API Technical Guide: Advanced Data Load

Cheetah Messaging

Table of Contents

1 Introduction

Purpose

The purpose of this document is to provide an overview of the ADVANCED DATA LOAD API within the Cheetah Messaging platform. This document discusses the intended use and potential benefits of the ADVANCED DATA LOAD API, and provides technical details for how to implement the API.

Overview

The ADVANCED DATA LOAD endpoint allows you to use an API request message to send data to update multiple tables in your database, including relational data. This endpoint is the "engine" that powers the ADVANCED EVENT TRIGGER (AET) endpoint. AET allows clients to trigger the deployment of an existing Event-based Campaign in Messaging through the use of an API request message. The ADVANCED DATA LOAD endpoint is essentially just the "load data" component of AET, without the "Campaign trigger" component.

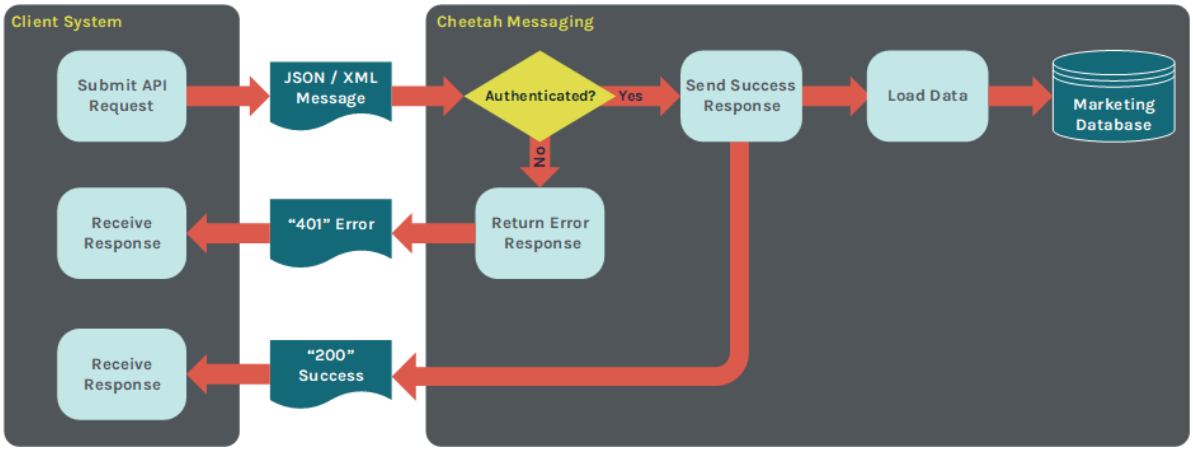
For more information on AET, please see the *Advanced Event Trigger API Technical Guide*, or the Cheetah Messaging Online Help.

This endpoint requires authentication using OAuth 2.0, and supports JSON and XML messages.

The URLs for this endpoint are:

- **North America**: https://aet.eccmp.com/services2/api/Data
- **Europe**: https://api.ccmp.eu/services2/api/Data
- **Japan**: https://api.marketingsuite.jp/services2/api/Data

The following diagram depicts the basic processing flow for loading data via the ADVANCED DATA LOAD endpoint.



Pre-requisites

In order to use the ADVANCED DATA LOAD endpoint to write data to your database, the following assets must be defined within your Messaging account:

- **Table** -- The destination Table(s) where you want to load your import data must be created, and must include all the necessary Fields, as well as any necessary Joins to other Tables. These Tables, Fields, and Joins can be created within the Messaging application.

Methods

The ADVANCED DATA LOAD endpoint supports the following HTTP method:

- **POST**: Submit data to load or write to your database.

