Upload XLS, XLSX or CSV data using Custom Widget (v.2.0)

** - Note: this is an early release of the custom widget and does not yet include Design Time property setting. The next update will hopefully include these components to minimize the required code snippets even further.*

1. Setting the Model ID

The first required step is to set the model ID. You can choose to hard code this value in the constructor of the JavaScript file by changing the following value:

```
//ODATA variables
this._modelId= '';
```

Or the more common scenario will be to pass this value using the API. I would suggest to set this once during initialization as this method also triggers some additional calls that enable the upload to run smoothly (i.e identifying Account Dimensions, Measures, etc.). Unless you plan to use the same widget to upload to multiple models making this call a single time is the recommended approach. The API calls to set the model ID will look as follows:

Page_1 – onInitialization

Called when the analytic application page has finished loading.

```
function onInitialization() : void
```

```
var modelId = 'Crroc214ckmi13fcd85jt2cgjf';
UploadXLS_1.setModelId(modelId);
```

2. Uploading the data

With the environment variables and model set, all you need to do to upload your Excel or CSV file is call the uploadData() method of the custom widget. This method takes the 4 required arguments defined below:

a) **Mapping** – this is a JSON object that contains the necessary mappings to ensure that your Dimension ID matches the column headers in your selected file. The format will look similar to the following:

```
var mappings = {
    'SAC_Dimension_ID':'Excel_Column_Name'
}
```

Or as below :

```
var mappings = {
    "Area":"AREA",
    "CriminalCode":"Crm Cd",
    "VictimAge":"Vict Age",
    "VictimGender":"Vict Sex",
    "VictimDescent":"Vict Descent",
    "Premise":"Premis Cd",
    "WeaponsUsed":"Weapon Used Cd",
    "Location":"LOCATION"
};
```

** - JavaScript is case-sensitive so **Area** is **NOT** the same as **AREA** and will require a mapping entry*

*** - if you are using the **Classic Account model** you **MUST** include a mapping for the hidden measure **SignedData** to the column that contains your value in the data file. e.g.*

{
"SignedData":"Value"
}

**** - if you do not need to map any values as all of your column headers match your dimension IDs then you can simply pass in an empty JSON object when calling the uploadData() function, below is an example where we are passing empty JSON objects for both the mappings and default values arguments.*

```
UploadXLS_1.uploadData({},{},false,"",false);
```

b) **Default Values** – this is a JSON object that contains any default values that you wish to pass. All dimensions included in the model MUST be explicitly referenced either in the file itself OR in the Default Values argument. This only applies to dimensions and not measures. Although multiple measures can be passed, only one is required. The syntax of the default values argument will look similar to the Mapping argument.

```
var defaultValues = {
      "Version":"public.Actual" ,
   };
```

c) **Reverse Signs** – this is a boolean that defines how you want to handle data being published to **INCOME** and **ASSET** accounts. Use the logic below to define which value to use for this argument:

- If these values are stored in the file as **negative** numbers then pass in **false**
- If these values are stored in the file as **positive** numbers then pass in **true**

d) **Sheet Name** – string representing the name of the worksheet you want to load data from. If your workbook only has a single sheet OR if you want to automatically use the first sheet then simply pass in an empty string.

e) **useFiscalDate** – boolean value that determines whether you want to interpret dates as calendar dates or fiscal dates. Set to true to pass fiscal dates and false to pass calendar dates

Your finished code will resemble something similar to the following:

```
var mappings = {
    "Area":"AREA",
    "CriminalCode":"Crm Cd",
    "VictimAge":"Vict Age",
    "VictimGender":"Vict Sex",
    "VictimDescent":"Vict Descent",
    "Premise":"Premis Cd",
    "WeaponsUsed":"Weapon Used Cd",
    "Location":"LOCATION"
};

var defaultValues = {
      "Version":"public.Actual"
   };

UploadXLS_1.uploadData(mappings,defaultValues,false,"",false);
```

At a minimum this is all that is required to upload data to your model. The next section will cover the events that are triggered by the custom widget.

