ELOQUA® BULK API 1.0:

IMPORTING AND EXPORTING DATA

Table of Contents

Chapter 1: Introduction

The Bulk API provides bidirectional access to data within Eloqua®. You should have a basic familiarity with software development, RESTful APIs and the Eloqua platform in order to fully understand and use the Bulk API.

The purpose of the Bulk API

The Bulk API allows you to UPSERT or export a large number of records asynchronously by submitting requests that are processed in the background by the Eloqua application. It is therefore best suited for applications that need to interact with large sets of data, and are able to interact with them in less than real-time.

The Bulk API is intended to provide the ability to programmatically access and operate every aspect of the Eloqua system available in the User Interface. It will also enable integration between existing systems and the Eloqua platform, with the benefit of leveraging the rich data store in Eloqua and providing actionable marketing intelligence to the enterprise.

What you can do with the Bulk API

Importing and Exporting Large Data Sets

The Bulk API is used for optimizing the importing and exporting of large data sets to and from the Eloqua application. The Bulk API can be used for CRM implementation, Data Warehousing and many other purposes and has several advantages over using the standard set of SOAP or RESTful APIs. It allows programmatic access to the data definitions of imports and exports, easy management of the size and timing of the transaction, and brings increased performance while performing the action.

CRM Integration

Updating CRM systems with Eloqua data can easily be accomplished using the Bulk API by creating a polling connector service that regularly polls the API for contact record changes. For example, instead of using batch transfers of flat files for integrating Eloqua with your CRM system, you can now have near real-time updates between your systems. In addition, your CRM system can call out to Eloqua through the Bulk API and update information in your marketing database.

Data Warehousing

A data warehouse is a central repository of data which is created by integrating data from multiple disparate sources. Using the Eloqua Bulk API, you can produce export data that has been created and from a variety of business areas in your organization, such as Marketing, Sales, Customer Service, etc.

The main source of the data is cleaned, transformed, cataloged and made available for use by managers and other business professionals for data mining, online analytical processing, market research and decision support.[1]

Cloud Connectors

Cloud Connectors allow you to extend the functionality of Eloqua's automation engines (Program Builder or the Campaign Canvas). They are applications hosted outside of Eloqua that use the Cloud Connector API to retrieve members from an automation step, perform an action on those members and allow those members to continue with the rest of the automation. The Bulk API now exposes access to these actions in bulk, through imports and exports.

Examples of these actions could be to pull in data to append to a contact record from external systems, or trigger a registration for an event by passing the contact's information to an event provider. For more information, see the *Cloud Connectors* section in the resource guide in the *Code It!* section of Topliners located at: http://topliners.eloqua.com/community/code_it.

How the Bulk API is structured

The Bulk API is a RESTful API that enables
1) Management of import and export definitions
2) Read access to system metadata necessary for creating import and export definitions
3) A staging area for data
4) Control over synching data between the staging area and the marketing database.

Eloqua uses a temporary storage area between the database and external system when performing a data import or export. Staging areas in general can provide for snapshotting of complex data, quick loading of information from the snapshot, and scalability across the system by controlled interactions with the data the rest of the system is using.

As a RESTful web API, the Bulk API is a web service that is implemented using HTTP and the principles of REST. It is a collection of resources, with four defined aspects:
- The base URI for the web service (e.g. http://example.com/resources)
- The type of data that is supported by the web serviced (usually JSON but can be any other valid type providing that it is a valid hypertext standard).
- The set of operations supported by the web service using HTTP methods (e.g. GET, PUT, POST or DELETE).

[1] Reference: Wikipedia article on Data Warehousing.

- The API must be hypertext driven.

Data Formats

The Bulk API can represent entities in two different data formats: JSON (JavaScript Object Notation) or CSV (Comma-Separated Values). The Bulk API uses HTTP Headers to differentiate between these formats for a given request, as described later on in this document.

