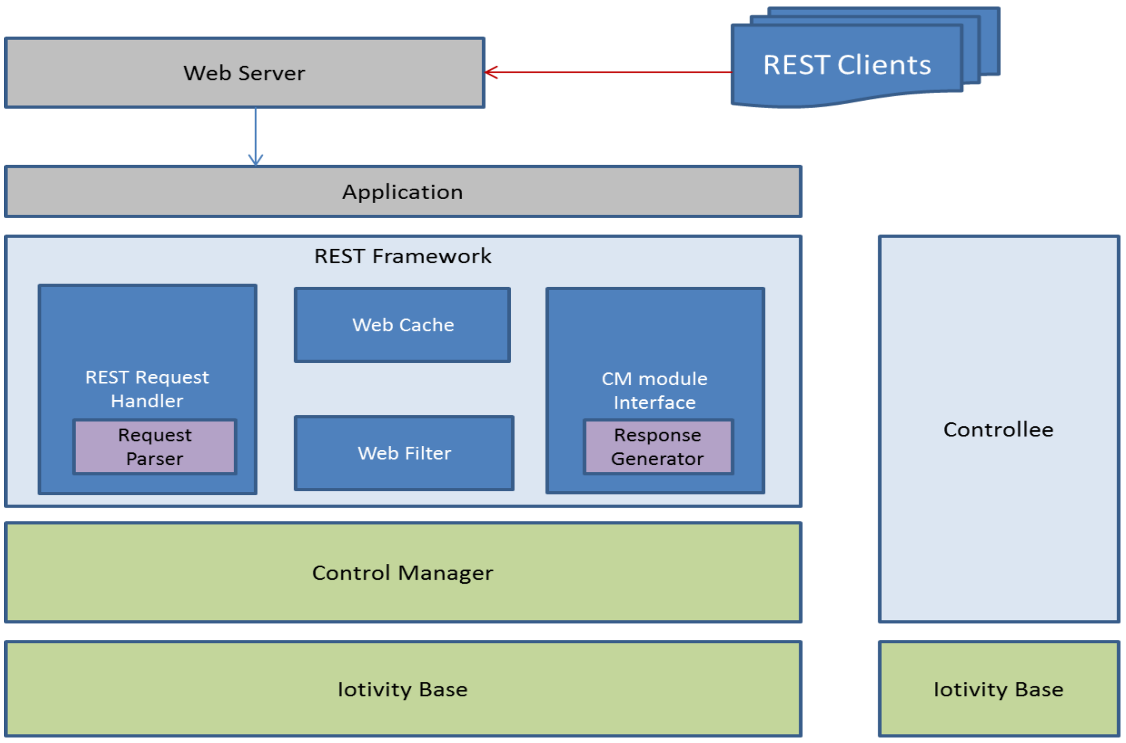


Figure 3. Architecture of REST Framework for Control Manager

REST Request handler:

Receives the REST request from Application, parses it, validates the request body(only schema validation) and forwards the request to the CM module via its interface. REST request handler will throw an error incase of invalid content (invalid URI/invalid request body, etc).

Web Cache:

A basic caching sub-module, which caches the REST requests received from application. It responds with '304 Not Modified' when there is no change in the system after the same request was processed previously.

Web Filter:

The main job of this sub-module is to parse the filter params from the request URI.

CM Module Interface:

This sub-modules acts as an interface in between REST Framework and the Control Manager. It is mainly responsible for forwarding the processed REST requests to the Control Manager. It creates and registers response listeners with the Control Manager, which uses them to respond back asynchronously. Also, a 'timeout' of 30 seconds is maintained here, after which if no response is received from Control Manager, an error is sent back to the application.

Application:

The application is the entity that instantiates the REST Framework and uses it for processing REST requests. It receives the requests from the RESTClient, parses the HTTP request to extract information (method, URI, request body, etc,.) and forward them to the REST Framework for handling.

Note:

The application can be developed in at-least two ways.

1. As a separate unit: Web Server, which is a different entity receives requests from the REST Client and forwards the request to application via some form of IPC.
2. As an integral part of the Web Server: Web Server receives the requests from the REST Client, passes it on the application directly.

