API Technical Guide: Looping Block

Cheetah Messaging

Table of Contents

1 Introduction

Purpose

The purpose of this document is to provide an overview of the LOOPING BLOCK API endpoint within the Cheetah Messaging platform. This document discusses the intended use of the LOOPING BLOCK endpoint, and provides technical details for how to implement the endpoint.

Overview

The LOOPING BLOCK endpoint is used to create, view, edit, and delete Looping Blocks.

A Looping Block is a special type of Content Block that allows you to create sophisticated messages that contain repeating content, complex personalization, calculations, and formatting.

Looping Blocks can be used in the following types of scenarios:

- Building dynamic content consisting of repeating values. For example, you could create an order confirmation email message that's personalized with all of the items that the customer purchased.
- Building dynamic content containing data gathered from tables other than the Campaign source table, regardless of the relationship between those tables. For example, you could create an order confirmation email message that contains the order number and date (stored on the Order table), the customer's name and address (stored on the Customer table), and product descriptions for each item purchased (stored on the Product table).

- Transforming or reformatting data. For example, you could reformat a date field in your Campaign content from the way it's stored in the database ("2015-15-05 02:31:11") into a more reader-friendly format ("May 5, 2015").
- Performing calculations on data. For example, you could create an email Campaign that references the consumer's Last Purchase Date by identifying all purchase records associated with a recipient, calculating which of those purchases was the "last" one, and then displaying that date within the email content.
- Ranking and ordering data. For example, you could populate a Campaign message with a list of recommended products, sorted by cost.

Creating an effective Looping Block relies on a strong understanding of your database tables and how they're joined together. A "join" refers to the relationship between tables. Joins have a direction:

- **One-to-many**: This join describes a table where one record can map to multiple records in a joined table.
- **Many-to-one**: This join describes a table where many records map back to a single record in a joined table.

Looping Blocks allow you to create complex, sophisticated personalization in your message content, drawing values from tables in either the one-to-many direction, or in the many-to-one direction. Looping Blocks offer more functionality than Personalization Fields, which are limited to accessing fields in joined tables only in the many-to-one direction.

A Looping Block consists of two components:

- **XML document**: XML is a mark-up language used to describe data -- including its attributes, and its relationship to other data -- in a manner that's readable by both machines and humans. Within Messaging, the platform creates the XML document for you once you've defined what data you want to use in your message content. You never work directly with the XML document.
- **XSL style sheet coded in the XLST language**: XSLT is a style sheet language for XML documents that can look through the tables of a database, extract the

desired values, and output them in a particular format or ranking based on the user's requirements. In short, XSLT transforms and presents the data that XML has defined.

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

The URLs for this endpoint are:

- **North America:**
- https://api.eccmp.com/services2/api/LoopingBlock
- https://api.eccmp.com/services2/api/LoopingBlock/{id}
- **Europe:**
- https://api.ccmp.eu/services2/api/LoopingBlock
- https://api.ccmp.eu/services2/api/LoopingBlock/{id}
- **Japan:**
- https://api.marketingsuite.jp/services2/api/LoopingBlock
- https://api.marketingsuite.jp/services2/api/LoopingBlock/{id}

Methods

The LOOPING BLOCK endpoint supports the following HTTP methods:

- **POST**: Create a new Looping Block.
- **GET**: Retrieve information about a specified Looping Block.
- **PUT**: Submit modifications to an existing Looping Block.
- **DELETE**: Delete a specified Looping Block.

Authentication

Access to the LOOPING BLOCK 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 *Messaging: API How-to Guide*.

2 Create a Looping Block

Overview

This section describes how to create a new Looping Block via a POST request to the LOOPING BLOCK endpoint.

A Looping Block includes two main parts: the data source and the content.

The data source represents the field (or fields) needed to personalize the content. These fields can reside on the Looping Block's primary source table, as well as on joined tables. After you define the data source, the system automatically generates sample XSLT code. You can modify this sample code as needed, and use it to populate the content. Or you can enter your own XSLT code, if desired.