Authentication

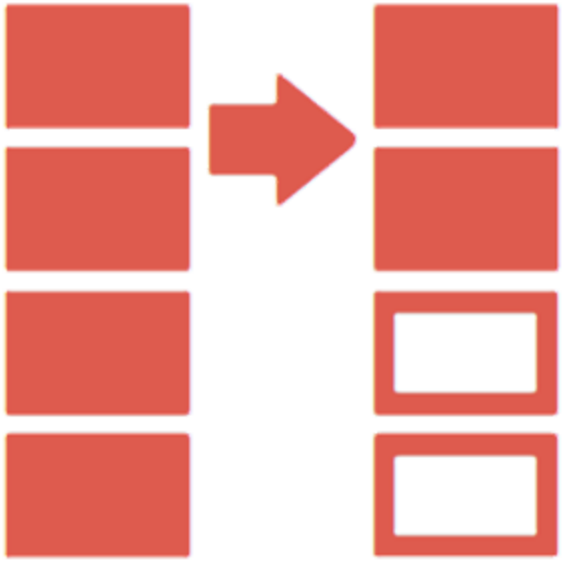
Access to the ADVANCED DATA LOAD endpoint requires that you first be authenticated within the platform. Within Messaging, authentication is handled by OAuth 2.0. To authenticate with OAuth 2.0, you must first obtain a "Consumer Key" and a "Consumer Secret." Both of

these values are managed at the user level, and can be obtained from within the Messaging application.

Next, you'll use your Consumer Key and Consumer Secret to request a "token." A token is a text string that, when provided in a request message, will allow the user access to the requested service. Tokens are valid only for a certain period of time.

For more details on how to authenticate your API request, please see the *Cheetah Messaging: API How-to Guide*.

2 Load Data



Overview

This section describes how to load data via a POST request to the ADVANCED DATA LOAD endpoint.

Payload

The ADVANCED DATA LOAD request payload is composed of the following main objects:

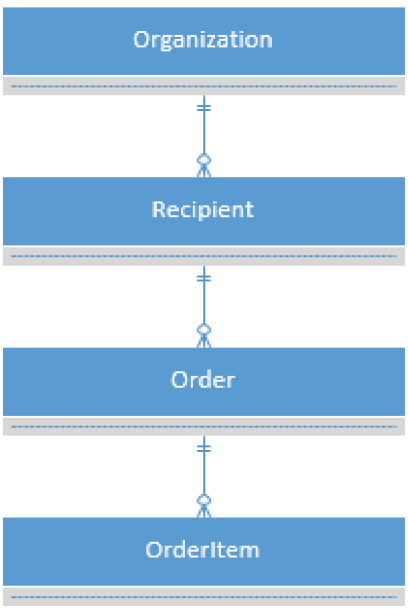
- **_data**: This required object defines the table relationships for the payload, and contains the data for the relational insert.
- **_importOptions**: If used, this optional object defines the import rules for tables (and fields) present in the _data section. If this object is not defined for a given table, the data will be imported according to the default behavior.

These objects are described below in more detail.

_data

The **_data** object contains the data that you want to load to your database. To illustrate the proper construction of this object, we will use the schema defined in the diagram below.

This account has the following tables: Organization, Recipient, Order, and OrderItem. Recipient has a join up to Organization (multiple Recipient records can belong to a single Organization), Order has a join up to Recipient (each recipient can be related to multiple orders), and OrderItem has a join up to Order (each Order can have multiple OrderItem records).



The **_data** object is composed of a nested series of objects, each one representing a table.

When sending data to the ADVANCED DATA LOAD endpoint, the root node within **_data** must be the target table.

Within each table object in **_data**, you can use three types of fields:

- Table fields (like first name, last name, or email address on a Recipient table)
- "Upward" joins (for a join from the current table up to a parent, like joining Recipient up to Organization); please note that the term "upward" is relative to whichever table is your reference point.
- Child joins (joins from "child" tables up to the current table, like a join from Order up to Recipient).

In regard to joins in Messaging -- joins represent a relationship between two tables, and they tell us how those two tables are related. In Messaging, all joins are "many-to-one," meaning that one record in the first table can be linked to many records in the second table. However, no record in the second table can be related to more than one record in the first table via that same join. Further, it should be noted that joins in Messaging are created against, and owned by, the "many" table.

Using the table diagram above, we can see a join between Recipient and Organization. This join might be named "recipient_to_organization," and it was created on the Recipient table (the "many" table), and it points to the Organization table. This join tells us that many recipients can be members of the same organization, but no recipient can be a member of two organizations (at least, not using this same join).