11 REST FRAMEWORK FOR CONTROL MANAGER API'S

11.1 ENUMERATION DATA TYPES

11.1.1 HttpStatusCode

```
typedef enum {

        kHttpStatusCodeProcessing = 100,

        kHttpStatusCodeSuccessOk = 200,

        kHttpStatusCodeSuccessCreated = 201,

        kHttpStatusCodeAccepted = 202,

        kHttpStatusCodeSuccessNoContent = 204,

        kHttpStatusCodeSuccessPartialContent = 206,

        kHttpStatusCodeRedirectSpecialResponse = 300,

        kHttpStatusCodeRedirectMovedPermanently = 301,

        kHttpStatusCodeRedirectMovedTemporarily = 302,

        kHttpStatusCodeRedirectSeeOther = 303,

        kHttpStatusCodeRedirectNotModified = 304,

        kHttpStatusCodeRedirectTemporaryRedirect = 307,

        kHttpStatusCodeErrClientBadRequest = 400,

        kHttpStatusCodeErrClientUnauthorised = 401,

        kHttpStatusCodeErrClientForbidden = 403,

        kHttpStatusCodeErrClientNotFound = 404,

        kHttpStatusCodeErrClientNotAllowed = 405,

        kHttpStatusCodeErrClientNotAcceptable = 406,

        kHttpStatusCodeErrClientRequestTimeOut = 408,

        kHttpStatusCodeErrClientConflict = 409,

        kHttpStatusCodeErrClientLengthRequired = 411,

        kHttpStatusCodeErrClientPreconditionFailed = 412,

        kHttpStatusCodeErrClientRequestEntityTooLarge = 413,
        kHttpStatusCodeErrClientRequestUriTooLarge = 414,
        kHttpStatusCodeErrClientUnsupportedMediaType = 415,
        kHttpStatusCodeErrClientRangeNotSatisfiable = 416,
        kHttpStatusCodeErrServerInternalServerError = 500,
        kHttpStatusCodeErrServerNotImplemented = 501,
        kHttpStatusCodeErrServerBadGateway = 502,
        kHttpStatusCodeErrServerServiceUnavailable = 503,
        kHttpStatusCodeErrServerGatewayTimeOut = 504,
        kHttpStatusCodeErrServerInsufficientStorage = 507,
} HttpStatusCode;
```

11.1.2 HttpMethod

```
        typedef enum {
                HTTP_METHOD_UNSET = -1,
                HTTP_METHOD_GET,
                HTTP_METHOD_POST,
                HTTP_METHOD_HEAD,
                HTTP_METHOD_OPTIONS,
                HTTP_METHOD_PROPFIND,
                HTTP_METHOD_MKCOL,
                HTTP_METHOD_PUT,
                HTTP_METHOD_DELETE,
                HTTP_METHOD_COPY,
                HTTP_METHOD_MOVE,
                HTTP_METHOD_PROPPATCH,
                HTTP_METHOD_REPORT,
                HTTP_METHOD_CHECKOUT,
                HTTP_METHOD_CHECKIN,
                HTTP_METHOD_VERSION_CONTROL,
                HTTP_METHOD_UNCHECKOUT,
                HTTP_METHOD_MKACTIVITY,
                HTTP_METHOD_MERGE,
                HTTP_METHOD_LOCK,
                HTTP_METHOD_UNLOCK,
                HTTP_METHOD_LABEL,
                HTTP_METHOD_CONNECT
        } HttpMethod;
```

11.1.3 RestResponse

```
    typedef enum {
        kRestFwUnknownError = -1,
        kRestFwProcessRequstFailed = 0,
        kRestFwUnInitialized,
        kRestFwProcessRequstSuccess,
    } RestResponse;
```