The content represents the message, usually consisting of one or more "format versions." Most channels in Messaging allow you to define multiple format versions of the message content in order to accommodate the different devices and applications used by recipients to view your message. A common example is an Email Campaign that includes both an HTML version and a Plain Text version. If a recipient has an email application that's not configured to view HTML messages, he or she can still see the Plain Text version of your message.

If you have your own XSLT code that you want to use in a Looping Block, then you can simply provide this code within the POST message used to create the brand new Looping Block.

If you intend to use the system-generated XSLT code, then creating a new Looping Block via the LOOPING BLOCK endpoint requires a multi-step process. At the time you're sending in a brand new POST request, you don't have the sample code, because the system hasn't generated it yet. The basic steps in this scenario are as follows:

1. Use a POST request to create a new "empty" Looping Block with the desired data source and format versions. The content for each format version will consist of only a valid, generic XSLT document.
2. Upon receiving the POST request, the system creates the new Looping Block, and automatically generates the sample XSLT code based on the defined data source. The response message to the POST request will contain the system-generated XSLT code.
3. Make any necessary changes to the system-generated XSLT code.
4. Create a PUT request message, and insert the appropriate XSLT code as the content for each format version. Submit the PUT request.

Parameters

The options and parameters described in this section explain how to create a new Looping Block.

The basic structure of a POST request to the LOOPING BLOCK endpoint is as follows:

```
"entity_id"
"view_id"
"cust_id"
"type_id"

"ContBodies"

"contPropNodes"

"Obj"
```

entity_id

This integer parameter is required.

The **entity_id** parameter represents the Entity ID of the Looping Block's source table.

Example:

```
"entity_id": 100
```

view_id

This integer parameter is required.

The **view_id** parameter represents the Object Reference ID of the Looping Block's source table. The **view_id** must refer to the same table as the **entity_id** parameter.

Example:

```
"view_id": 1002
```

cust_id

This integer parameter is required.

The **cust_id** parameter represents the Customer ID of your Messaging account. The Customer ID is a unique, system-generated identifier for every Messaging client account. This value isn't displayed anywhere within the Messaging application, so you must retrieve it by means of an API request (several different endpoints will return the Customer ID as part of the response message), or speak to your Client Services Representative, who can provide you with this value.

Example:

```
"cust_id": 103
```

type_id

This string parameter is required.

For a Looping Block, the value of this parameter must be "LOOPING_BLOCK."

Example:

```
"type_id": "LOOPING_BLOCK"
```

contBodies

This object specifies the format versions for your Looping Block, and the message content for each format version. Within this same object, you may have one or more format versions (HTML, Plain Text, etc.), optionally with different content for each version.

Example:

```
"contBodies":
 [
  {
    "type_id": "HTML",
    "usage_mask": "EMAIL",
    "body": "XSLT code goes here"
  },
  {
    "type_id": "TEXT",
    "usage_mask": "EMAIL",
    "body": "XLST code goes here"
  }
 ]
```

The parameters in this object are described below in more detail.

type_id

This string parameter is required.

The **type_id** parameter is used in combination with the **usage_mask** parameter (described below) to define the format version of the content.

The possible values for this parameter are:

- HTML
- TEXT
- XML

Example:

```
"type_id": "HTML"
```

usage_mask

This string parameter is required.

The **usage_mask** parameter is used in combination with the **type_id** described above to define the format version of the content.

Most of the format versions in Messaging are actually groups containing multiple sub-options. For example, the "HTML" format version contains sub-options for: Email Message, Web, Viral, Share to Social, Mobile Web, and iPhone. By default, all of these

sub-options are contained within the parent HTML version, meaning that the same HTML content will be used for all of those different contexts.

The **usage_mask** parameter allows you to pull one or more of those sub-options out of the parent version, and create a new, separate format version. This process is referred to as "promoting" a format option into a new group. For example, if you need the "Mobile Web" version of your HTML content to be different than the "Email Message" HTML content, you could promote "Mobile Web" to its own format version, and then provide the content unique to "Mobile Web" recipients.

