Lab Note #3 - Resource Design Cookbook

Resources review

Resources are how applications and device endpoints exchange information and commands, through the Device Server, using the REST API. Each resource has a unique path on the server, consisting of a base path, endpoint name, and resource path. The base path for resource API access in these examples is:
http://coen296.cloudapp.net:8080/domain/endpoints/

Each hardware device that registers with the Device Server has a unique endpoint name. In the examples, the endpoint name is of the form mbedDEMO-112233445566 where the number 112233445566 represents the unique MAC ID of the endpoint.

The resource path consists of an object ID, an object instance number, and a resource ID of the form: /3301/0/5700
where 3301 is the object ID, 0 is the object instance, and 5700 is the resource ID.

The Object ID and Resource ID have semantic meaning to applications. The object ID specifies an object type and a resource template, implying that certain resources may be available. Some resources are mandatory and some are optional, depending on the Object type.

In the example project, there are resources that represent the most recent value obtained from the sensor, or the value to be used to actuate (e.g. set the brightness and color of an attached LED).

A path to a resource on the server constructed from the above example is:
http://coen296.cloudapp.net:8080/domain/endpoints/mbedDEMO-112233445566/3301/0/5700

Resource Design

Resources are designed to facilitate REST API interactions between sensors and actuators, and applications. Some of the design considerations are based on how the application is designed to interact with the device.

For example, what is the appropriate data format? What is the range of scalar data (for example 0-100%), or what kind of bit mapped, character, or vector data is appropriate? What is the smallest increment reported (1 degree, 0.1 degree, etc.); floating point or integer?

How will your application interact with the resources? One simple pattern is to periodically poll (read using GET) the sensor value. This is easy to implement and debug, and fine for many applications, but it introduces some latency to the detection of events and keeps some parts of the system busy if frequent polling is needed.

How long should the server cache responses from the resource? An application for a temperature sensor probably doesn't usually need frequent updating and can likely cache values longer than something like a sound sensor.

Another interaction pattern is asynchronous notification, which reduces latency and uses the system more efficiently in some cases, but at the expense of somewhat more difficult programming of the resource code and the application code. A presence sensor might make use of asynchronous notification in order to signal infrequent events without the need for polling.

For actuators, the application can simply use PUT to update the resource state.

Please import the example project at:
http://developer.mbed.org/teams/MBED_DEMOS/code/SensorExamples-mDS-Ethernet/

to refer to in the following discussion of examples. The project exposes and registers resources for a 10 digit LED bar, a light sensor, a presence sensor, and a gas sensor.

Data types in the example

For the example LED bar, a bitmap using a 10 character string of '1' and '0' characters is used, for example '0000000000' turns all LEDs off and '1111111111' turns them all on, similar to the LED control example in the first project.

For the gas sensor and light sensor in the example are mapped to a 0-100 scale for simplicity.

The presence sensor produces a boolean value represented by a '0' or '1', where '1' indicated a recent detection event that has a 5 second hold time.

Object and Resource ID

There are already IPSO objects for representing an Illuminance sensor, type 3301, and a presence sensor, type 3302, so these will be reused in the example. The LED bar uses a custom object type of 11101 with a new resource type 5901 to represent the 10 digit string. The gas sensor uses a custom object type 11102 with a reused resource 5700 for the current (numeric) value.

Interaction Patterns

In the example project, the Illuminance sensor and gas sensor are polled with caching disabled. When the application does a http GET from the REST API on the server, the server does a CoAP GET from the device endpoint.

The presence sensor allows GET with server caching disabled, and also allows asynchronous notification. When an application subscribes to the resource using the REST API, the server performs an OBSERVE operation, which starts asynchronous notification.

The LED bar is an actuator that is updated using PUT.