3. Events and other functions

There are a handful of events and additional functions that can be used to improve the end-user experience with the custom widget.

First we will cover the events that are triggered by the widget itself. These events and their descriptions are below:

- **onFileUpload** – this is triggered after the file has been selected by the user and has been parsed by the custom widget. This event can be used in conjunction with the getTotalRows() method to return the number of rows/records that will be uploaded to SAC
- **onDataUpload** – this is triggered after the data has been uploaded to SAC. This event can be used to hide any busy indicators and to refresh the necessary data sources. You can also use the jobStatus attribute of the getUploadResult() function to see if the job was COMPLETED or COMPLETED_WITH_FAILURES. If it was the latter then you can retrieve the failed records from the UploadResult object that is returned from the getUploadResult() function as well
- **onFailedUpload** – this is triggered if the upload fails. Again using the getUploadResult() function will allow you to see the error message
- **onBatchUpload** – if the number of rows is larger than the defined chunk size (default is 100,000) then the widget will post the data in batches. After each batch this event will be triggered allowing the report designer to use the getCurrentBatchStartingNumber() function to determine how much progress has been made

Next we will cover the additional methods that are available:

- **getUploadResult()** – this function will returns an UploadResult object which contains all of the relevant data about the status of the upload request. The information contained in the object that is returned by this method call should be the primary troubleshooting/error-handling tool. Below are the properties exposed by the UploadResult object:
  - **currentStep** – the current step of the upload process
  - **jobId** – the ID of the created job
  - **modelId** – the ID of the model the data is being uploaded to
  - **mapping** – the mapping values that were used in the upload request
  - **defaultValues** – the default values that were used in the upload request
  - **data** – the data that was uploaded as parsed from the flat file
  - **httpStatus** – the Http status that was returned by the last ODATA request. Anything other than 200 usually indicates that there was an error
  - **errorMessage** – if the Http status was NOT 200 then any error message will be stored here
  - **totalNumberRowsInJob** – total number of records that were in the upload request
  - **failedNumberRows** – number of records that failed
  - **failedRows** – (see below) - returns an array of FailedRow objects representing any failed records and includes the following attributes:
    - **row** – JSON object representing the data that was attempted to be uploaded
    - **rowAsString** – a string version of the JSON object that can easily be displayed in an SAC object that allows string values (e.g. Text Area object)
    - **reason** – Reason that the record was rejected
  - **jobStatus** – returns the ENUM value for the job status (see below)
  - **body** – returns the entire body object from the last http request

- **getUploadResult().jobStatus** – this function will return an ENUM value of one of the following:
  - NOT_STARTED – before any ODATA calls have been made
  - READY_FOR_WRITE – once the job has been created but no data has been posted

    - PROCESSING – data has been posted and the job is processing the update
    - COMPLETED – data upload is complete with no errors
    - COMPLETED_WITH_FAILURES – data upload is complete, however some rows were rejected
    - FAILED – the upload failed and no records were updated

- **getUploadResult().failedRows** – as described above this function returns an array of JSON objects and can be used in conjunction with a **getUploadResult().jobStatus** call that returns a COMPLETED_WITH_FAILURES status to determine which records were rejected.

- **downloadFailedRecords()** – downloads the failed records as an Excel file

- **getChunkSize()** – returns the value set for the maximum chunk size. The default is 100,000 rows. If the file contains more records than the defined chunk size then it will automatically use batch mode.

- **getCurrentBatchStartingNumber()** – returns the starting index of the current batch being uploaded when uploading in batch mode

- **getTotalRows(sheetName)** – this function requires a string parameter representing the sheet name of the workbook. Passing an empty string will result in the function using the first sheet in the workbook. This function returns the total rows contained in the worksheet.

