3.x - Preparing the Response

Obviously a web service will have little value if it cannot return content that is being requested. Even APIs that are only meant to send data to the server should still provide some content in the response. The following information will provide some basic guidelines for preparing the HTTP response.

An HTTP response is normally composed of three components:

- The HTTP status code and phrase
- One or more HTTP response header fields and values
- The HTTP body

HTTP Status Code

All responses need to set the status code. This is the primary indicator to the client how the request was processed. Web services should be designed to follow proper HTTP protocols and attempt to set the the status code that is most appropriate given the final disposition of the API logic.

Use the *SetResponseStatus* service from the HTTP_Services module to set the status code and (optionally) the phrase. This service can be called multiple times, but only the last status will be saved. One reason for using this service more than once is so a default (or expected) status can be set early in the API logic, but if an unexpected condition occurs later on, then the status can be overridden to reflect the new situation. Here is an example *SetResponseStatus* service being used:

```
HTTP_Services('SetResponseStatus', 405, HTTPMethod : ' is not valid for this service.')
```

If the *SetResponseStatus* service is never called, the SRP HTTP Framework will automatically return a status code of **`200 OK`** to ensure a well formed response.

HTTP Response Headers

Responses are not strictly required to include HTTP response headers, but specific circumstances might require particular response headers. It is beyond the scope of this documentation to explore these cases, but this www.w3.org article can provide some useful guidance.

Use the *SetResponseHeaderField* service from the HTTP_Services module to set response header fields and values. This service can be called multiple times for the same header field, but it will always replace the current value unless the *Append* argument is set to *True*. The *Append* argument would be used when the header field needs to return multiple values. For example, the *Allow* header field is used to inform the client which HTTP methods are supported by the current URL. Let's assume that the *GET* and *OPTIONS* methods are supported. There are two ways to set these values:

Option #1

```
HTTP_Services('SetResponseHeaderField', 'Allow', 'GET, OPTIONS')
```

Option #2

```
HTTP_Services('SetResponseHeaderField', 'Allow', 'GET', True$)
HTTP_Services('SetResponseHeaderField', 'Allow', 'OPTIONS', True$)
```

HTTP Body

GET requests are expected to return some form of content in the HTTP body. This could be a JSON or XML formatted database row, a Base64 encoded image, or a report published as a PDF file. Even a *POST* request or an error condition should endeavor to return content in the HTTP body that is human readable so that it is obvious how the request was processed.

Use the *SetResponseBody* service from the HTTP_Services module to set the body content. This service allows you to set the *Content-Type* response header value at the same time. This is an important header that identifies the media type being sent back. The client will use this value to determine how to properly handle the content. Alternatively, the *SetResponseHeaderField* service could be used to set the *Content-Type* response header value. For example:

Option #1

```
HTTP_Services('SetResponseBody', JSONRecord, False$, 'application/hal+json')
```

Option #2

```
HTTP_Services('SetResponseBody', JSONRecord, False$)
HTTP_Services('SetResponseHeaderField', 'Content-Type', 'application/hal+json')
```

It's a Wrap

The above services are all that are required of the API developer. Before the HTTP_MCP *controller* returns control back to the OECGI, it calls the *GetResponse* service from the HTTP_Services module. This service is responsible for collecting all of the response content and packaging it in a way that is expected by the OECGI so it can send it back to the client.