11.2 Structures

11.2.1 HttpHeaderData

```
    typedef struct {
        enum {NUM_HEADER_PARAMS = 20};
        std::string param_name[NUM_HEADER_PARAMS];
        std::string param_value[NUM_HEADER_PARAMS];
        int flag_outheader_overwrite[NUM_HEADER_PARAMS];
        int is_this_set[NUM_HEADER_PARAMS];
        HttpHeaderData();
        ~HttpHeaderData();
    } HttpHeaderData;
```

11.2.2 HTTPReq

```
typedef struct {
        int req_type;
        std::string req_uri;
        std::string req_body;
        std::string query_parameters;
        HttpHeaderData* http_request_info;
} HTTPReq;
```

11.2.3 HTTPResponse

```
typedef struct {
        HttpStatusCode status;
        std::string response_body;
        HttpHeaderData headers;
} HTTPResponse;
```

11.2.4 HttpAPIVersion

```
typedef struct {
        unsigned int majorVersion;
        unsigned int minorVersion;
        unsigned int releaseVersion;
} HttpAPIVersion;
```

11.3 Classes

11.3.1 RestRequestHandler

RestRequestHandler is the main interface class. The application should use this to send REST requests to the REST framework.

```
class RestRequestHandler

{

  public:

        static webservice::RestRequestHandler* GetInstance(void);

        bool Init(void);

        bool DeInit(void);

        void DestroyInstance(void);

        std::string ProcessRequest(int req_type, std::string req_uri,

                        std::string query_parameters, std::string req_body,

                        const web_util::HttpHeaderData& http_request_info,

                        int* response_code, web_util::HttpHeaderData* http_response_info);

};
```

RestRequestHandler's exposes the below methods:

a. GetInstance:

   Returns the singleton instance of RestRequestHandler.

b. Init:

   Initialize the RestRequestHandler object.

c. ProcessRequest:

   Process the given REST request. This method takes 'request method, resource URI, query parameters, request body and request headers' as input parameters. Application should take care of parsing all these values from REST request.

d. DeInit:

   Un-initialize the RestRequestHandler object.

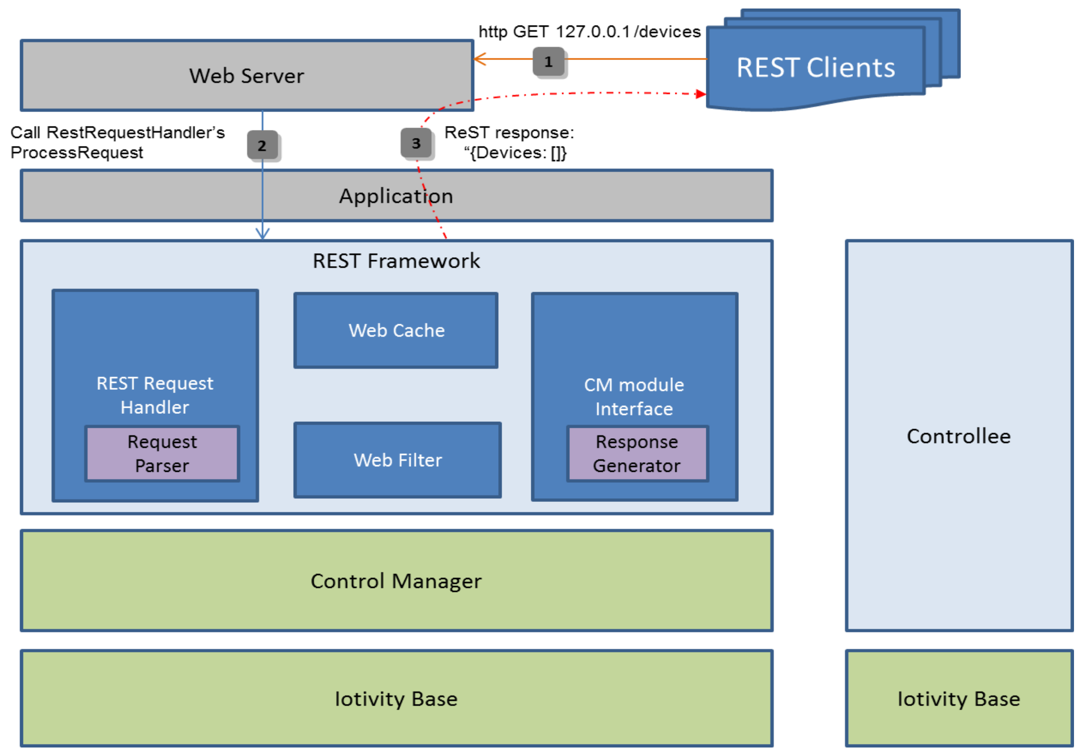
e. DestroyInstance:

   Destroys the singleton instance of RestRequestHandler.