First steps in resource design

Determine the appropriate data types and interaction patterns to be used between the application and the sensors. See if there are existing objects already defined for your use case. The IPSO Smart Object Guideline defines the following objects:

Object	Object ID	Multiple Instances?
IPSO Digital Input	3200	Yes
IPSO Digital Output	3201	Yes
IPSO Analogue Input	3202	Yes
IPSO Analogue Output	3203	Yes
IPSO Generic Sensor	3300	Yes
IPSO Illuminance Sensor	3301	Yes
IPSO Presence Sensor	3302	Yes
IPSO Temperature Sensor	3303	Yes
IPSO Humidity Sensor	3304	Yes
IPSO Power Measurement	3305	Yes
IPSO Actuation	3306	Yes
IPSO Set Point	3308	Yes
IPSO Load Control	3310	Yes
IPSO Light Control	3311	Yes
IPSO Power Control	3312	Yes
IPSO Accelerometer	3313	Yes
IPSO Magnetometer	3314	Yes
IPSO Barometer	3315	Yes

Please refer to the IPSO Smart Object Guideline for details on resource IDs and data types:

http://www.ipso-alliance.org/smart-object-guidelines

If there are no suitable Object types, then it is necessary to create a new object type to represent your sensor or actuator.

ID numbers starting with 11100 are in the experimental space and do not need to be formally registered. For this project, we will create a local registry of objects using IDs starting at 11000. I have chosen to use 111XX IDs in the example.

OMA LWM2M Object ID:

Category	Object ID Range	Description
oma-label	0 – 1023	Objects defined by the Open Mobile Alliance
reserved I	1024 – 2047	Reserved for future use
ext-label	2048 – 10240	Objects defined by a 3rd party SDO
x-label	10241 – 32768	Objects defined by a vendor or individual such an object may be either private (no DDF or Specification made available) or public. These objects are optionally private this is indicated at the time to submission.

Next, define the resource type. The IPSO Smart Object Guideline lists a number of useful reusable resource types. New resource types may also be created and registered.

Resources are also registered, from a different overlapping number space:

OMA LWM2M Resource ID:

Category	ResourceID Range	Description
0 – 2047	Defined by the Object specification	
Reusable Resource ID	2048 – 32768	Registered by an Object Specification, with the Resource ID assigned by OMNA. Defined in any Object specification. Resources from thisResource ID range can be

		re-used in any Object
Reserved Resource ID	32769 -	Range or Resource IDs reserved for future use

Since there is no pre-defined experimental space for resources, we will assign internal resource IDs starting with 20000 as needed.

Make a brief resource summary for the endpoints you plan to use in your project using this example as a template:

<u>Resource Summary for the Sensor Example Demo</u>
3301/0/5700 0-100% Illuminance value from the light sensor, cache disabled
3302/0/5500 boolean presence value from the PIR sensor, observable, notify on change
11101/0/5901 10 bit string value to control LED bar, update and readback
11102/0/5700 0-100% measured value from the gas sensor, cache disabled

Next step: Implement the resources in mbed

Using the Sensor Example program imported into your workspace from above, use the following discussion as a guide to create your own resources. Also please review and refer to the earlier lab note 2 on resources.

Each resource is implemented in a separate .cpp file in the Resources folder of the project. There is a corresponding .h file to link the create_resource function to the startup code in main.cpp (more about that later).

First, create the resource implementation files for the sensors and actuators in your project, using the resource files in the example as templates for the resources in your project that most closely match the interaction patterns for the example resources.

You may choose to rename the files to something descriptive and meaningful to your project. The files that implement IPSO objects are named after the object name in the IPSO guideline.

The step by step edits for the .h and .cpp files are listed here. Perform steps 1 through 7 below for each resource:

1. Import the driver or sensor example code into the project workspace. The example uses driver code for the LED bar, imported in the folder called LED_Bar.

