# Introduction



# Introduction

## API Introduction

Cisco Unified Communications Domain Manager introduces a completely new approach to Unified Communications management that emphasizes the API as the common point of entry to the system. All Cisco Unified Communications Domain Manager functionality is available through the API, including the Cisco Unified Communications Domain Manager GUI itself, which uses the same API. Because the GUI uses the same API that is available to service providers externally, developers can use trace tools (such as Chrome developer tools) to learn details on the API schema, how the API operates, how to create, update, and delete objects, etc. Any operation performed by the Cisco Unified Communications Domain Manager GUI can be traced and replicated by an external client.

The Cisco Unified Communications Domain Manager API is a secure, normalized, and integrated REST-based API. Secure in the sense that it supports HTTPS digest authentication. Normalized in the sense that all APIs follow the same overall schema, semantics, and operations; only the detailed attribute schemas and authorized operations differ with each object depending on the type of object and the user credentials used to access the API. Integrated in the sense that the Cisco Unified Communications Domain Manager platform allows access to the following HCS components with this normalized API:

- Cisco Unified Communications Domain Manager orchestrated workflows
- Cisco Hosted Collaboration Mediation Fulfillment database
- Cisco Unified Communications Manager cluster databases
- Cisco Unity Connection databases
- LDAP user data

The API uses a well defined JSON schema with consistent meta-data across all device types integrated by Cisco Unified Communications Domain Manager , regardless of the devices native API format. For example, the native Cisco Unified Communications Domain Manager API is SOAP based XML but this is converted to object-oriented REST-based JSON, consistent with all other APIs in the Cisco Unified Communications Domain Manager system.

There are many APIs in the HCS system, including the native APIs for the devices integrated into Cisco Unified Communications Domain Manager (Cisco Hosted Collaboration Mediation Fulfillment, Cisco Unified Communications Domain Manager, Cisco Unity Connection), APIs for other devices not integrated in Cisco Unified Communications Domain Manager . These APIs are still available and can still be used for operations that aren't supported by Cisco Unified Communications Domain Manager orchestrated workflows. For example, Cisco Hosted Collaboration Mediation Fulfillment supports Prime Collaboration Assurance (PCA) configuration via API, but Cisco Unified Communications Domain Manager 10.1 does not currently support configuring PCA directly. Therefore, you can use the native Cisco Hosted Collaboration Mediation Fulfillment API for configuring PCA on Cisco Hosted Collaboration Mediation Fulfillment directly.

# API System Concepts

In order to understand the API, an understanding of two basic concepts is required:

- Models
- Hierarchy

The term "model" is used to describe the types of JSON objects fulfilling purposes such as defining data structures, containing data, defining GUI forms, mapping data from devices or other models. The system employs the following types of models:

- Data Models
- Device Models
- Domain Models
- Relations
- Views

Data in the system is represented using Data and Device models.

Device models are generated from the application API of entities that are provisioned on devices.

Domain models, relations and views wrap the Data or Device models by means of references to them.

Data models can be created and are stored in the database. Data models contain a JSON schema/metadata for the entities exposed by the underlying database. The schemas for the data models are stored in the database and represent the structure that instances of the data model conforms to.

Device models interface with devices and services on the system. For example:

- Unified CM device models interface with the Call Manager's AXL SOAP API.
- CUC device models interface with Unity Connection's RESTful API.

The ability to rapidly develop and deploy new device interfaces provides an extensible mechanism to add support for additional provisioning tasks or additional southbound integration into other business systems.

Domain models act as "containers" of other data-, device- and domain models along with provisioning workflows to represent the management of a created feature.

Relations do not store data on the system. Instead, they relate groups of resource types such as device models, data models or other domain models.

Views provide a mechanism to define an arbitrary schema which can be used to define a user input screen.

# Hierarchy

A system hierarchy node is present at first startup of the system. Each entity that is attached to the hierarchy has an address represented by a PKID, that is defined as a standard URI. Hierarchies can be created under the system hierarchy node, because the hierarchy is exposed as a RESTful API. API calls are made with reference to the hierarchy.

For more information, refer to the Hierarchy section in the *Cisco Unified Communications Domain Manager, Release 11.5(1) Planning and Install Guide*.