JSON is a lightweight data interchange format based on a subset of the JavaScript programming language.[2] It is text-based, human-readable, and easily produced and consumed from JavaScript as well as other programming languages. In the Bulk API, the JSON format will typically be used for the import and export definitions, as well as for retrieving system metadata.

CSV (Comma-Separated Values) is a common, relatively simple file format that is widely supported by consumer, business, and scientific applications. Among its most common uses is moving tabular data between programs that natively operate on incompatible (often proprietary and/or undocumented) formats. Since CSV is not a single, well-defined format, this works because so many programs support some variation of CSV at least as an alternative import/export format. The Bulk API supports CSV files in the RFC-4180 format,[3] with the additional requirement that a header row is required. In the Bulk API, the CSV format will typically be used for importing and exporting data.

The default data format for all endpoints is JSON; if no format is explicitly specified, JSON is assumed.

[2] See http://www.json.org/ for a JSON description and grammar.
[3] See http://tools.ietf.org/html/rfc4180 for details.

Chapter 2: API Overview and Patterns

The Bulk API is resource-oriented, rather than activity-oriented. It exposes resources and provides basic operations to create, retrieve, update, delete and search those resources. It does not expose specialized activities. It has been built in the REST architectural style (see http://en.wikipedia.org/wiki/REST). REST is an architectural style, not a specific architecture, technology or protocol. REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

The Bulk API has been built using HTTP verbs and messages for requests and responses, so it can be consumed by any application that can issue HTTP requests. This includes server languages and frameworks as well as browser-based applications.

The Bulk API does not adhere to all principles of REST. However, it is RESTful (as are most other Web APIs in the wild). The following sections discuss some of the details of how REST-style principles were interpreted within the Bulk API.

HTTP Requests

The Bulk API supports four HTTP verbs on entities and collections of entities, as shown below:

VERB	Description

GET	Retrieves one or more entities.
	
	Examples:
	
	The following request would retrieve a single contact import definition by id:
	
	`GET https://.../contact/import/123`
	
	The following request would retrieve all contacts with ‘col’ in their name:
	
	`GET https://.../contact/imports?search=col`
	
	Note: Actions on a single entity are performed by accessing the entity using singular nouns (i.e. contact, import, activity, etc.). Actions on a list of the entities are performed by using plural nouns (i.e. contacts, imports, activities, etc.)

POST	Creates a new entity.
	
	Example:
	
	The following request would create a new contact from the body of the request:
	
	`POST https://.../contact/import`
	`<JSON describing the new contact import definition>`

PUT	Updates an existing entity.
	
	Example:
	
	The following request would update the existing contact import definition with the identifier ‘123’ from the body of the request:
	
	`PUT https://.../contact/import/123`
	`<JSON describing the existing contact import definition>`

DELETE	Deletes an existing entity. Example: The following request would delete the existing contact import definition with the identifier ‘123’: `DELETE https://.../contact/import/123`

Common Uri Parameters

Most endpoints on The Bulk API accept one or more standard parameters, described below.

In the reference documentation (http://secure.eloqua.com/api/docs/Static/bulk/1.0/doc.htm), endpoints are described using Uri Templates, where "`{`" and "`}`" characters enclose the parameters to the Uri for the endpoint. An example of a Uri Template is `/contact/import/{id}`, where the name of the parameter would be `id`.

Note that URL parameters must be URL Encoded,[4] but have been left decoded for the purposes of documentation readability.

id

The `id` parameter specifies the identifier of the entity on which the operation should be performed. The value is part of the base URL, not a URL parameter.

For example, the following request would return a representation of the contact entity with the identifier 123:

```
GET https://.../contact/import/123
```

page & pageSize

The `page` parameter specifies the page of entities to return. The value can be any positive whole number. The size of the page is determined by the value of the `page` parameter. If the `page` parameter is not supplied, the default value is 1.

The `pageSize` query parameter specifies the maximum number of the `pageSize` query parameter. The value can be any integer between 1 and 1000. If the count parameter is not supplied, the default is 1,000.