This parameter can optionally contain multiple values, separated by commas.

Example:

```
"usage_mask": "EMAIL, WEB_MOBILE"
```

The valid values and combinations with **type_id**, along with the name used within the application's user interface, are listed below.

Some of the **usage_mask** values described below are designed as "bundles" of commonly used format versions.

type_id	usage_mask	User Interface Name
HTML	EMAIL	HTML > Email Message
HTML	WEB_IPHONE	HTML > iPhone
HTML	WEB_SOCIAL	HTML > Share to Social
HTML	WEB	HTML > Web
HTML	VIRAL	HTML > Viral
HTML	WEB_MOBILE	HTML > Mobile Web
HTML	ALL_EMAIL_STYLE_USAGE_MASK	Encompasses the following HTML options: Email Message, iPhone, Share to Social, Web, Viral, and Mobile Web.
HTML	ALL_WEB_STYLE_USAGE_MASK	Encompasses the following HTML options: Web, Share to Social, Mobile Web, and iPhone.
HTML	REPORT_SOCIAL_MASK	Encompasses the following HTML options: Share to Social, Viral.

type_id	usage_mask	User Interface Name
HTML	REPORT_MSG_MASK	Encompasses the following HTML options: Email Message, Web, Mobile Web, and iPhone.
HTML	NONE	Creates a "blank" Looping Block with no format versions (Please note that the user interface doesn't allow you to create a Looping Block with no format versions).
HTML	ALL	All possible usage masks.
TEXT	EMAIL	Plain Text > Email Message
TEXT	WEB_SOCIAL	Plain Text > Social Text
TEXT	WEB	Plain Text > Web
TEXT	1024	Plain Text > Push Notification
TEXT	VIRAL	Plain Text > Viral
TEXT	ALL_EMAIL_STYLE_USAGE_MASK	Encompasses the following Plain Text options: Email Message, Web, Viral, and Social Text.
TEXT	ALL_WEB_STYLE_USAGE_MASK	Encompasses the following Plain Text options: Web, Social Text.
TEXT	REPORT_SOCIAL_MASK	Encompasses the following Plain Text options: Viral, Social Text.
TEXT	SMS	Mobile Text
TEXT	HTTP_REQUEST	Data > HTTP
TEXT	DATA_FILE	Data > Flat File
TEXT	NONE	Creates a "blank" Looping Block with no format versions (Please note that the user interface doesn't allow you to create a Looping Block with no format versions).
TEXT	ALL	All possible usage masks.
XML	WEB	XML

body

This string parameter is required.

The **body** parameter is used to provide the content, in the form of an XSLT document, for this format version. If you already have this code, you can provide it in this parameter.

If you intend to use the system-generated XLST code, then you can simply provide a valid, generic XSLT document in this parameter, as you'll later use a PUT request to insert the system-generated XLST code into this Looping Block.

For example, the bare minimum required in the **body** parameter is the following:

```
"body": "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n<xsl:stylesheet
version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\r\n<xsl:output
method=\"text\" encoding=\"UTF-8\" indent=\"no\"
omit-xml-declaration=\"yes\" />\r\n<xsl:template
match=\"/\">\r\n</xsl:template>\r\n</xsl:stylesheet>"
```

contPropNodes

This object specifies the hierarchy for the Looping Block's data source. Within this same object, you may have one or more fields defined, and those fields can be on the Looping Block's source table, or on a joined table.