In the figure below, you can see the structure for the **_data** section. First, notice that the root node within **_data** is named "recipient," and that it's a collection of objects. At the current time, ADVANCED DATA LOAD doesn't support multiple records in this root node collection (i.e., inserting multiple records at once, for the target table).

Example:

```
{
                                                    "_data": {

"recipient": [
```

```
                                                                    {
"name_first": "John",
"name_last": "Smith",
"email": "john.smith@gmail.com",
"recipient_to_organization": {  //upward join
                              "organization_name": "Cheetah Digital",
                                   "organization_city": "Costa Mesa"
                                                                     },
"order.order_to_recipient":[  // A child join
                                                                    {

"order_no": "2834737",
"total": "129.99"
                                                                   },
                                                                    {

"order_no": "2834738",
"total": "59.99"
                                                                    }
                                                                     ]
```

```
                                                                    }
                                                                     ]
                                                                    }
    }
```

The above example illustrates all three types of fields:

- **Table Fields** -- 'name_first,' 'name_last,' and 'email' are all fields on the Recipient table.
- **Upward Joins** -- "recipient_to_organization" is an upward join from the Recipient table to the Organization table. The join was created on the Recipient table, and it points at the Organization table. These types of joins will always go from "many" on the current table to "one" on the referenced table.
- **Child Joins** -- Because a table can't have a join that points "down" to a child table, advanced data load uses dot notation to first reference the table that owns the join, and then specify which join on that table defines the specific relationship between these two tables. The dot notation convention looks like <table_name>.<join_name>, such as "order.order_to_recipient" in this example. This join tells us that each recipient can have many orders, but no order can belong to more than one recipient.

You can also see in this example that the recipient being targeted here has multiple orders being submitted. Using the join from Order to Recipient, we can pass through all of the related data needed for the message for that given Recipient.

Below is another example that shows a recipient record linked to the Order table two different times using two different joins.

First, through the same join we see above ("order_to_recipient") and second, through another join named "order_to_recipient_return_order." The second join still uses the Order table, but it will associate it to the Recipient through a different join to be called out in different areas of the campaign.

Example

```
    {
                                                                "_data": {

    "recipient": [
```

```
                                                                    {
"name_first": "John",
"name_last": "Smith",
"email": "john.smith@gmail.com",
"order.order_to_recipient":[  // a child join
                                                                    {
"order_no": "2834737",
"total": "129.99"
                                                                  },
                                                                    {
"order_no": "2834738",
"total": "59.99"
                                                                    }
                                                                     ],
"order.order_to_recipient_return_order": [ // a child join
                                                                    {
"order_no": "2834737",
```

```
    "total": "129.99"
                                                                    }
                                                                        ]
                                                                    }
                                                                        ]
                                                                    }
      }
```

The next example shows the **_data** section referencing multiple Orders that each have Items associated with them. This example shows how the relational structure can be used to reference multiple nested child elements.

Example:

```
    {
                                                            "_data": {
    "recipient": [
                                                                    {

    "name_first": "John",

    "name_last": "Smith",

    "email": "john.smith@gmail.com",

    "order.order_to_recipient":[ // A child join
                                                                    {


    "order_no": "2834739",


    "total": "59.99",


    "orderitem.orderitem_to_order":[ //A nested child join
```

```
                                                                    {

    "orderitem_no": "3473657",

    "itemname": "Product 1"

                                                                  },

                                                                    {

    "orderitem_no": "2345788",

    "itemname": "Product 2"

                                                                    }

                                                                     ]

                                                                  },

                                                                    {

    "order_no": "2834740",

    "total": "89.99",
```

```
    "orderitem.orderitem_to_order":[ //A nested child join
                                                                    {
    "orderitem_no": "8794355",
    "itemname": "Product 3"
                                                                    }
                                                                     ]
                                                                    }
                                                                     ]
                                                                    }
                                                                     ]
                                                                    }
    }
```

Above you can see that we now have two Orders, each with one or more Items. The "orderitem_to_order" join is used within each Order to associate the correct child records.

_importOptions

The optional **_importOptions** object can be used to specify import behavior on the ADVANCED DATA LOAD endpoint. This object is necessary only when you want to apply special rules when importing data.