For example, the following request would return the first page of 20 contact import definitions (results 1...20):

```
GET https://.../contact/imports?pageSize=20
```

The following request would return the second page of 20 contact import definitions (results 21...40):

```
GET https://.../contact/imports?page=2&pageSize=20
```

search

[4] As defined in RFC- 1738, section 2.2 (see http://www.ietf.org/rfc/rfc1738.txt for details).

The `search` query parameter specifies the search criteria to used for filtering returned entities. The value is in the format:

```
[<term> <operator>] <value>
```

Where `<term>` is the name of a property to filter on, `<operator>` is the comparison operator to apply to the property, and `<value>` is the value to compare the property with. If `<term>` and `<operator>` are not supplied, the name property is compared to the value using the equality operator.

The following request would return all contact import definitions whose name is exactly the string "Test":

```
GET https://.../contact/imports?search=Test
```

The following request would return all contact import definitions whose identifierFieldName property[5] contains the string "Email":

```
GET https://.../contact/imports?search=identifierFieldName=*Email*
```

orderBy

The `orderBy` query parameter specifies the name of the property by which the returned entities are sorted. The value is in the format:

```
<term> (ASC|DESC)?
```

Where `<term>` is the name of a property to order by. If neither `ASC` or `DESC` are specified, then the default is `ASC`.

The following requests would return a list of contact import definitions sorted by first name in ascending order, and by URI in descending order:

```
GET https://.../contact/imports?orderBy=name ASC
GET https://.../contact/imports?orderBy=uri DESC
```

If the `orderBy` query parameter is not supplied, the results are returned with a default sort (ascending).

HTTP Request Headers

Content-Type

The Content-Type header specifies the media type that the client is sending to the server. If a value is supplied and contains "application/json", the request will be interpreted as JSON. If a

[5] The identifierFieldName property is specific to import definitions. The terms that are available for searching are documented in the reference docs for each endpoint.

value is supplied and contains "text/csv", the request will be interpreted as a CSV (comma separated value). If no value is supplied, an error will occur.

In the following request the body of the request will be interpreted as CSV:

```
PUT https://.../contact/import/123
Content-Type: text/csv
<the existing contact contact data in csv>
```

In the following request the body of the request will be interpreted as JSON:

```
PUT https://.../contact/import/123
Content-Type: application/json
<the existing contact data in JSON>
```

Use of the Content-Type header is mandatory for requests with the PUT and POST verbs.

Accept

The `Accept` header specifies the media types that the client is willing to accept from the server. If a value is supplied and contains "application/json", the response will be returned in JSON. If a value is supplied and contains "text/csv", the response will be returned in CSV. If no value is supplied, or if the supplied value doesn't contain either "application/json" or "text/csv", then the response will default to JSON. Use of the `Accept` header is optional.

The following request will result in a CSV response:

```
GET https://.../contact/import/123
Accept: text/csv
```

The following requests will each result in a JSON response:

```
GET https://.../contact/import/123
Accept: application/json

GET https://.../contact/import/123
Accept: application/json; text/csv
```

HTTP Response Body

The response body will contain the entity data, returned in the format specified by the `Access` header. `DELETE` responses generally do not have a body. `GET` responses may have no response body returned when a search returned no results.

Below is a sample `GET` request, with a sample response body:

```
GET https://.../contact/import/123
```

05 PM

```
{
  "name": "Contact Import Definition ",
  "updateRule": "always",
  "fields": {
    "EmailAddressField": "{{Contact.Field(C_EmailAddress)}}",
    "FirstNameField": "{{Contact.Field(C_FirstName)}}",
    "LastNameField": "{{Contact.Field(C_LastName)}}"
  },
  "identifierFieldName": "EmailAddressField",
  "syncActions": [
    {
      "destinationUri": "/contact/list/106",
      "action": "add"
    }
  ],
  "isSyncTriggeredOnImport": true,
  "secondsToRetainData": 3600,
  "uri": "/contact/import/123",
  "createdBy": "system.user",
  "createdAt": "\/Date(1326734851377+0000)\/",
  "updatedBy": "system.user",
  "updatedAt": "\/Date(1326734851377+0000)\/"
}
```