# Basic REST

The system uses a REST (Representational State Transfer) API. For details on this type of API, see for example:

- http://en.wikipedia.org/wiki/Representational_state_transfer

# API Traversal

The system represent the reference of an entity in the system as http://en.wikipedia.org/wiki/HATEOAS (HATEOS). Each reference position is represented by an object pair PKID and href.

A client integrates with Cisco Unified Communications Domain Manager 10.x/11.5(x) entirely through hypermedia dynamically provided by the application and does not need any prior knowledge of how to interact with the system other than a generic understanding of hypermedia. This means that no http://en.wikipedia.org/wiki/Web_Application_Description_Language is provided. This also means that the client and the application can be decoupled in a way that allows the application to evolve independently.

A client enters the the application through a simple fixed URL. All future actions the client may take are discovered within resource representations returned from the server

The URL tree information is obtained in the form of a list response from the application endpoint:

GET /api/?format=json

This response emulates the HierarchyNode list response and utilizes the parent and children in the meta references section of the response as discussed in Meta Data References.

# Request and Response Patterns

The request and response patterns between service requester and Cisco Unified Communications Domain Manager 10.x/11.5(x) is summarized below.

For synchronous operations:

1. Service Requestor sends an accessor (e.g. Get, List) request with request parameters.
2. Either:
   a. responds synchronously with a Get/List response.
   b. responds synchronously with a fault response.

For asynchronous operations:

1. Service Requestor sends a mutator (e.g. Add, Modify, Delete) request with parameters.
2. The Add/Update/Delete transaction is scheduled in the transaction queue with a transactionID.
3. Responds synchronously with either:
   a. An Add/Update/Delete response and a transactionID.
   b. A fault response.
4. The external system either:
   a. Polls the system to retrieve the status of the transaction as needed, or
   b. Specifies a callback URL (with an optional username and password if the interface is secured (recommended)) and waits for a asynchronous transaction status callback (recommended).

      When the transaction completes, Cisco Unified Communications Domain Manager 10.x/11.5(x) sends an async transaction status callback message to the callback URL specified in the request.

# Anatomy of an API Request

## General Structure of the API

The Cisco Unified Communications Domain Manager 10.x/11.5(x) API accesses system resources or tools.

- **Resources**

The general structure of an API URL for accessing a system resource (an endpoint) is:

*Method https://servername/api/Version/Resource/Action/?Parameters*

Where:

Method

```
[GET|POST|DELETE|PUT|PATCH]
```

Servername

The installation server determines the base URL, e.g. `https://servername`. In a cluster environment, this is the address of the web proxy node. Refer to the Install Guide for cluster deployment information.

Version

```
(int:major[.int:minor])/
```

Resource

```
(str:modeltype/str:modelname)[/pkid]
```

Action

```
[add|create|list|update|remove|configuration_template|field_display_policy|
 export_bulkload_template|bulk_update|clone|move|export|execute|translation|
 migration|help|+tag|+tag_version|+[non-CRUD operation|CustomWF|
 other custom actions (see API Reference Guides)]]
```

Parameters

```
[(str:api parameter)[&(str:api parameter)...]]
```

The HTTP methods and parameters are described in relevant sections. The different resources supported in the system are described in the API Reference Guides.

- **Tools**

For tools, the general structure of the URL structure is for example:

*[GET|POST] /api/tool/(str:tool_name)/*

# Format

The system API supports the following format HTTP headers when handling and responding to requests.

| Field Name | Description | Value |
|---|---|---|
| Content-Type | The format type of the body of the request (used with POST and PUT requests) | application/json |
| Content-Type | The format type of the body of the request (used with PATCH requests) | application/json-patch+json |
| Accept | Content-Types that are acceptable in response | application/json |

# Authentication

The system controls access to its service through HTTP basic authentication. The technique is defined in section 11.1 of RFC1945 which is simple to implement, uses standard HTTP headers.

The HTTP Basic Access Authentication requires authorization credentials in the form of a user name and password before granting access to resources in the system. The username and password are passed as Base64 encoded text in the header of API requests.

The HTTP header format for authentication is defined in the table below.

| Field Name | Description | Value |
|---|---|---|
| Authorization | Basic authentication is supported. | Basic [Base64 encoded credentials] |

For example:

The Base64 encoded credentials for user name of joe and a password of bloggs.