2. In the resource .h file, edit the definition and name the create function as shown in bold below:

(IPSO_presence.h)

```
// IPSO Presence Sensor implementation

#ifndef IPSO_PRESENCE_H
#define IPSO_PRESENCE_H

#include "nsdl_support.h"

int create_IPSO_presence_resource(sn_nsdl_resource_info_s *resource_ptr);

#endif
```

3. In the resource .cpp file, edit the resource ID and RT. The Resource Type is a descriptive string that identifies the resource. It can be any string, and here we use an experimental URN.

```
// IPSO Presence sensor resource implementation

#include "mbed.h"
#include "rtos.h"
#include "nsdl_support.h"

#define PRESENCE_RES_ID    "3302/0/5500"
#define PRESENCE_RES_RT    "urn:X-ipso:presence"
```

4. Define the I/O pins, local variables, and library objects to create an instance of the driver for the hardware and the variables you will use to create the representation of the object to communicate with the server. Since we're using JSON, the data will be encoded as arrays of bytes (char will work).

```
DigitalIn presenceSensor(D2);
uint8_t presence = 0;
uint8_t last_presence = 0;
char presenceString[1];
```

Any local code needed to interface to the driver can be added here as a function. See set_leds() in LEDbar.cpp for an example of a driver interface function that can be implemented in the resource file.

5. If using asynchronous notification, create the notifier. In this example, a thread is created (as part of the create resource function) that reads the sensor 10 times a second and reports changes as notifications. The code waits for **`pres_obs_number`** and **`pres_obs_token_ptr`** to be set by a GET with the observe option before starting notifications:

```
static uint8_t pres_obs_number = 0;
static uint8_t *pres_obs_token_ptr = NULL;
static uint8_t pres_obs_token_len = 0;
static void pres_observe_thread(void const *args)
    {
    while (true)
        {
        wait(.1);
        presence = presenceSensor.read();
        if((presence != last_presence) && pres_obs_number != 0 &&
pres_obs_token_ptr != NULL)
            {
            last_presence = presence;
            pc.printf("presence: %d\r\n", presence);
            pres_obs_number++;
            sprintf(presenceString,"%d", presence);
            if(sn_nsdl_send_observation_notification(pres_obs_token_ptr,
pres_obs_token_len, (uint8_t*)presenceString, strlen(presenceString),
&pres_obs_number, 1, COAP_MSG_TYPE_NON_CONFIRMABLE, 0) == 0)
                pc.printf("Presence observation sending failed\r\n");
            else
                pc.printf("Presence observation\r\n");
            }
        }
    }
```

In **create_IPSO_presence_resource()** the following code starts the thread.

```
    static Thread exec_thread(pres_observe_thread);
```