HTTP Responses

The Bulk API returns standard HTTP response codes for every request Below is a sample of returned response codes:

The response codes are broken into a three classes of responses:

- 2xx Successful Responses – success
- 3xx Redirection Requests – errors that originate on the client
- 4xx Client Error – errors that originate on the client
- 5xx Server Error – errors in the server

HTTP Response Codes[6]

Response Code	Description	Notes
200 OK	Success *On GET/PUT verbs*	Response Body contains expected Data Transfer Object ("DTO")
201 Created	Success *On POST verb*	Response Body contains expected DTO
204 No Content	Success *On DELETE verb*	
301 Moved Permanently	Login required.	Login failure from Forms Auth only.
400 Bad Request	*One of:* There was a parsing error. There was a missing reference. There was a serialization error. There was a validation error.	Response Body contains the details of the failure in a "FailureResponse" DTO
401 Unauthorized	*One of:* Login Required. You are not authorized to make this request.	
403 Forbidden	*One of:* This service has not been enabled for your site. XSRF[7] Protection Failure.	
404 Not Found	The requested resource was not found.	
409 Conflict	There was a conflict.	Response Body contains the details of the failure in a "FailureResponse" DTO
412 Precondition Failed	The resource you are attempting to delete	

[6] See http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html for standard response code meanings.
[7] Refer to http://en.wikipedia.org/wiki/Cross-site_request_forgery for information on XRSF.

Response Code	Description	Notes
	has dependencies, and cannot be deleted.	
413 Request Entity Too Large	Storage space exceeded.	
500 Internal Server Error	The service has encountered an error.	
503 Service Unavailable	*One of:* Service is unavailable. There was a timeout processing the request.	

HTTP Response Headers

The Bulk API includes a number of HTTP response headers with the response that provide additional information on the data returned.

Content-Type

The Content-Type header specifies the media type that the server is sending to the client. If the response is JSON, the value will be "application/json". If the response is CSV, the value will be "text/csv".

Authentication

Basic Authentication

The Bulk API supports HTTP basic authentication (see http://www.ietf.org/rfc/rfc2617.txt). Each request must include an Authentication request header with a base-64-encoded value in the form:

```
siteName + '\' + username + ':' + password
```

For example, with the site name of "TESTCOMPANY", username of "testuser", and password of "testpassword", the value would be the base-64-encoded string:

```
TESTCOMPANY\testuser:testpassword
```

This means that the necessary HTTP header would be:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Forms Authentication

Note that the Bulk API also supports forms authentication and single sign-on with Eloqua 9 and Eloqua 10. If a request sent to the API includes a forms authentication cookie issued by the Eloqua application, the API will attempt to use the cookie to authenticate the user.

Chapter 3: Import Workflow

The Bulk API supports importing contacts, accounts[8], custom objects and external activities. The workflow for each of them is very similar.

The import process can be described in three steps:

1. **Define**: In this step, the definitions for various import steps are created. This includes field mappings, interaction with the staging area, and the priority of data being imported.
2. **Import**: In this step, the data is imported into the staging area.
3. **Sync**: In this step, the data from the staging area is synchronized into the Eloqua database.

Note that a demo of this workflow is available online at http://secure.eloqua.com/api/docs/Dynamic/Bulk/1.0/Import.aspx.

Step 1: Defining the Import

The import definition begins with the type of data being affected in the Eloqua system: Contact data (a contact in the Eloqua system), Account data (an account or company in the Eloqua system), Custom Object data (a custom object in Eloqua with its own set of field definitions) or External Activity data. For the purposes of this document, the examples use **Contact** data.

Defining the Field Mappings

After choosing which type of data needs to be imported, the next step is to define the field mappings. Field mappings define how the external data maps to the entity fields in the Eloqua database when a sync is ultimately executed. For example, if the incoming data has three fields FirstName, LastName and EmailAddress we might want to map these to the Eloqua Contact fields with the internal names C_FirstName, C_LastName and C_EmailAddress respectively.