For example, from a command line (note the removal of the new line in the `echo` command):

```
$ echo -n "joe:bloggs" | base64
am9lOmJsb2dncw==
```

the header is:

```
Authorization: Basic am9lOmJsb2dncw==
```

For example, using **curl**:

```
curl -k -H "Authorization: Basic am9lOmJsb2dncw==" 'https://hostname/api/data/MyModel/'
```

It is required that all requests be conducted over a secure session, such as HTTPS or SSL.

A Cisco Unified Communications Domain Manager 10.x/11.5(x) self-signed certificate needs to be installed into a local trust store of the client application.

# Authorization

The access profile of a user determines whether he or she can perform a given operation on a model. The user can also only access items below the position they are defined in the hierarchy.

# HTTP Methods

The API supports the following HTTP methods:

**GET**

- Used to query a resource or a list of resource.

**POST**

- Used to create a new resource.
- The data is submitted as a JSON object.
- The return value is the PKID of the resource.

**PUT**

- Used to update the data of a resource.
- The resource URL includes the resource PKID.
- The data to be updated is submitted as a JSON object.

**PATCH**

- Used to update the data of a resource.
- PATCH request body in JSON Patch format
- Content-Type is "application/json-patch+json"
- JSON Patch: http://tools.ietf.org/html/rfc6902

**DELETE**

- Used to delete a resource.

- The resource URL includes the resource PKID.
- The DELETE method can also be used to delete multiple resources on one request as a "bulk delete".

# PUT Versus PATCH

For PUT methods the resource data is replaced with the data specified in the request. All fields of the resource are replaced with the fields in the request.

This means that:

- Fields not present in the request that are present in the resource are dropped from the resource.
- Fields present in the request that are not present in the resource are appended to the resource.
- The data of fields present in the request is used to update fields that already exist in the resource.

PATCH methods operate in two modes depending on the content type:

- `Content type: application/json`
- The values of data fields present in the request is used to update the corresponding resource fields. This means that:
  - Fields present in the request but not in the resource are appended to the resource.
  - The value of each field that is already present in the resource is updated from the request data.
  - Field values that are set to null in the request are dropped from the resource.
  - Fields that are present in the resource but not in the request are left untouched.
- `Content type: application/json-patch+json`
  - Existing resource data is patched according to RFC6902.

Modifying data fields:

- To drop the field from a data model, specify null as the parameter value (i.e. `{"field": null}`).
- To blank out a string value set the parameter value to an empty string (i.e. `{"field":""}`).

# API Parameters

The hierarchy parameter is required for each API request and can be specified as any of the following:

- the primary key id (PKID) of the hierarchy node in the form of a Universally Unique ID(UUID), for example `1c055772c0deab00da595101`
- in dot notation, for example `ProviderName.CustomerName.LocationName`

To obtain the PKID of a hierarchy node, refer to the `path` element in the metadata of `data/HierarchyNode` resource.

**Note** For the purposes of simplifying the documentation, the hierarchy API parameter `&hierarchy=[hierarchy]` is not included in all examples in this document. Specifying the hierarchy is however required in all API requests where the instance PKID is not referenced. In the examples, `[hierarchy]` is substituted with the caller's hierarchy id.

The system API supports the following request parameters for data format when handling requests.

| Key | Description | Value |
|---|---|---|
| format | The format type of the body of the request | json |

A request of the following format returns HTML:

```
GET /api/(str:model_type)/(str:model_name)/help/
```

A parameter `&format=json` is not displayed in all examples, but is required for all requests unless a different format is specifically stated.

The Configuration Template can be specified in the POST request parameters for a resource as follows:

```
POST /api/(str:model_type)/(str:model_name)/&template_name=[CFG name]
```

| Key | Description | Value |
|---|---|---|
| template | Apply the Configuration Template with PKID [CFG pkid] to the payload of the POST request. | [CFG pkid] |
| template_name | Apply the Configuration Template with name [CFG name] to the payload of the POST request. | [CFG name] |

Field Display Policy can be specified in the GET request parameters for a resource as follows:

```
GET /api/(str:model_type)/(str:model_name)/add/
```

| Key | Description | Value |
|---|---|---|
| policy | Return a model form schema where the Field Display Policy with PKID [FDP pkid] is applied to it. Use `policy` with the parameters `schema` and `format=json`. | [FDP pkid] |
| policy_name | Return a model form schema where the Field Display Policy with name [FDP name] is applied to it. Use `policy` with the parameters `schema` and `format=json`. | [FDP name] |