When an inbound record matches on a primary (or unique) key to an existing record in the database, the inbound record becomes an "update" instead of an "insert." Within the platform, you can use a Data Map to specify how each field should react in that case. The ADVANCED DATA LOAD endpoint supports the same behavior, using the **_importOptions** object

instead of a Data Map. In normal operation, all such options are disabled, or "false," by default. This object, and each individual option within it, only needs to be included and specified if you need some behavior other than the default behavior.

The **_importOptions** object should be flat. You don't need to nest the related tables as they were nested in the **_data** object. Each table that needs special import options can appear only once per join used to reference it. For example, if you want to define import behaviors for each of the tables in the **_data** example used above, the **_importOptions** object would have sub-sections for each for the tables used.

In addition, if you define **_importOptions** for the schema described above in the **_data** section, you can declare separate import behaviors for each reference to the Order table. The tables in this case must be differentiated with a declaration of the join in the table reference. Ultimately, for the sample schema above, you could have up to three objects in the **_importOptions** object.

Example:

```
_"importOptions":
                                                                    [
                                                                     {
                                                    "_table": "recipient",
                                                "_doNotUpdateExisting": false,
                                                                     },
                                                                     {
                                                          "_table": "order",
                                                "_join": "order_to_recipient",
                                                "_doNotUpdateExisting": false,
                                                                     },
                                                                     {
                                                          "_table": "order",
                                    "_join": "order_to_recipient_return_order",
                                                "_doNotUpdateExisting": false,
                                                                     }
                                                                    ]
```

The parameters in the **_importOptions** object are described below in more detail.

_table

This string parameter is optional.

The **_table** parameter represents the name of the table for which you're defining import behaviors.

_join

This string parameter is optional.

If you're defining import behaviors for a table that's joined to the main target table, you must use the **_join** parameter to indicate the name of the table join.

_doNotUpdateExisting

This Boolean parameter is optional.

Applied at the table level, the **_doNotUpdateExisting** flag dictates whether the submitted information should be imported if a record with a matching unique key has been found in the database. Setting this to "true" means that only new records will be created, and existing records will not be updated. If you don't provide this parameter, the system defaults the value to "false."

- "True" = Create new records only
- "False" = Update/Create records (default)

_fieldOptions

The **_fieldOptions** object is used to define field-level import behaviors. This object should be nested beneath the **_importOptions**, for the desired table. For example:

```
"_importOptions":
                                                                    [
                                                                     {
                                                    "_table": "recipient",
                                                "_doNotUpdateExisting": false,
                                                    "_fieldOptions": [

                                                                    {
```

```
"_applyToFields": [

"name_first",

"name_last"

                                                                    ],

"_insertNull": true,

"_ignoreCase": true

                                                                    },

                                                                     {

"_applyToFields": [

"create_date"

                                                                    ],

"_preserveData": true

                                                                     }

                                                                      ]

                                                                     }
                                                                      ]
```

_applyToFields

This array is optional.

The **_applyToFields** array contains the **Column Name** of the field (or fields) for which you are defining import behavior.

_preserveData

This Boolean parameter is optional.

Setting the **_preserveData** parameter to "true" will preserve existing data if this is an update to an existing record. For example, let's say you have a "create_date" field in your request payload. You don't want to overwrite the existing value in this field in your database, so you would set **_preserveData** to "true" for this field. If you don't provide this parameter, the system defaults the value to "false."

- "True" -- Do not overwrite existing data
- "False" -- Overwrite existing data

_insertNull

This Boolean parameter is optional.

Setting the **_insertNull** parameter to "true" will overwrite data in the database as NULL if the supplied value is empty. Using this option, you can "blank out," or delete, existing data for a record that may no longer be valid. If you don't provide this parameter, the system defaults the value to "false."

- "True" -- Overwrite existing data as NULL if parameter value is empty.
- "False" -- Don't overwrite existing data if parameter value is empty.

_caseSensitive

This Boolean parameter is optional.

The **_caseSensitive** parameter is valid only for Unique Identifier and Primary Key fields. Setting this parameter to "true" will instruct the system to perform case-sensitive comparisons to the value in this field. If you don't provide this parameter, the system defaults the value to "false."

- "True" -- Perform case-sensitive comparisons to this field.

- "False" -- Perform case-insensitive comparisons to this field.