The Bulk API uses ML statements (refer to http://topliners.eloqua.com/docs/DOC-2497 for more information on Eloqua Markup Language) to define mappings between external entities and Eloqua entities. Retrieving entity fields through the Bulk API will return the specific ML statements used to map those fields.

Retrieving the Fields

Searching or listing fields can be done through the `/contact/fields` endpoint using the common URI query parameters for `page`, `pageSize`, search and `orderBy`.

To find the "Email Address" field, the following request could be made:

```
GET https://.../contact/fields?search=name='Email Address'
```

[8] Referred to as **Companies** in Eloqua9.

To get a response like:

```
{
  "elements": [
    {
      "name": "Email Address",
      "internalName": "C_EmailAddress",
      "dataType": "emailAddress",
      "hasReadOnlyConstraint": false,
      "hasNotNullConstraint": false,
      "hasUniquenessConstraint": true,
      "statement": "{{Contact.Field(C_EmailAddress)}}",
      "uri": "/contact/field/100001",
      "createdAt": "\/Date(-2208970800000+0000)\/",
      "updatedAt": "\/Date(-2208970800000+0000)\/"
    }
  ],
  "total": 1,
  "pageSize": 1000,
  "page": 1
}
```

In the response above, the **`statement`** property contains the ML.

A similar call to the one above could be used to list all available Contact fields in the system, which will allow you to choose which Contact fields in Eloqua to update.

You can include a maximum of 100 fields in an import.

(Note: The list of fields from which you can choose can also be viewed in the Fields & Views section of the **Setup** > **Fields & Views** area in Eloqua 10).

Mapping the Fields

Once the desired fields are selected, the field mapping JSON object needs to be constructed. The key of the JSON object is a valid JSON string (either based on the name of the field in Eloqua or based on the incoming headers in the csv file of data that would be mapped). The key is an internal reference and does NOT need to line up with the Eloqua entity field name.

The value for the JSON object must be the ML Statement used to identify the field in Eloqua. You can find more information about MLStatement (and the Eloqua Markup Language) by referring to this article on Topliners: http://topliners.eloqua.com/docs/DOC-2497.
The following is an example of a JSON string that will import the EmailAddress, FirstName and LastName fields for a contact:

```
{
  "fields": {
    "C_EmailAddress": "{{Contact.Field(C_EmailAddress)}}",
    "C_FirstName": "{{Contact.Field(C_FirstName)}}",
    "LastName": "{{Contact.Field(C_LastName)}}"
  }
}
```

Mapping Related Fields

In certain scenarios is also possible to UPSERT related entity fields through an import. This works in cases where the primary entity type that you are importing has a 1-to-1 or n-to-1 relationship with the secondary entity type that you want to reference. In both of the previously mentioned cases it is possible to get a unique match between the entities. For example, when importing Custom Objects, it is possible to import fields to the linked Contact entity.

Choosing the Identifier Field

Since imports work by UPSERTing data, the system needs a way to identify existing data in the system. For contact imports, the most common scenario is to use the email address as the identifier field.

From the perspective of an import definition, this works by choosing one of the fields in the field mappings as the identifier field. To do this, set the `identifierFieldName` property to the key value of the fields object.

For example, to update the above example and use the email address field as the identifier field, you would use the following JSON:

```
{
  "identifierFieldName": "C_EmailAddress",
  "fields": {
    "C_EmailAddress": "{{Contact.Field(C_EmailAddress)}}",
    "C_FirstName": "{{Contact.Field(C_FirstName)}}",
    "LastName": "{{Contact.Field(C_LastName)}}"
  }
}
```

Defining Sync Actions (optional)

Sync actions are extra actions that are taken at time of synchronization, or in the case of imports, after the data has been synchronized from the staging area into the Eloqua database. Sync actions are not required for an import.