11.4 Sample Usage

11.4.1 Process Flow

The below diagram shows a sample request flow from REST client.



11.4.2 Initialize Rest Framework

```
bool InitRestFw() {
    webservice::RestRequestHandler* rest_engine =
webservice::RestRequestHandler::GetInstance();
    if (rest_engine) {
        // initialize the Rest framework
        if (!rest_engine->Init()) {
            std::cout << "Rest framework Initialize failed" << std::endl;
            return false;
        }
        return true;
    }
    return false;
}
```

11.4.3 ReceiveAndProcess REST Request

```
void ReceiveAndrocessHttpRequest() {
    bool stop = false;
    while(!stop) {

       // Add code to receive ReST request (TODO)

       // below are the input_params (these values should be received from ReST
request)
       // Static values are given for sample usage
       // request method
       web_util::HttpMethod req_method = web_util::HTTP_METHOD_GET;
       // request URI
       std::string resource = "devices";
       // request query params
       std::string query_parameters = "";
       // request headers to be sent (This request doesn't require any request headers)
       web_util::HttpHeaderData req_hdr;

       // output_params
       // response code received
       int response_code = 500;
       // response headers
       web_util::HttpHeaderData resp_hdr;
       // response body
       std::string response = "";
       web_util::RestResponse rest_resp;

       webservice::RestRequestHandler* rest_engine =
webservice::RestRequestHandler::GetInstance();
       if (rest_engine) {
           // send request to Rest framework
           rest_resp = rest_engine->ProcessRequest(req_method, resource,
query_parameters, req_body, req_hdr, &response_code, &resp_hdr, &response);
           if (rest_resp == web_util::kRestFwProcessRequstSuccess) {
             std::cout << "Successfully processed the request" << std::endl;
           } else if (rest_resp == web_util::kRestFwUnInitializedError) {
             std::cout << "Rest framework is not initialized" << std::endl;
           } else if (rest_resp == web_util::kRestFwProcessRequstFailed) {
             std::cout << "Failed to process the request" << std::endl;
           } else {
             std::cout << "Unknown error occurred" << std::endl;
           }
       }
       if (/*any_error*/)
          stop = true;
     }
    return;
}
```

11.4.4 DeInit REST framework

```
bool DeInitRestFw() {
    webservice::RestRequestHandler* rest_engine =
webservice::RestRequestHandler::GetInstance();
    if (rest_engine) {
        // Un-initialize the Rest framework
        if (!rest_engine->DeInit()) {
            std::cout << "Rest framework Initialize failed";
            return false;
        }
    }
    return true;
}
```

11.4.5 Destroy REST framework

```
bool DestroyRestFw() {
    webservice::RestRequestHandler* rest_engine =
webservice::RestRequestHandler::GetInstance();
    if (rest_engine) {
        // destroy the Rest framework
        rest_engine->DestroyInstance();
        return true;
    }
    return false;
}
```

11.4.6 Combined Usage

```
int main() {
    // initialize rest framework
    if (!InitRestFw()) {
      std::cout << "InitRestFw failed" << std::endl;
      return 0;
    }

    // Start processing ReST requests
    ReceiveAndrocessHttpRequest();

    // un-initialize rest framework
    if (!DeInitRestFw()) {
        std::cout << "DeInitRestFw failed" << std::endl;
        return 0;
    }



    // destroy rest framework
    if (!DestroyRestFw()) {
        std::cout << "DestroyRestFw failed" << std::endl;
        return 0;
    }
    return 0;
}
```