_ignoreBanList

This Boolean parameter is optional.

The **_ignoreBanList** parameter is valid only for fields with a Data Type of "Email." Setting this value to "true" will instruct the system not to check this email field against the platform's Global and Custom Email Ban lists. If you don't provide this parameter, the system defaults the value to "false."

- "True" -- Skip the Ban List validation for this email field.
- "False" -- Run the Ban List validation for this email field.

_preserveOptOut

This Boolean parameter is optional.

The **_preseveOptOut** parameter is valid only for fields with a Data Type of "Preference." When set to "true," this parameter forbids the new data in the API message from overwriting a current opt-out preference status. That is, once a recipient has opted-out, he or she can’t re-opt-in again if this flag is set to "true." If you don't provide this parameter, the system defaults the value to "false."

- "True" -- Don't use the value in this field to re-opt-in a recipient who previously opted-out.
- "False" -- Allow the system re-opt-in a recipient who previously opted-out.

_appendMultiValue

This Boolean parameter is optional.

The **_appendMultiValue** parameter is valid only for multi-value fields. When set to "true," this parameter instructs the system to append new data to this field, rather than overwriting it.

- "True" -- Append values to this field.
- "False" -- Overwrite value in this field.

Payload Validation

Messaging will validate the schema and data types of the submission. Any problems with the payload will result in an appropriate error response message; see the **Payload Errors** section for more details on these messages.

The message is checked to make sure all tables exist, are appropriately nested, and are joined correctly. In addition, the platform will check that nested child records that require primary keys have those records in their parent chain.

Further, all submitted data will be checked to confirm that it matches the data type specified in the platform for the given field.

3 Response

This section describes the possible response messages sent back from the ADVANCED DATA LOAD endpoint .

Success

A successful response to a POST message will generate a response code of "200," followed by the entire request message payload.

Authorization Errors

The following table lists the possible error messages that might be returned if the authorization step fails.

Error Code	Error Message	Possible Solution
401	Authorization has been denied for this request.	Get a token from the Token endpoint and pass it to the ADVANCED DATA LOAD endpoint in an Authorization header, with the value "Bearer: {token}".
403	Forbidden to access /data on this server. Please use correct URL.	Find out the correct URL and use that in the request.

Payload Errors

The following table lists the possible error messages that might be returned if the payload validation step fails.

Error Code	Error Message	Possible Solution
400	Field {0} does not exist for table {1}.	Field passed in payload doesn't exist in the table. Change field in payload, or add field to table.
400	Table {0} does not exist.	Choose correct table, or create table.
400	You must not send a blank submission.	Add data to submission (might be caused by incorrect JSON).
400	Join {0} is not a valid join. It does not Join table {1} to table {2}.	Join exists but doesn't join tables used in payload; Use correct join, or correct tables.
400	Join {0} is not a valid join. It does not exist.	Use correct join, or correct tables.
400	Join {0} is invalid. Table {1} does not exist.	Invalid join used. Create valid join, or add missing table.
400	You must include the 'Join' field with the value of the join name.	Join added incorrectly to payload.
400	All fields that make up the Alternate Key sequence key of table '{0}' are not present. You must include all Alternate Key sequence fields: '{1}'	Not all fields that are needed to create the Alternate Key are passed in the message. Add missing Alternate Key fields to the payload.
400	All fields that make up the Alternate Key sequence key of table '{0}' do not have values. You must provide values for all Alternate Key sequence fields: '{1}'	Some (or all) of the fields for the Alternate Key have a null or empty value. Add a valid value for all Alternate Key fields in the payload.
400	You cannot set the Null Flag Update Option on a Primary Key or Alternate Key Field.	Primary Key and Alternate Key fields need to be part of the update option.
400	Advanced Data Load not enabled. Please contact administrator.	Contact an administrator to confirm that you have access; ensure that the correct username is being used when getting an API token.
400	The "apiData" parameter is null. This can be caused by an improperly-formatted payload.	Confirm that the payload is not blank. Confirm that your payload has properly-formatted and complete JSON or XML, including delimiters. Also confirm that the Content-Type header you are passing matches the payload type; for JSON, it should be "application/json"; for XML, it should be "application/xml".

