How do I create a resource?

This article provides instructions on creating a simple resource that will be represented by a JSON object. Complex resources, as well as resources that need to be represented in other formats, will be covered in other articles. This article will also focus on creating a resource based on a database table. Since the SRP HTTP Framework ships with a sample *CONTACTS* database table, we'll use it for our demonstration purposes. All of our examples will assume the following API is being called:

`GET /contacts/1`

Resources that are not based on data in a table can still use some of the principles presented below, but there will need to be a little more effort. We will provide additional articles demonstrating how non-database resources can be created.

The Resource Object

We've already explained that a resource can be represented in any digital format, but the conventional format used by RESTful APIs is to use JSON. In our documentation we will often make reference to the **resource object**, which is short-hand for the "resource as represented by a serialized JSON object". Please note that at times we will also use the term *resource* as a synonym for *resource object*, even though a *resource* can mean much more than that.

Have it Your Way

There are three basic methods for creating a resource within the SRP HTTP Framework although no method completely excludes the use of features from the other methods. You will choose the method that best suits your needs. We will list the pros and cons of each method and provide sample code that creates the same resource using each method.

Method 1: Using the *GetDatabaseItem* Service

For simple database related resources, the GetDatabaseItem service (a member of the HTTP_Resource_Services module) can be used with minimal code. Just get the Key ID from the prepopulated *EndpointSegment* variable and pass in the name of the database table:

```
API contacts.ID.GET

   KeyID   = EndpointSegment

   HTTP_Resource_Services('GetDatabaseItem', 'CONTACTS', '', KeyID)

end api
```

This produces the following *resource object*:

```
{
   "address":"6649 N Blue Gum St",
   "birthdate":"",
   "city":"New Orleans",
   "company":"Benton, John B Jr",
   "county":"Orleans",
   "email":"jbutt@gmail.com",
   "first_name":"James",
   "last_name":"Butt",
   "notes":"",
   "picture":"\\WebAppData\\ContactPictures\\1.jpeg",
   "state":"LA",
   "url":"http://www.bentonjohnbjr.com",
   "zip":"70116",
   "phone":[
      {
         "phone_number":"(504) 621-8927",
         "phone_type":"Phone 1"
      },
      {
         "phone_number":"(504) 845-1427",
         "phone_type":"Phone 2"
      }
   ]
}
```

One of the drawbacks of calling the *GetDatabaseItem* service is that property names are formatted with underscores (i.e., how they appear in the dictionary) rather than as camel case (which is the conventional format for JSON objects). Another drawback is that the *GetDatabaseItem* service attempts to create a resource object from all column data (both physical and calculated). This might be undesirable if some of the data is meaningless to the client. Consider the *picture* property in the above resource object. It references an image file stored locally on the server, which has no value to the client. This can also be problematic if a calculated column encounters a runtime error or is dependent upon information that only exists within an OpenInsight desktop session. To avoid these problems, we will take advantage of the optional *ColumnNames* and *ItemArrayLabel* arguments. To keep our sample code simple and concise, we will limit our resource to just the *FIRST_NAME*, *LAST_NAME*, *ADDRESS*, *CITY*, *STATE*, and *ZIP* database columns:

```
API contacts.ID.GET

    KeyID           = EndpointSegment

    ColumnNames     = 'FIRST_NAME' : @FM : 'LAST_NAME' : @FM : 'ADDRESS' : @FM : 'CITY' : @FM : 'STATE' : @FM :
'ZIP'
    ItemArrayLabel  = 'firstName' : @FM : 'lastName' : @FM : 'address' : @FM : 'city' : @FM : 'state' : @FM :
'zipCode'
    HTTP_Resource_Services('GetDatabaseItem', 'CONTACTS', '', KeyID, ColumnNames, ItemArrayLabel)

end api
```

Our *resource object* now appears like this:

```
{
    "address": "6649 N Blue Gum St",
    "city": "New Orleans",
    "firstName": "James",
    "lastName": "Butt",
    "state": "LA",
    "zipCode": "70116"
}
```

Pros:

- Simple to call.
- It creates the HTTP response *Body* and *Content-Type Header* automatically.

Cons:

- Deprecated (but there are no plans to remove it).
- HATEOAS support is limited.
- Cannot include the Key ID within the resource object.

Method 2: Using the *GetObject* Service

GetDatabaseItem is an example of a high-level service. That is, it relies upon a simple interface and default behavior. Like other high-level services, *GetDatabaseItem* is built on top of slightly lower-level services. The most important one of these is ***GetObject*** (which is another member of the HTTP_Resource_Services module). Let's look at an example of an API that uses the *GetObject* service to produce the same resource object as the *GetDatabaseItem* service:

```
API contacts.ID.GET

    KeyID           = EndpointSegment

    ColumnNames     = 'FIRST_NAME' : @FM : 'LAST_NAME' : @FM : 'ADDRESS' : @FM : 'CITY' : @FM : 'STATE' : @FM :
'ZIP'
    PropertyNames   = 'firstName' : @FM : 'lastName' : @FM : 'address' : @FM : 'city' : @FM : 'state' : @FM :
'zipCode'
    // Create a JSON object in memory.
    objResource     = HTTP_Resource_Services('GetObject', 'CONTACTS', KeyID, ColumnNames, PropertyNames)
    If Error_Services('NoError') then
        // Serialize the JSON object.
        jsonResource    = HTTP_Resource_Services('GetSerializedResource', objResource)
        // Set the response body with the serialized JSON object and set the Content-Type response header.
        HTTP_Services('SetResponseBody', jsonResource, False$, 'application/hal+json')
    end else
        // There is an error condition so call the SetResponseError service.
        HTTP_Services('SetResponseError', '', '', 500, Error_Services('GetMessage'), FullEndpointURL)
    end

end api
```