The API can return cached data from the system or data from devices, using the following format:

```
GET /api/(str:model_type)/(str:model_name)/[pkid]/
```

| Key | Description | Value | Default |
|---|---|---|---|
| cached | System will respond with resource information where the data was obtained from cache. (Functionally only applicable to device models and domain models containing device models) | true, false | true |

To identify a single resource, the API call contains the single resource (PKID) using the following format:

```
GET /api/(str:model_type)/(str:model_name)/(pkid)/
```

To obtain the schema or schema rules of a resource, use the following parameters to an API request:

```
GET /api/(str:model_type)/(str:model_name)/?
  hierarchy=[hierarchy]&schema=true&schema_rules=true&schema_version=[version_no]
```

| Key | Description | Value |
|---|---|---|
| schema | Return the schema of the resource. Use with the parameter `format=json` | true, false |
| schema_rules | Return the GUI Rules and Field Display Policies of the resource if available. Use with the parameters `format=json` and `schema` to see `schema_rules` in the response. | true, false |

The system API supports the following API request parameters for when specifying the format of and structure of the resources to list.

| Key | Description | Value | Default |
|---|---|---|---|
| skip | The list resource offset | | 0 |
| limit | The maximum number of resources returned | | 50 |
| count | Specify if the number of resources should be counted. If false, the `pagination` object in the response shows the `total` as 0, so no total is calculated and the API performance is improved | | |
| order_by | The summary attribute field to sort on | | First summary attribute |
| direction | The direction of the summary attribute field sort (asc:ascending, desc: descending) | asc, desc | asc |
| summary | Only summary data is returned in the data object | true, false | true |

| Key | Description | Value | Default |
|---|---|---|---|
| traversal | The direction of the resource lookup of resources tied to the hierarchy tree from the hierarchy node provided as parameter | up, down, local | down |

Models that have the list action defined in their schema can also be filtered by using a number of URL filter parameters in parameter sets of four key-value pairs.

Filters also apply to the api/tool/Transaction/ endpoint, which has additional filter functionality to filter by transaction ID. Refer to the topic on Filter Transactions.

These parameters can be added in addition to the parameters available to list resources as in the topic on API Parameters.

A single filter query can contains one or more sets of the following four parameters:

| Key | Description | Value | Default |
|---|---|---|---|
| filter_field | The model attribute name to filter. | The name of the attribute in the list of `summary_attrs` in the model schema. | 0 |
| filter_condition | The matching operator for the `filter_field`. If `equals` is used in a condition, then other filter sets are ignored. | One of the conditions below, applied to a filter_text string value.<br>• `startswith`<br>• `endswith`<br>• `contains`<br>• `notcontain`<br>• `equals`<br>• `notequal` | `contains` |
| filter_text | A text string applied to the `filter_field` by a `filter_condition`. | Plain text | |
| ignore_case | Additional specifier applied to the case of the `filter_text`. | Either `true` or `false`. | `true` |

Example showing a single filter set:

```
GET /api/(str:model_type)/(str:model_name)/?
 hierarchy=[hierarchy]
 &filter_field=[attribute_name]
 &filter_condition=startswith
 &filter_text=John
 &ignore_case=false
```

If more than one filter set is used, all similar keys are grouped, so that the key position indicates the filter set. For example:

```
GET /api/(str:model_type)/(str:model_name)/?
 hierarchy=[hierarchy]
 &filter_field=[attribute_name]
 &filter_field=[attribute_name2]
 &filter_condition=startswith
 &filter_condition=endswith
 &filter_text=John
 &filter_text=an
 &ignore_case=false
 &ignore_case=false
```

The two filter sets in this example, are:

- `&filter_field=[attribute_name]`
- `&filter_condition=startswith`
- `&filter_text=John`
- `&ignore_case=false`

and

- `&filter_field=[attribute_name2]`
- `&filter_condition=endswith`
- `&filter_text=th`
- `&ignore_case=false`

It is possible to submit mutator type operations with API parameters to complete synchronously, in which case the synchronous response to the transaction either includes the status of the transaction or a fault response. This is not recommended as long-running transactions or a busy system may exceed the HTTP timeout.