The following example describes a data source with fields across three different tables:

- Two fields on the Looping Block source table ("name_first" and "name_last").
- A join to the "order" table.
- Two fields on the "order" table ("orderid" and "itemid")
- A join from the "order" table to the "item" table.
- Two fields on the "item" table ("itemname" and "cost")

Example:

```
"contPropNodes":
  [
    {
    "prop_id": 1100,
    "comment": "name_first"
    },
    {
    "prop_id": 1120,
    "comment": "name_last"
    },
    {
```

```
"prop_id": 2406,
"tag": "order2recipient",
"array_tag": "order2recipients",
"contPropNodes":
  [
    {
    "prop_id": 2403,
    "comment": "orderid"
     },
     {
    "prop_id": 2404,
    "comment": "itemid"
     },
     {
     "prop_id": 2408,
     "tag": "itemtoorder",
     "array_tag": "itemtoorders",
     "filter_id": 34343,
     "contPropNodes":
       [
         {
         "prop_id": 2398,
         "comment": "itemname"
         },
         {
         "prop_id": 2399,
         "comment": "cost"
         }
       ]
     }
   ]
 }
]
```

The parameters in this object are described below in more detail.

prop_id

This integer parameter is required.

The **prop_id** parameter represents either a Field ID (or a Join ID). These IDs are system-generated identifiers for every field and join in your database.

comment

This string parameter is optional.

The **comment** parameter represents the XML tag for the field, and if provided, will be used in the system-generated XSLT code.

tag

This string parameter is optional.

The **tag** parameter is used when defining a join to another table. This value represents the XML "item tag" for this join, and if provided, will be used in the system-generated XSLT code.

In the following sample XSLT code, the **tag** value was provided as "order."

```
<h2>OrdertoRecipient (jk_order)</h2>
<xsl:for-each select="/Msg/Props/orders/order">
<xsl:value-of select="Prop[@prop_name = 'order_date']/@val" /><br />
  <xsl:value-of select="Prop[@prop_name = 'order_id']/@val" /><br />
```

array_tag

This string parameter is optional.

The **array_tag** parameter is used when defining a join to another table. This value represents the XML "collection tag" for this join, and if provided, will be used in the system-generated XSLT code.

In the following sample XSLT code, the **array_tag** value was provided as "orders."

```
<h2>OrdertoRecipient (jk_order)</h2>
<xsl:for-each select="/Msg/Props/orders/order">
<xsl:value-of select="Prop[@prop_name = 'order_date']/@val" /><br />
  <xsl:value-of select="Prop[@prop_name = 'order_id']/@val" /><br />
```

filter_id

This integer parameter is optional.

The **filter_id** parameter is used when defining a join to another table, and you want to narrow down the records that the Looping Block will loop through, to only the records selected by the Filter.

The **filter_id** parameter represents the **Object Reference ID** of the Filter.

obj

This object contains the name and location of the new Looping Block.

Example:

```
"obj":
  {
  "display_name": "Sample Looping Block",
  "parent_obj_id": 37249
  }
```

The parameters in this object are described below in more detail.

display_name

This string parameter is required.

This parameter contains the name of the Looping Block. This name must be unique within the selected folder location.

Example:

```
"display_name": "Sample Looping Block"
```

parent_obj_id

This integer parameter is required.

The **parent_obj_id** parameter represents the **Folder ID** of the folder where you want to save the new Looping Block.

Example:

```
"parent_obj_id": 37249
```

3 Edit a Looping Block

Overview

This section describes the options for viewing, modifying, and deleting Looping Blocks via the LOOPING BLOCK endpoint.

Retrieve a Looping Block

The GET method is used to retrieve all of the information about a specified Looping Block, including the system-generated XLST code. This sample code is contained within a parameter named **contBodySampleCode**.

When submitting a GET request to the LOOPING BLOCK endpoint, the request message must include the Looping Block's **Object Reference ID** as a query type parameter within the URL.

For example:

```
https://api.eccmp.com/services2/api/LoopingBlock?id=3456
```

Or, optionally, you can include the Object Reference ID as a path type parameter within the URL.

For example:

```
https://api.eccmp.com/services2/api/LoopingBlock/3456
```

Delete a Looping Block

The DELETE method is used to delete a specified Looping Block.

When submitting a DELETE request to the LOOPING BLOCK endpoint, the request message must include the Looping Block's **Object Reference ID** as a query type parameter within the URL.