In many ways, *GetObject* and *GetDatabaseItem* are the same and share some of the same arguments. *GetObect* differs from *GetDatabaseItem* in the following ways:

- *GetDatabaseItem* returns the resource object in serialized JSON (i.e., stringified JSON) whereas *GetObject* returns a handle to the JSON object.
- *GetDatabaseItem* automatically updates the HTTP response *Body* with the resource object and the *Content-Type Header* whereas *GetObject* leaves this to the calling process to handle.
- *GetDatabaseItem* will create *self* and *collection* HATEOAS links in the resource object if the optional *SelfURL* argument is used whereas *GetObject* only creates a *self* HATEOAS link.
- *GetDatabaseItem* does not do any error checking before it updates the HTTP response *Body* and the *Content-Type Header* whereas *GetObject* performs a lot of error checking.

Since *GetObject* only returns a handle to the JSON object, it is the responsibility of the calling process to serialize the resource and update the HTTP response (e.g., *Body*, *Content-Type Header*, *Status Code*, etc.). The reason developers might choose to work with the *GetObject* service instead of the *GetDatabaseItem* service (aside from the latter being deprecated) is because developers might require a little more control and ability to customize the resource object. For this reason, developers have access to several other useful services (aka *companion* services) such as *AddProperty*, *AddSubProperty*, *AddSubResource*, *AddLInkRelation*, *AddEmbeddedResources*, and *AddFormAction*.

Pros:

- Custom control over the creation of the resource object.
- Ability to include the Key ID within the resource object.
- Ability to identify a part of a Key ID as the resource ID.
- Lower-level errors can be trapped and handled as desired.
- Can use the *SRP_JSON* member services for additional functionality.

Cons:

- Caller must serialize the JSON object.
- Caller responsible for setting HTTP response elements.

There is also a ***GetObjects*** service. This is a wrapper around the *GetObject* service. It provides the developer with a *Filter* argument so multiple rows from a database table can be selected and thus converted into object handles. These object handles are returned as an @FM delimited list to the calling routine. Look at the final code sample in the *How do I add a sub-resource to a resource?* article to see how useful this can be.

Method 3: Using the *SRP_Json* Utility Function

When complete control at the lowest level is required, you'll want to use SRP_JSON to create your resource. As noted above, the handle returned by the *GetObject* service is compatible with the *SRP_JSON* function and vice-versa. Therefore, a developer can choose to start with either method and continue to use *SRP_JSON* member services and the higher level *GetObject* companion services (e.g., *AddProperty*, *AddSubProperty*, etc.) at will. One unique feature of *SRP_JSON* is its ability to interrogate the resource object using services like *GetValue*. This is useful when a resource object is generated elsewhere. Examples of how this can work will be documented in another article. For now, here is an example of how our simple resource can be created primarily using *SRP_JSON*:

```
API contacts.ID.GET

    KeyID           = EndpointSegment

    // Create a JSON object in memory.
    If SRP_JSON(objResource, 'New') then
        ContactRow  = Database_Services('ReadDataRow', 'CONTACTS', KeyID)
        If Error_Services('NoError') then
            SRP_JSON(objResource, 'SetValue', 'firstName', ContactRow<CONTACTS_FIRST_NAME$>, 'String')
            SRP_JSON(objResource, 'SetValue', 'lastName', ContactRow<CONTACTS_LAST_NAME$>, 'String')
            SRP_JSON(objResource, 'SetValue', 'address', ContactRow<CONTACTS_ADDRESS$>, 'String')
            SRP_JSON(objResource, 'SetValue', 'city', ContactRow<CONTACTS_CITY$>, 'String')
            SRP_JSON(objResource, 'SetValue', 'state', ContactRow<CONTACTS_STATE$>, 'String')
            SRP_JSON(objResource, 'SetValue', 'zipCode', ContactRow<CONTACTS_ZIP$>, 'String')
            // Serialize the JSON object and release the object from memory.
            jsonResource    = SRP_JSON(objResource, 'Stringify', 'Fast')
            SRP_JSON(objResource, 'Release')
            // Set the response body with the serialized JSON object and set the Content-Type response header.
            HTTP_Services('SetResponseBody', jsonResource, False$, 'application/hal+json')
        end else
            // There is an error reading the CONTACTS row. Probably a non-existent row.
            HTTP_Services('SetResponseError', '', '', 404, 'Contact ' : KeyID : ' does not exist.',
FullEndpointURL)
        end
    end else
        // There is an error condition so call the SetResponseError service.
        HTTP_Services('SetResponseError', '', '', 500, 'Unable to create JSON object', FullEndpointURL)
    end

end api
```

There are a few items about this sample code that need to be pointed out:

- The above represents one way of utilizing *SRP_JSON* but is certainly not the only way.
- The *CONTACTS* database row is read using the *ReadDataRow* service (a member of the Database_Services module). This service module ships with the SRP HTTP Framework but is not required. The row can be read using any statement, SSP, or custom written stored procedure.
- The success of the *ReadDataRow* service is confirmed using the *NoError* service (a member of the Error_Services module). This service module ships with the SRP HTTP Framework but is not required.

Pros:

- All the benefits of the *GetObject* service method.
- Can use the *GetObject* companion services for convenience.
- Provides access to services that can interrogate the handle to the resource object.

Cons:

- Caller must serialize the JSON object.
- Caller responsible for setting HTTP response elements.
- Caller must handle errors with *SRP_JSON* services directly.
- Object handle to *SRP_JSON* must be released directly.
- Uses generic JSON terminology rather than resource object terminology.