This is only available for models where the actions in the meta data contains `support_async`.

| Key | Description | Value | Default |
|---|---|---|---|
| nowait | Controls the API synchronous or asynchronous behavior for requests resulting in transactions. Please refer to the `support_async` property in the model schema under **meta** -> **actions**, for an indication of support per action. | true, false | false |

To specify a specific API version of a resource, use the following parameter to an API request:

```
GET /api/(str:model_type)/(str:model_name)/?
hierarchy=[hierarchy]&api_version=<version_number>
```

| Key | Description | Value |
|---|---|---|
| api_version | Return the the resource with `api_version`. Use with the parameter `format=json` | supported version no. |

Where the parameters below are added to the GET call:

```
schema=true&
format=json&
schema_rules=true
```

then the JSON schema `meta` property will contain `schema_version` in accordance with `api_version=<version_number>`.

For Unified CM device models, the `schema_version` will match the version of the Unified CM that corresponds with the `api_version`, for example:

```
GET /api/device/cucm/(str:model_name)/?
hierarchy=[hierarchy]&
schema=true&
format=json&
schema_rules=true&
api_version=10.1.2
```

Returns:

```
"meta": {
"tags": [],
"cached": true,
"title": "",
"business_key": {},
"schema_version": 10.0,
```

The following table shows the current mapping for `/device/cucm` models:

| api_version | schema_version |
|---|---|
| 10.1.2 | 10.0 |
| 10.6.1 | 10.5 |
| 10.6.2 | 10.5 |
| 10.6.3 | 10.5 |

If no `api_version` is specified in the GET call, then the default `schema_version` is determined by the latest entry in the mapping table.

**Note** For more details on API versioning, refer to the topic on API Backwards Compatibility.

# API Request Headers

API Headers are available for:

- Pagination of choices and macro results in an API call.

  The headers are `X-range` and `Range`, with the starting value as 0. These can be used instead of the `skip` and `limit` API parameters.

For example, the following examples return the same results:

```
GET /api/tool/Macro/?method=evaluate
&hierarchy=[hierarchy]
&input={{fn.lines}}
&skip=0
&limit=6
GET /api/tool/Macro/?method=evaluate
&hierarchy=[hierarchy]
&input={{fn.lines}}
Request headers:
X-Range: items=0-5
Range: items=0-5
```

If the request is items=0-199 (for 200 items) and there are more results, the response will show:

```
Content-Range:items 0-199/999999999
```

In this example, if a subsequent request is for the next 200 items (200-399), the response will also show the total number of items (298) returned by the macro:

```
Content-Range:items 200-399/298
```

- Backward compatibility. The X-Version header is available to take an API version as value. For example:

```
GET /api/data/Countries/?hierarchy=[hierarchy]
&schema=true
&format=json
Request headers
X-Version: 10.1.2
```

Refer to the topics on API backwards compatibility.

# Login and Authorization Tokens

The API includes, as part of responses, a `X-CSRFToken` response header that is set to the CSRF token, for example to `KEMzraBRygy2ZJ7fLuvbfKhAEIPK9D4s`. API clients should source the CSRF token from this header.

The API also includes, as a part of responses, a `csrftoken` cookie containing the CSRF token. This cookie is marked `httponly` and as such is not readable by browser-based client scripts. API clients should not try to source the CSRF token from this cookie.

The `X-CSRFToken` response header and `csrftoken` cookie values are identical.

When performing requests that require CSRF token validation, API clients should follow the general procedure:

1. Prior to performing the principal request, perform a request to the API and retrieve a CSRF token from the resulting response's `X-CSRFToken` response header. The CSRF token remains constant for the duration of a session, so clients could perform this request once per session (post authentication), storing the CSRF token and using it for subsequent requests.

   Clients should also retrieve the `csrftoken` cookie from the response.

2. For the primary request, include a `X-CSRFToken` request header containing the CSRF token as sourced from the response header, as well as the unchanged `csrftoken` cookie.