For example:

```
https://api.eccmp.com/services2/api/LoopingBlock?id=3456
```

Or, optionally, you can include the Object Reference ID as a path type parameter within the URL.

```
https://api.eccmp.com/services2/api/LoopingBlock/3456
```

Edit a Looping Block

The PUT method allows you to submit modifications to an existing Looping Block. Using this method, you can change the Looping Block name, modify the XLST code, and add or remove format versions. The parameters for the PUT method are the same as described in the **Create a Looping Block** section.

To remove a format version from an existing Looping Block, simply omit it from the request message; any existing format versions that aren't referenced in the PUT request message will be removed from the Looping Block.

When submitting a PUT request to the LOOPING BLOCK endpoint, the request message must include the Looping Block's **Object Reference ID** as a query type parameter within the URL.

For example:

```
https://api.eccmp.com/services2/api/LoopingBlock?id=3456
```

Or, optionally, you can include the Object Reference ID as a path type parameter within the URL.

```
https://api.eccmp.com/services2/api/LoopingBlock/3456
```

4 Response

This section describes the possible response messages sent back from the LOOPING BLOCK endpoint.

Success

A successful response to a POST message to create a new Looping Block will generate a response code of "200," followed by the details of the new Looping Block (including the system-generated XSLT code) contained within the body of the response message.

A successful response to a GET message to retrieve a Looping Block will generate a response code of "200," followed by the details of the specified Looping Block (including the system-generated XSLT code) contained within the body of the response message.

A successful response to a PUT message to update a Looping Block will generate a response code of "200," followed by the details of the modified Looping Block contained within the body of the response message.

A successful response to a DELETE message will generate a response code of "204;" the body of the response message will be empty.

Errors

If Messaging encounters a problem with a LOOPING BLOCK request message, the platform will send an "error" message with details of the problem. Below is a list of error codes and their descriptions.

Response Code	Error message	Description
400	"Each content body must have at least one Use"	Required parameter **usage_mask** is missing or contains an invalid value.
400	" Looping Block with that name <display_name> already exists in this folder."	A Looping Block with this same name already exists within the designated folder; the **display_name** value must be unique within a folder.
400	"View Id is missing for Looping Block Content."	Required parameter **view_id** is missing.
400	"Entity Id is missing for Looping Block Content."	Required parameter **entity_id** is missing.
400	"Entity Id and View Id are not associated to the same entity."	Mismatch between the **view_id** and **entity_id** values. Both parameters must refer to the same table.
400	"Active Customer with the cust_id <cust_id> does not exist in the cache."	Required parameter **cust_id** is missing or contains an invalid value.
400	"Invalid type_id for content."	Parameter **type_id** must contain "LOOPING_BLOCK."
400	"One or more of the content bodies have empty body."	Required parameter **body** is missing or blank.
400	"Xslt provided is not valid:XSLT compile error."	Parameter **body** does not contain valid XSLT code.

5 Sample Messages

This section contains a sample request message for the LOOPING BLOCK endpoint.

Sample Request Message #1

This sample POST request message creates a new Looping Block with two format versions. In this scenario, the user intends to use the system-generated XSLT code, and so she is submitting valid, generic XSLT code in the **body** parameter for each format version.

JSON Payload


```
{
  "cust_id": 103,
  "entity_id": 100,
  "view_id": 1002,
  "type_id": "LOOPING_BLOCK",
  "contBodies":
    [
      {
      "type_id": "HTML",
      "usage_mask": "ALL_EMAIL_STYLE_USAGE_MASK",
      "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\r\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\r\n<xsl:output
method=\"text\" encoding=\"UTF-8\" indent=\"no\"
omit-xml-declaration=\"yes\" />\r\n<xsl:template
match=\"/\">\r\n</xsl:template>\r\n</xsl:stylesheet>"
      },
      {
      "type_id": "TEXT",
      "usage_mask": "EMAIL",
       "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\r\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\r\n<xsl:output
method=\"text\" encoding=\"UTF-8\" indent=\"no\"
omit-xml-declaration=\"yes\" />\r\n<xsl:template
match=\"/\">\r\n</xsl:template>\r\n</xsl:stylesheet>"
      }
    ],
```

```
  "contPropNodes":
    [
      {
      "prop_id": 1100,
      "comment": "first_name"
      },
      {
      "prop_id": 1120,
      "comment": "last_name"
      }
   ],
  "obj":
    {
    "display_name": "API Looping Block",
    "parent_obj_id": 13552
    }
}
```

Sample Response Message #1

This sample GET response message is used to retrieve information about the Looping Block created above. The response includes the system-generated sample XLST code (highlighted below in green).