Error Code	Error Message	Possible Solution
400	Your content is not valid. Please check {key}	Confirm that the payload does not contain multiple entries for the specified key within one JSON object. This message might also appear if the payload is missing a required element such as "_data".
400	Cannot have an array at parent level join : {0}	What is actually expected is an array, but an object was found.
400	Message cannot be sent. The given email address was rejected by Email Ban Mask.	It's possible that the ban masks in the account are wrong -- that can be checked in the UI -- but it's more likely that the ban masks are correct and that you are trying to send to an email address that is on a banned list. There is nothing wrong with the payload, but the request can't succeed with the specified email address.

Other Errors

The following table lists the other error messages that might be returned in response to your request message.

Error Code	Error Message	Possible Solution
500	The 'ObjectContent`1' type failed to serialize the response body for content type 'application/xml; charset=utf-8'.	The Accept header is set to return XML, but the response data could not be serialized to XML. Setting the Accept header to "application/json" could be the solution.
500	The top level data object must have one property the name of the root table (e.g. \"recipient\") and value - an array with one element representing a single row from that table.	Confirm that the "_data" object in the payload contains an array property for the table, and an element for the row to insert.
500	PkId is null for targeting entity-{0}	It could be that fixing an error in the format of the value of the unique identifier property would prevent this, but it's more likely that you will just need to contact an administrator.

Error Code	Error Message	Possible Solution
500	Invalid MailAddress parts: name={0}, address={1}	The most likely cause of this error is that the specified email address for the message has invalid characters -- particularly double-byte characters.

4 Sample Messages

Overview

This section provides an example of an ADVANCED DATA LOAD request message.

Request Message #1

Below is a sample of an ADVANCED DATA LOAD request message in JSON format. This message uses a simple _data structure that references only one table.

```
{
                                                                "_data":{
"email_brand":[
                                                                        {

"email":"john.smith@cheetahdigital.com",

"brand_cd":"UO",

"uo_us_pref":"100"
                                                                        }
                                                                  ]
                                                                },
                                                     "_importOptions":[
                                                                  {
                                                "_table":"email_brand",
                                                      "_fieldOptions":[
                                                                        {

                                                     "_applyToFields":[
```

```
"ems_orig_date",

"orig_data_src_id",

"orig_data_src_name"
                                                                    ],
                                                          "_preserveData":true
                                                                    }
                                                                   ]
                                                                    }
                                                                   ]
}
```

Request Message #2

Below is a sample of an ADVANCED DATA LOAD request message in JSON format. This message uses a more complex **_data** structure with joins to several additional tables.

```
{
                                                                "_data":{
"recipient":[
                                                                    {

"name_first": "John",

"name_last": "Smith",

"email": "john.smith@example.com",

"recipient_to_organization": {

                                "oraganization_name": "Cheetah Digital",
```

```
                                    "organization_city": "Costa Mesa"
                                                                    },
"order.order_to_recipient":[
                                                                    {
"order_no": "2834737",
"total": "129.99",
"email": "john.smith@example.com",
"order_items.order_items_to_order":[
                                                                    {
"item_name": "Misc Clothing",
"price": "29.99",
"size": "Medium"
                                                                   },
                                                                    {
```

```
"item_name": "Nice Shirt",

"price": "79.98",

"size": "Small"

                                                                    }

                                                                     ]

                                                                    }

                                                                     ],

"order.replacement_order_to_recipient":[

                                                                    {

"order_no":"2837473",

"total":"0.00",

"trackingNumbers":"78542",

"order_items.order_items_to_order":[

                                                                    {
```

```
"item_name": "Nice Shirt",

"price": "79.98",

"size": "Medium"

                                                                    }

                                                                     ]

                                                                    }

                                                                     ]

                                                                    }
                                                                     ]
                                                                   },
                                                        "_importOptions":[
                                                                     {

                                                               "_table": "recipient",

                                                      "_doNotUpdateExisting": false,

                                                                "_fieldOptions":[

                                                                     {

                                                              "_applyToFields":[

"name_first",

"name_last"

                                                                   ],
```

```
                                                    "_insertNull": true
                                                                    },
                                                                    {
                                                     "_applyToFields":[
"email"
                                                                    ],
                                                  "_caseSensitive": true
                                                                    }
                                                                   ]
                                                                    },
                                                                    {
                                                      "_table":"order",
                                                "_joinName":"order_to_recipient",
                                                    "_doNotUpdateExisting": true,
                                                                "_fieldOptions":[
                                                                    {
                                                     "_applyToFields":[
"order_no",
"total"
                                                                    ],
                                                    "_insertNull": true
                                                                    }
```

```
                                                                    ]
                                                                     },
                                                                     {

                                                        "_table":"order",

                                        "_joinName":"replacement_order_to_recipient",

                                                    "_doNotUpdateExisting": true,

                                                              "_fieldOptions":[]
                                                                     }
                                                                    ]
}
```