For example:

```
GET http://localhost:8000/login/

Raw response headers:
Cache-Control: max-age=0
Connection: keep-alive
Content-Encoding: gzip
Content-Language: en-us
Content-Type: text/html; charset=utf-8
```

```
Date: Mon, 20 Apr 2015 09:18:47 GMT
Expires: Mon, 20 Apr 2015 09:18:47 GMT
Last-Modified: Mon, 20 Apr 2015 09:18:47 GMT
Server: nginx/1.4.6 (Ubuntu)
Set-Cookie: csrftoken=KEMzraBRygy2ZJ7fLuvbfKhAEIPK9D4s; httponly; Path=/
sessionid=5d1ccc96cbd7e7f290020aaedd64c1b3; httponly; Path=/
sso_login_url=; Path=/
Transfer-Encoding: chunked
Vary: Accept-Encoding, Cookie, Accept-Language, X-CSRFToken
X-CSRFToken: KEMzraBRygy2ZJ7fLuvbfKhAEIPK9D4s
```

1. Source the CSRF token from response's `X-CSRFToken` header.
2. Retain the CSRF cookie from response's `csrftoken` cookie.
3. Now perform the primary POST /login/ request to login, including the CSRF token as a `X-CSRFToken` request header as well as the unchanged `csrftoken` cookie:

```
POST http://localhost:8000/login/

Raw request headers:
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:37.0) Gecko/20100101 Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8000/login/
Cookie: sessionid=5d1ccc96cbd7e7f290020aaedd64c1b3;
csrftoken=KEMzraBRygy2ZJ7fLuvbfKhAEIPK9D4s; sso_Connection: keep-alive
X-CSRFToken: KEMzraBRygy2ZJ7fLuvbfKhAEIPK9D4s
```

With for example payload as parameters:

```
&username=joe
&password=bloggs
&next=%2F
```

# Non-interactive Login

The following request and endpoint is available on the API:

```
POST <hostname>/noninteractivelogin/
```

The payload:

- Content-Type: application/json
- JSON containing user credentials, for example:

  co

  ```
  {
  "username":"joebloggs@email.com",
  "password":"mysecret"
  }
  ```

The purpose of the endpoint, payload and response is for an API client to be able to better report the reason for failed logins without having to parse the login form.

- If the request is successful:
  - the HTTP response is 200
  - the JSON body is for example:

    ```
    {
    "last_successful_login_time": "2016-04-25T09:50:34Z",
    "num_of_failed_login_attempts": 2
    }
    ```

- the X-CSRFToken value

Upon the first successful login, the `last_successful_login_time` is empty. Upon a subsequet successful login, the `last_successful_login_time` is the login time prior to current session. The `num_of_failed_login_attempts` value is reset to 0 after a successful login.

- If the request fails:
  - the HTTP response is 403
  - the JSON body is:

```
{
"error_message": "Please enter a valid username and password.",
"error_code": 27009
}
```

  - the X-CSRFToken value

# Anatomy of an API Response

## API Response Overview

Below are the typical elements of an API response:

- meta - Meta data.
- data - Actual data contained in the model as name: value pairs.
- schema - Schema describing the structure of the data of the resource, in particular the data types of the names in the name: value pairs in the data.
- resources - An object grouping a list of single resource's meta and data objects in an API list response
- pagination - an object containing pagination data in an API list response

Not all the elements above exist in each response. These differ depending on request parameters and whether response contains a list of resource or a single resource.

## Single Resource Response

A single resource response outline is as follows:

```
{
  "meta": {
    ...
  },
  "data": {
    ...
  },
  "schema": {
    ...
  }
}
```

# Resource List Response

The response object outline is as follows:

```
{
    "pagination": {
        ...
    },
    "meta": {
        ...
    },
    "resources": [{
        "meta": {
            ...
        },
        "data": {
            ...
        }
    },
    {
        "meta": {
            ...
        },
        "data": {
            ...
        }
    }]
}
```

# POST/PUT/DELETE/PATCH Response

Support for synchronous and asynchronous request resulting in transactions, is controlled by the `nowait` parameter in the request URL. The support for asynchronous request handling is indicated in the API schema structure **actions** with the `support_async` property.