For example, a sync action could be defined to add contacts to an existing contact list in Eloqua. Alternatively, an action could be defined to update a Cloud Connector member status in a Cloud Connector Instance[9]. Another action could be used to update the subscription status of a contact with respect to an Email Subscription Group.

Defining Import Characteristics

The following properties are also used to configure the import:

name	The name of this import (maximum 100 characters).
importPriorityUri	The priority of data that gets imported from this import definition. You can use the default, or in the workflow, you can search through all the available /import/priorities, and select the one you wish to use. Note that these are the same import priorities that are used for other integrations.
isSyncTriggeredOnImport	This property is used to indicate whether a sync is automatically created by the system when data is imported. The system automatically sets this field to **true** by default. If you set it as **false**, after you push the data into the system, the data will not be processed until you explicitly create a sync for this import.
updateRule	This property indicates when to update values in the Eloqua database from values imported with this import definition. The available options are: **Always** — Always update **ifNewIsNotNull** — Update if the new value is not blank **ifExistingIsNull** — Update if the value is not blank **useFieldRule** — Use the rule defined at the field level **ifNewIsEmailAddress** — Update if it's a valid email address
secondsToRetainData	The amount of time that unsynched data from this import will be stored in the staging area, in seconds. Valid values are anything from 3600 (1 hour) to 1209600 (2 weeks).
secondsToAutoDelete	The amount of time (in seconds) before the import definition will be automatically deleted. If the property is not set then automatic deletion will not occur. This is typically used for one-time-use export definitions.

[9] Also called a Cloud Connector Step.

Step 2: Import the Data

Step 2 is to push the data into the system. Bulk API supports data in JSON or CSV format. The following is an example of importing data in the application/json format:

```
[
  {
  "C_EmailAddress": "john.doe@company.com",
  "C_FirstName": "John",
  "LastName": "Doe"
  },
  {
  "C_EmailAddress": "mary.smith@company.com",
  "C_FirstName": "Mary",
  "LastName": "Smith"
  }
]
```

The following is an example of importing data using the text/csv format:

```
C_EmailAddress,C_FirstName,LastName
john.doe@company.com,John,Doe
mary.smith@company.com,Mary,Smith
```

Important: Ensure that the CSV header and JSON key names correspond to the field mapping names in contact import definition.

Step 3: Synchronize the Imported Data

The last step is to synchronize the data from the staging area into the Eloqua database. The import definition property (`isSyncTriggeredOnImport`) can be configured to automatically sync the data (**true**). If you set this value to **false**, the data will not be processed (imported) until you explicitly create the sync for the import.

Depending on the activity in your Eloqua instance, you may wish to manually initiate the sync operation, rather than set it as an automatic process in order to mitigate any potential performance issues, or to avoid importing (or exporting) large sets of data when other system-intensive processes are running. If you have generally smaller sets of contact data to import, it is a good idea to set the sync to be automated, in which case no manual intervention is necessary (can be changed later if required).

The following is an example of how to sync the data from the import into the Eloqua database. To sync `https://../contact/import/123:`

```
POST https://.../sync

{
  "syncedInstanceUri": "/contact/import/123"
}
```

The response to the above HTTP request will include a uri for the sync, for example:

```
{
  "syncedInstanceUri": "/contact/import/123",
  "uri": "/sync/456"
}
```

Then, to check the status of the sync, perform a GET operation, for example:

```
GET https://.../sync/456
```

The log showing the results of the sync processing is available at the results endpoint, for example:

```
GET https://.../sync/456/results
```

Chapter 4: Export Workflow

The steps for performing a data export in Eloqua are similar to those described in the previous chapter for importing records.

The Bulk API supports exporting contacts, accounts[10] and custom objects. The workflow for each of them is very similar.

The export process can be described in three steps:

1. **Define**: In this step, you create the definition for the export. This includes field mappings, interaction with the staging area, any applicable filters, the priority of data being exported, etc.
2. **Sync**: In this step, you snapshot the data from the database into the staging area.
3. **Export**: In this step, you export the data to an external system.