JSON Payload

```
{
  "cont_id": 1566,
  "cont_name": "API Looping Block",
  "cust_id": 103,
  "entity_id": 100,
  "type_id": "LOOPING_BLOCK",
  "view_id": 1002,
  "contBodySampleCode": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\n<xsl:output
method=\"html\" encoding=\"UTF-8\" indent=\"yes\"
omit-xml-declaration=\"yes\" />\n<xsl:template
match=\"/\">\n<xsl:value-of select=\"/Msg/Props/Prop[@prop_name =
'name_last']/@val\" /><br />\n<xsl:value-of
select=\"/Msg/Props/Prop[@prop_name = 'name_first']/@val\" /><br
/>\n\n</xsl:template>\n</xsl:stylesheet>",
  "obj": {
    "obj_id": 13558,
    "display_name": "API Looping Block",
    "type_id": "LoopingBlock",
    "ref_id": 1566,
    "parent_obj_id": 13552,
    "eligibility_status_id": "READY"
```

```
  },
  "contBodies": [
    {
      "cont_id": 1566,
      "type_id": "TEXT",
      "usage_mask": "EMAIL",
      "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\r\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\r\n<xsl:output
method=\"text\" encoding=\"UTF-8\" indent=\"no\"
omit-xml-declaration=\"yes\" />\r\n<xsl:template
match=\"/\">\r\n</xsl:template>\r\n</xsl:stylesheet>"
    },
    {
      "cont_id": 1566,
      "type_id": "HTML",
      "usage_mask": "ALL_EMAIL_STYLE_USAGE_MASK",
      "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\r\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\r\n<xsl:output
method=\"text\" encoding=\"UTF-8\" indent=\"no\"
omit-xml-declaration=\"yes\" />\r\n<xsl:template
match=\"/\">\r\n</xsl:template>\r\n</xsl:stylesheet>"
    }
  ],
  "contPropNodes": [
    {
      "node_id": 624,
      "cont_id": 1566,
      "prop_id": 1120,
      "comment": "last_name"
    },
    {
      "node_id": 625,
      "cont_id": 1566,
      "prop_id": 1100,
      "comment": "first_name"
    }
  ]
}
```

Sample Request Message #2

This sample PUT request message updates the Looping Block created above, by inserting the XLST code into both of the defined format versions. Please note that if you provide the sample XLST code back to the platform in the **contBodySampleCode** parameter, the platform will ignore it.