- **getData(sheetName)** – this function returns the entire dataset after it has been parsed by the custom widget. This is primarily used for debugging purposes OR if you need to use the Table API to publish the data (e.g. you want to load data into parent nodes and allow SAC to disaggregate the data)

- **getSheetNames()** – returns an array of the worksheets found in the workbook

- **setChunkSize()** – allows you to override the default chunk size for batch processing

Below are some example snippets of how you can use the getUploadResult() function in conjunction with a few of the exposed events to provide error handling:

Example 1: Automatically downloading the failed records as an Excel file in the onDataUpload() event

UploadXLS_1 – onDataUpload

Called when data has been uploaded to SAC.

```
function onDataUpload() : void
```

```
Application.hideBusyIndicator();
var uploadResult = this.getUploadResult();
var status = uploadResult.jobStatus;
if (status === sdk_com_sap_sample_uploadxls__2_JobStatus.COMPLETED_WITH_FAILURES){
    this.downloadFailedRecords();
}
```

Example 2: Providing a record counter in the Busy Indicator using the onFileUpload() and onBatchUpload() events

UploadXLS_1 – onFileUpload

Called when a file has been selected by the user.

```
function onFileUpload() : void
```

```
var sheets = UploadXLS_1.getSheetNames();
var records = UploadXLS_1.getTotalRows(sheets[0]);
var chunkSize = UploadXLS_1.getChunkSize();
if(records <= chunkSize){
    Application.showBusyIndicator('Uploading '+records.toString() + ' rows from the Excel file');
}else{
    Application.showBusyIndicator('Uploading first '+chunkSize.toString() + ' rows from the total '+records.toString() +' in the Excel file');
}
```

UploadXLS_1 – onBatchUpload

Called when data batch is uploaded.

```
function onBatchUpload() : void
```

```
var records = UploadXLS_1.getTotalRows('');
var chunkSize = UploadXLS_1.getChunkSize();
var currentBatchStartingIndex = UploadXLS_1.getCurrentBatchStartingNumber();
var currentBatchEndingIndex = currentBatchStartingIndex+chunkSize;
if(currentBatchEndingIndex>records){
    currentBatchEndingIndex=records;
}
if(currentBatchStartingIndex>=chunkSize){
    Application.showBusyIndicator('Uploading rows '+(currentBatchStartingIndex).toString() + ' to ' +currentBatchEndingIndex.toString()+  ' rows from the total '+records.toString() +' in the Excel file');
}
```

Example 3: Displaying an error message in the onFailedUpload() event

UploadXLS_1 – onFailedUpload

Called when data upload failed.

```
function onFailedUpload() : void
```

```
var errorMessage = this.getUploadResult().errorMessage;
Application.showMessage(ApplicationMessageType.Error,'The following error was returned: '+errorMessage);
Application.hideBusyIndicator();
```

4. More Information

Below is a list of other key points to consider when using this custom widget.

- Currently the IMPORT API only works with PUBLIC versions. Support for PRIVATE versions will be added shortly
- The IMPORT API will always PUBLISH data after upload. If you want to load data into a staging area before committing then you may want to consider adding a staging version to load the data into. Once the data has been approved for final publishing you can use a Data Action to copy data from the staging version to the final version.
- Further to the PUBLISHING note above, as with other similar use cases within SAC, users will not see newly published data if the version is in a “dirty” state. For this reason it is recommended to either PUBLISH or REVERT data PRIOR to using the file upload so that the user will see the necessary updates.

- If you are experiencing out of memory errors with large XLS/XLSX files consider saving the file as a CSV and try again
- As noted above, if you are uploading to a Classic Account model and your data point column is NOT called SignedData in your XLS or CSV file then you MUST include a mapping of your data point column to SignedData when calling the uploadData() function
- This custom widget is completely designed using client-side scripting. As such large file sizes may fail on machines with less RAM. If you're looking to support larger file sizes consistently regardless of the client environment the you may want to investigate the possibility of adapting this widget to use a server-side technology (e.g. Node.js)