Note that a demo of this workflow is available online at http://secure.eloqua.com/api/docs/Dynamic/Bulk/1.0/Export.aspx.

Step 1: Defining the Export

As with the import process, you must first create the definition. The export definition begins with the type of data being affected in the Eloqua system: Contact data (a contact in the Eloqua system), Account data (an account or company in the Eloqua system) or Custom Object data (a custom object in Eloqua with its own set of field definitions). For the purposes of this document, **Contact** data is used.

Defining the Field Mappings

After choosing which type of data to be exported, the next step is to define the field mappings. The field mappings will define how the data coming from the fields in the Eloqua database will be represented in the final export when a sync is executed. For example, if the data to be exported has three fields FirstName, LastName and EmailAddress, you might want to map the Eloqua Contact fields with the internal names C_FirstName, C_LastName and C_EmailAddress to these fields, respectively.

To define the mapping, you need to identify which fields exist in the Eloqua system and what the ML Statement is for referencing those fields. If you retrieve the contact fields through the Bulk API, they will have the ML Statement as a property to make the mapping easier. More detailed information on the Eloqua Markup Language (EML) can be found here: http://topliners.eloqua.com/docs/DOC-2497.

Note that exports from Eloqua are limited to a maximum of 100 fields.

Retrieving the Fields

[10] Referred to as **Companies** in Eloqua9.

Searching or listing fields can be done through the /contact/fields endpoint using the common Uri parameters for page, pageSize, search and orderBy.

For example, to find the “Email Address” field the following request could be made:

```
GET https://.../contact/fields?search=name='Email Address'
```

To get a response like:

```
{
  "elements": [
    {
      "name": "Email Address",
      "internalName": "C_EmailAddress",
      "dataType": "emailAddress",
      "hasReadOnlyConstraint": false,
      "hasNotNullConstraint": false,
      "hasUniquenessConstraint": true,
      "statement": "{{Contact.Field(C_EmailAddress)}}",
      "uri": "/contact/field/100001",
      "createdAt": "\/Date(-2208970800000+0000)\/",
      "updatedAt": "\/Date(-2208970800000+0000)\/"
    }
  ],
  "total": 1,
  "pageSize": 1000,
  "page": 1
}
```

In the above resonse, the **`statement`** property contains the ML statement.

A similar call to the one above could be used to list all available Contact fields in the system, which will allow you to choose which Contact fields in Eloqua to update.

(Note: The list of fields from which you can choose can also be viewed in the Fields & Views section of the **Setup** > **Fields & Views** area in Eloqua 10).

Mapping the Fields

Once the desired fields are selected, the field mapping JSON object needs to be constructed. The key of the JSON object is a valid JSON string (either based on the name of the field in Eloqua or based on the incoming headers in the csv file of data that would be mapped). The key is an internal reference and does NOT need to line up with the field name of the Eloqua field.

The value for the JSON object must be the ML Statement used to identify the field in Eloqua. You can find more information about MLStatement (and the Eloqua Markup Language) by referring to this article on Topliners: http://topliners.eloqua.com/docs/DOC-2497.

The following is an example of a JSON string that will import the EmailAddress, FirstName and LastName fields for a contact:

```
{
  "fields": {
    "C_EmailAddress": "{{Contact.Field(C_EmailAddress)}}",
    "C_FirstName": "{{Contact.Field(C_FirstName)}}",
    "LastName": "{{Contact.Field(C_LastName)}}"
  }
}
```

Note that with export definitions, more complex results can be returned. For example, fields can be concatenate together. So, if you desired a single result with both first and last name, you could use a definition like the following:

```
{
  "fields": {
    "C_EmailAddress": "{{Contact.Field(C_EmailAddress)}}",
    "FirstandLastName": "{{Contact.Field(C_FirstName)}}
{{Contact.Field(C_LastName)}}"
  }
}
```

Mapping Related Fields

Finally, it is also possible to access fields for related entities through an export. This works in the scenario where the primary entity type that you are exporting has a 1-to-1 or an n-to-1 relationship with the secondary entity type that you want to reference. In both of the previously mentioned cases it is possible to get a unique match between the entities. For example, when exporting Contact data, it is possible to export fields from the linked Account data.