6. Edit the **resource callback function** where shown in bold below, to correspond with your resource implementation. This is where you handle GET and PUT operations and interface these to the sensor and actuator hardware. Also note the setup code for observe that looks for tokens and the observe option. If you don't implement observe, you can leave this code out.

```
static uint8_t presence_resource_cb(sn_coap_hdr_s *received_coap_ptr,
sn_nsdl_addr_s *address, sn_proto_info_s * proto)
{
    sn_coap_hdr_s *coap_res_ptr = 0;
    presence = presenceSensor.read();
    sprintf(presenceString,"%d", presence);
    pc.printf("presence callback\r\n");
    pc.printf("presence state %s\r\n", presenceString);

    if(received_coap_ptr->msg_code == COAP_MSG_CODE_REQUEST_GET)
        {
        coap_res_ptr = sn_coap_build_response(received_coap_ptr,
COAP_MSG_CODE_RESPONSE_CONTENT);

        coap_res_ptr->payload_len = strlen(presenceString);
        coap_res_ptr->payload_ptr = (uint8_t*)presenceString;

        coap_res_ptr->content_type_ptr = &presence_content_type;
        coap_res_ptr->content_type_len = sizeof(presence_content_type);

        coap_res_ptr->options_list_ptr = 
(sn_coap_options_list_s*)nsdl_alloc(sizeof(sn_coap_options_list_s));
        if(!coap_res_ptr->options_list_ptr)
            {
            pc.printf("cant alloc option list\r\n");
            coap_res_ptr->options_list_ptr = NULL; //FIXME report error and 
recover
            }
        memset(coap_res_ptr->options_list_ptr, 0, 
sizeof(sn_coap_options_list_s));
        coap_res_ptr->options_list_ptr->max_age_ptr = &presence_max_age;
        coap_res_ptr->options_list_ptr->max_age_len = 
sizeof(presence_max_age);

        /* The code below only used for observe */
        if(received_coap_ptr->token_ptr)
            {
            pc.printf("Token included\r\n");
            if(pres_obs_token_ptr)
                {
                free(pres_obs_token_ptr);
                pres_obs_token_ptr = 0;
                }
            pres_obs_token_ptr = (uint8_t*)malloc(received_coap_ptr-
>token_len);
            if(pres_obs_token_ptr)
                {
                memcpy(pres_obs_token_ptr, received_coap_ptr->token_ptr, 
received_coap_ptr->token_len);
                pres_obs_token_len = received_coap_ptr->token_len;
                }
            }

        if(received_coap_ptr->options_list_ptr->observe)
            {
            coap_res_ptr->options_list_ptr->observe_ptr = &pres_obs_number;
            coap_res_ptr->options_list_ptr->observe_len = 1;
            pres_obs_number++;
            }
        /* The code above only used for observe */

        sn_nsdl_send_coap_message(address, coap_res_ptr);
        nsdl_free(coap_res_ptr->options_list_ptr);
        coap_res_ptr->options_list_ptr = NULL;
        coap_res_ptr->content_type_ptr = NULL;// parser_release below tries
to free this memory
        }

    sn_coap_parser_release_allocated_coap_msg_mem(coap_res_ptr);

    return 0;
}
```

7. Edit the resource creation code. Change the variable and constant names to the local names for the resource and set GET_ALLOWED and PUT_ALLOWED to correspond with the operations implemented in the callback function.

```
int create_IPSO_presence_resource(sn_nsdl_resource_info_s *resource_ptr)
{
    static Thread exec_thread(pres_observe_thread);

    nsdl_create_dynamic_resource(resource_ptr, sizeof(PRESENCE_RES_ID)-1,
(uint8_t*)PRESENCE_RES_ID, sizeof(PRESENCE_RES_RT)-1,
(uint8_t*)PRESENCE_RES_RT, 1, &presence_resource_cb, (SN_GRS_GET_ALLOWED));
    return 0;
}
```

The Observe option is indicated by setting the boolean argument in the position before the callback pointer:
`1, &presence_resource_cb`

Setting this to ‘1’ will inform the server on registration that the resource supports asynchronous notification through the observe option. Setting to ‘0’ disables observation from the server.

That completes the edits needed in the resource file. The last step is to add a call to create the resource to the startup code in the main.cpp file.

8. In main.c, add a call to the resource creation function for each resource, so the resource is created and registered on device startup. In main.cpp, line 140:

```
   // Dynamic resources
    create_LEDbar_resource(resource_ptr);
    create_IPSO_illuminance_resource(resource_ptr);
    create_gas_sensor_resource(resource_ptr);
    create_IPSO_presence_resource(resource_ptr);
```

For each resource, insert a call to the create_XX_resource function that is defined in the resource .h file for that resource. This creates and registers the links with the Device Server, registers the callback handler for the resource which is called on CoAP requests to the resource, and performs any resource initialization, for example the observe thread initialization in the presence sensor example.

After successfully implementing the resources, the resource list should be readable in JSON format by performing a GET from the endpoint:
http://coen296.cloudapp.net:8080/domain/endpoints/mbedDEMO-112233445566

Chapter 5 of the Device Server document describes the library interfaces used in the callback handler and resource creation functions.

https://github.com/connectIOT/COEN296-IoT/blob/master/NanoService_Platform_UserGuide.pdf

Please see Chapter 4 of the above document for a description of the data model and REST API that is exposed to the web application HTTP interface.