JSON Payload


```
{
    "cont_id": 1566,
    "cont_name": "API Looping Block",
    "cust_id": 103,
    "entity_id": 100,
    "type_id": "LOOPING_BLOCK",
    "view_id": 1002,
    "contBodySampleCode": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\n<xsl:output
method=\"html\" encoding=\"UTF-8\" indent=\"yes\"
omit-xml-declaration=\"yes\" />\n<xsl:template
match=\"/\">\n<xsl:value-of select=\"/Msg/Props/Prop[@prop_name =
'name_last']/@val\" /><br />\n<xsl:value-of
select=\"/Msg/Props/Prop[@prop_name = 'name_first']/@val\" /><br
/>\n\n</xsl:template>\n</xsl:stylesheet>",
    "obj": {
        "obj_id": 13558,
        "display_name": "API Looping Block,
        "type_id": "LoopingBlock",
        "ref_id": 1566,
        "parent_obj_id": 13552,
        "eligibility_status_id": "READY"
    },
    "contBodies": [
        {
            "cont_id": 1566,
            "type_id": "TEXT",
            "usage_mask": "EMAIL",
            "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\n<xsl:output
method=\"html\" encoding=\"UTF-8\" indent=\"yes\"
omit-xml-declaration=\"yes\" />\n<xsl:template
match=\"/\">\n<xsl:value-of select=\"/Msg/Props/Prop[@prop_name =
'name_last']/@val\" /><br />\n<xsl:value-of
select=\"/Msg/Props/Prop[@prop_name = 'name_first']/@val\" /><br
/>\n\n</xsl:template>\n</xsl:stylesheet>"
        },
        {
            "cont_id": 1566,
            "type_id": "HTML",
            "usage_mask": "ALL_EMAIL_STYLE_USAGE_MASK",
            "body": "<?xml version=\"1.0\"
encoding=\"utf-8\"?>\n<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\n<xsl:output
method=\"html\" encoding=\"UTF-8\" indent=\"yes\"
omit-xml-declaration=\"yes\" />\n<xsl:template
match=\"/\">\n<xsl:value-of select=\"/Msg/Props/Prop[@prop_name =
'name_last']/@val\" /><br />\n<xsl:value-of
select=\"/Msg/Props/Prop[@prop_name = 'name_first']/@val\" /><br
/>\n\n</xsl:template>\n</xsl:stylesheet>"
```

```
            }
        ],
        "contPropNodes": [
            {
                "node_id": 624,
                "cont_id": 1566,
                "prop_id": 1120,
                "comment": "last_name"
            },
            {
                "node_id": 625,
                "cont_id": 1566,
                "prop_id": 1100,
                "comment": "first_name"
            }
        ]
    }
```

6 Appendix A -- Identifiers

Messaging uses several different types of IDs when referencing assets, such as tables, fields, folders, Filters, and so forth. This appendix describes these different types of IDs, and provides steps on how to look up the value of an ID.

Entity ID

The Entity ID is a unique, system-generated identifier for every table in your database. This value is not displayed within the application user interface anywhere, so to get the Entity ID for a table, you must retrieve it by means of the TABLE API endpoint.

To retrieve the Entity ID for a table:

1. Submit a request to the TABLE API endpoint. The simplest method is to use the version of the TABLE endpoint that allows you to retrieve 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 the field details, the response message provides the Entity ID for this table.

Sample Response:

```
{
  "viewId": 1002,
  "entityId": 100,
  "displayName": "create_date",
  "propId": 1030,
  "columnName": "create_date"
}
```

For more details on the TABLE endpoint, please see the Messaging Online Help system or the *Messaging -- Table API Technical Guide*.

Folder ID

The Folder ID is a unique, system-generated identifier for each folder and sub-folder in your system. This value is not displayed within the application user interface anywhere, so to get your Folder ID, you must retrieve it by means of the SEARCH API endpoint.

1. Submit a GET request to the SEARCH endpoint. The easiest method is to use the version that lets you search by object type -- use a type value of "Folder." For example:

```
https://api.eccmp.com/services2/api/Object?type=Folder
```

2. The response message provides a list of all the folders in your system. Find the desired folder in the response message.
3. As part of the API response message, the system provides the Folder ID, which is referred to as the "obj_id."

> **Note**
>
> If this Folder is a sub-folder, the "parent_obj_id" is the Folder ID of the parent folder.

Sample Response:

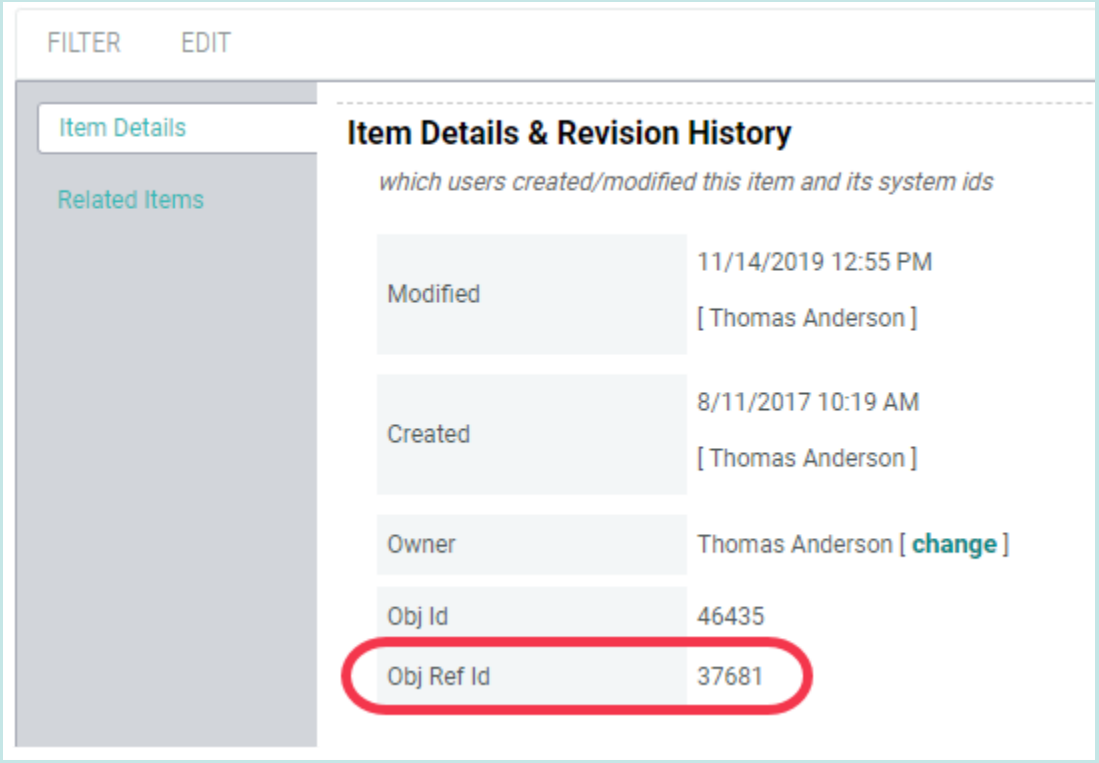
```
{
  "obj_id": 37465,
  "display_name": "Content Block Folder",
  "type_id": "Folder",
  "ref_id": 37465,
  "parent_obj_id": 22817,
  "eligibility_status_id": "READY"
}
```

Object Reference ID

The Object Reference ID is a system-generated identifier for every item and asset in your account.

For some asset types, the value for this identifier can be found within the Messaging application:

1. From the System Tray, navigate to desired screen for this asset type.
2. In the Tool Ribbon, click the first tab; the name of this tab corresponds to the asset type, such as "Filter" if you're on the Filter screen, for example.
3. The "Item Details" screen is displayed. The Object Reference ID is listed on this screen.



Optionally, for many asset types, you can use the SEARCH endpoint, and search for the desired asset:

1. Submit a GET request to the SEARCH API endpoint. The simplest method is to use the versions of the SEARCH endpoint that allow you to retrieve information based on either the asset's name or its type. For example, to retrieve information about all of your Filters:

```
https://api.eccmp.com/services2/api/Object?type=Filter
```

2. The response message provides a list of all the assets in your system that match the search criteria. Find the desired asset in the response message.
3. As part of the API response message, the system provides the Object Reference ID, which is referred to as the "ref_id." For example:

```
{
  "obj_id": 44737,
  "display_name": "Reward Members Filter",
  "type_id": "Filter",
  "ref_id": 40329,
  "parent_obj_id": 43269,
  "eligibility_status_id": "READY"
}
```

Field ID

The Field ID (or "Property ID") is a unique, system-generated identifier for every field in a table. This value is not displayed within the application user interface anywhere, so to get the Field ID for a field, you must retrieve it by means of the TABLE API endpoint.

To retrieve the Field ID for a field:

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 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 Field ID (referred to as the **propId**).

Sample Response:

```
{
 "viewId": 1002,
 "entityId": 100,
 "displayName": "create_date",
 "propId": 1030,
 "columnName": "create_date"
}
```