To do this, we'd first need to get the ML statement for the Account fields that were important. This would be done in exactly the same way as getting the ML Statement for the Contact fields (but using the /account/fields endpoint instead).

Assuming that we wanted to use the Company Name field for the Account, this has the ML Statement `{{Account.Field(M_CompanyName)}}`. The modification we need to make is to tell the system to start with the Contact, and to look at the linked Account, and then at the Account's field. This is done through a . notation, as follows:

```
{{Contact.Account.Field(M_CompanyName)}}
```

Defining Sync Actions (optional)

Sync actions are extra actions that are taken at time of synchronization, or in the case of exports, after the data has been snapshotted into the staging area from the Eloqua database. Sync actions are not required for your export.

For example, a sync action could be defined to update a Cloud Connector member status in a Cloud Connector Instance[11].

Defining a Data Filter (optional)

If you wish to filter the data during the export in order to return results related to a specific set of contacts, you can use one of four filtering methods:

1. Inclusion in a Contact List or Contact Segment or Contact Filter
2. Subscribed or Unsubscribed to an Email Subscription Group
3. Active or Pending status in a Cloud Connector Instance
4. Field value Comparison

Filters offer different ways to limit the export to a subset of your Eloqua Contact database. Note that some of these filters only apply to Contact data. Filtering your data for the export is not required.

Defining Export Characteristics

The following properties are also used to configure the export:

name	The name of this export (maximum 100 characters).
secondsToRetainData	The amount of time that unsynched data from this export will be stored in the staging area, in seconds. Valid values are anything from 3600 (1 hour) to 1209600 (2 weeks).
secondsToAutoDelete	The amount of time (in seconds) before the export definition will be automatically deleted. If the property is not set then automatic deletion will not occur. This is typically used for one-time-use export definitions.

[11] Also called a Cloud Connector Step.

Step 2: Synchronize Data to the Staging area

The second step is to synchronize the data from the Eloqua database to the staging area, so it can be exported.

The following is an example of how to sync the data from the Eloqua database to be available to export. To sync `https://../contact/export/123:`

```
POST https://.../sync

{
  "syncedInstanceUri": "/contact/export/123"
}
```

The response to the above HTTP request will include a uri for the sync, for example:

```
{
  "syncedInstanceUri": "/contact/export/123",
  "uri": "/sync/456"
}
```

Then, to check the status of the sync, perform a GET operation, for example:

```
GET https://.../sync/456
```

The log showing the results of the sync processing is available at the results endpoint, for example:

```
GET https://.../sync/456/results
```

Step 3: Export the Data

Once the sync has succeeded, you can retrieve the data from the staging area. The Bulk API supports exporting data in JSON or CSV format. The `Accept` HTTP header is used to determine which format the data will be exposed in.

To retrieve the first page of JSON data for `/contact/export/123`, you could use the following HTTP request:

HTTP Header:

```
Accept: application/json

GET https://.../contact/export/123/data?page=1&pageSize=50000
```

To retrieve the second page of CSV data for `/contact/export/123`, you could use the following HTTP request:

HTTP Header:

```
Accept: text/csv
```

```
GET https://.../contact/export/123/data?page=2&pageSize=50000
```

Note that `pageSize` for exports is limited to 50000 records.

Appendix

Other Reference Materials

For more information on the Bulk API, the following resources are available. Please note that you must be logged into Eloqua prior to being able to access either link.

https://secure.eloqua.com/api/docs/Static/Bulk/1.0/index.htm
https://secure.eloqua.com/api/docs/Dynamic/Bulk/1.0/Reference.aspx

The reference document for the Eloqua Markup Language (EML) is located here:

http://topliners.eloqua.com/docs/DOC-2497