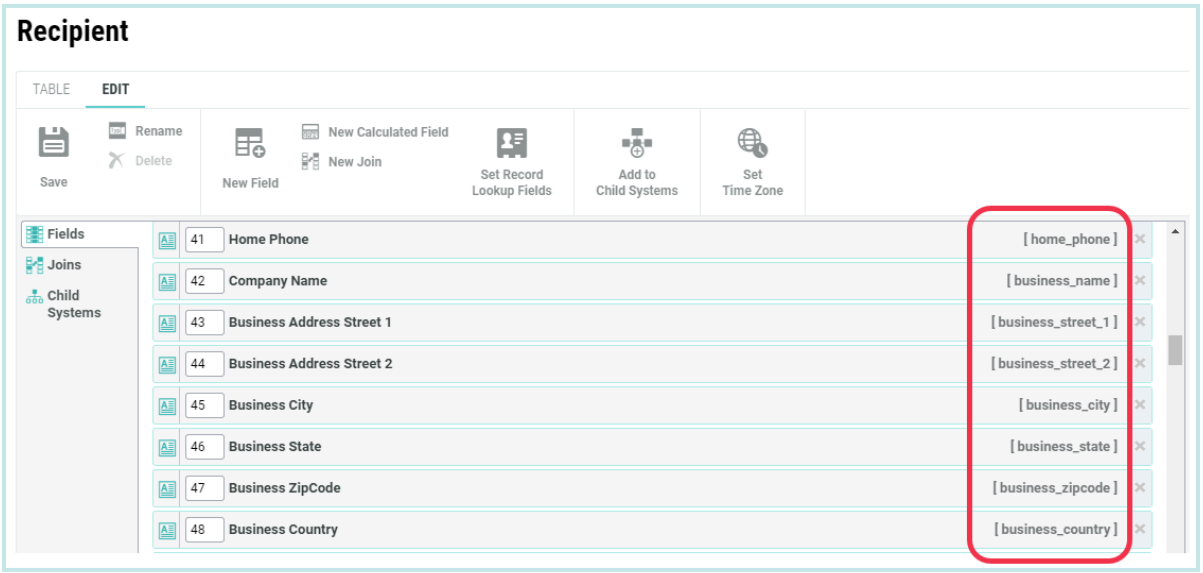
5 Appendix A -- Identifiers

Column Names

For the field names in the **_applyToFields** array, you must use the system "Column Name," and not the viewer-friendly "Display Name." The Column Names for fields can be found on the Tables screen within the Messaging application, or by using the TABLE endpoint.

To look up the Column Name within the Messaging application:

1. From the System Tray, select *Data Management* > *Structures* > *Tables*. The system displays a list of all the tables in your account.
2. Select the desired table. The Table Details screen is displayed.
3. Within the list of fields in this table, the Column Name is displayed on the far-right of the screen.



To retrieve the Column Name for a field using the TABLE endpoint:

1. Submit a GET request to the TABLE API endpoint. The simplest method is to use the version of the TABLE endpoint that allows you to retrieve table information based on the table's name. For example:

```
https://api.eccmp.com/services2/api/Table?tableName=recipient
```

2. Within the API response message, the system lists every field in this table. As part of that field definition, the response includes the Column Name (referred to as the **columnName**).

Sample Response:

```
{
 "viewId": 1002,
 "entityId": 100,
 "displayName": "First Name",
 "propId": 1030,
 "columnName": "name_first"
}
```