The outline of the default synchronous transaction response of mutator transactions when the API parameter `nowait` is set to be false, is as follows:

```
{
  "pkid": "51f7e09bd0278d4b28e981da",
  "model_type": "data/CallManager",
  "meta": {
    "parent_id": {
      "pkid": "51f7d06ad0278d4b34e98134",
      "uri": "/api/data/HierarchyNode/51f7d06ad0278d4b34e98134"
    },
    "summary_attrs": [
      {
        "name": "description",
        "title": "Description"
      },
      {
        "name": "host",
        "title": "Host Name"
      },
      {
        "name": "port",
        "title": "Port"
      }
    ],
    "uri": "/api/data/CallManager/51f7e09bd0278d4b28e981da"
  },
  "success": true
}
```

The outline of the synchronous response to asynchronous mutator transactions when the API parameter `nowait` is set to be true, is as follows:

```
{
  "href": "/api/tool/Transaction/cfe8a8fd-98e6-4290-b0c3-2dfa2224b808",
  "success": true,
  "transaction_id": "cfe8a8fd-98e6-4290-b0c3-2dfa2224b808"
}
```

To retrieve (for example by polling) the transaction status of any mutator transactions, use the *transaction_id* in the synchronous response to the asynchronous mutator transaction as follows:

```
GET /api/tool/Transaction/cfe8a8fd-98e6-4290-b0c3-2dfa2224b808
```

The response contains the status ad replay action URL, for example:

```
{
 "meta": {
        "model_type": "tool/Transaction",
        "summary_attrs":
        {
         "name": "name",
         "title": "Name"
        },
        "references": {}
        "actions":
          {
           "replay": {
             "class": "execute",
             "href": "/api/tool/Transaction/cfe8a8fd-98e6-4290-b0c3-2dfa2224b808/
               replay?format=json",
             "method": "GET",
             "title": "Replay"
            }
          }
        }
 "data": {
       "status": "Completed",
       "username": "sysadmin",
       "resource": {
        "hierarchy: "sys",
        "after_transaction": "/api/data/GeneralHelp/5268c7d3a616540a766b91f5/?
          cached=5268f2eba616540a736b926c Entity",
        "current_state": "/api/data/GeneralHelp/5268c7d3a616540a766b91f5/ Entity",
        "before_transaction": "/api/data/GeneralHelp/5268c7d3a616540a766b91f5/
          ?cached=5268c7d3a616540a766b91f7 Entity",
        "pkid": "5268c7d3a616540a766b91f5",
        "model_type": "data/GeneralHelp",
       }
  [...]
```

This mechanism can be used to retrieve the transaction status of any transaction or its sub-transaction, using the PKID of the (sub) transaction.

# Asynchronous Mutator Transaction Status Callback

When using the API parameter `nowait=true`, the service requester can submit optional request meta data - containing a callback URL - with any mutator request by appending the `request_meta` tag to the normal payload of the request.

In order to receive asynchronous transaction status notifications, the requesting system needs to publish an HTTP service to service requests made by the callback URL. An example of a simple http service is provided in a separate section.

The callback operation supports an optional username and password that Cisco Unified Communications Domain Manager 10.x/11.5(x) uses to perform HTTP basic authentication on requests made to the callback

service. The optional elements `external_id` and `external_reference` are explained in the section on correlation identifiers.

```
{
   <Actual request data goes here>,
   "request_meta": {
        "external_id": "3x4mpl3-3xtern4l-FF",
        "external_reference": "Example External Reference-FF",
        "callback_url": "http://my.callbackservice:8080",
        "callback_username": "username",
        "callback_password": "password"
    }
}
```

The following details should be noted here:

- The schema of system resources or system tools do not include reference to the request meta data in the schema definition of each resource in the system.
- The `<Actual request data goes here>` request data needed to for example add a `country_name` instance for data/Countries would be similar to: `"country_name": "South Africa"`.
- The request data for deleting two countries for example would be

  ```
  "hrefs":[
     "/api/data/Countries/534fdf190dd19012066433ce",
     "/api/data/Countries/534fda1d0dd1901206643397"
  ]
  ```

- For the callback service to function, the callback service needs to be accessible from the fulfillment server.

Upon completion of the asynchronous mutator transaction posted with a callback URL, the application POSTs an HTTP request (asynchronous transaction status callback) to the callback service specified by the callback URL. The callback service needs to respond with an HTTP 200 ACK *before* internal processing of the callback. The callback includes the transaction ID sent to the requesting system as part of the synchronous response. To correlate the asynchronous transaction status callbacks with the original request, the requesting system would need to record the `transaction_id` returned in the synchronous response.

The HTTP headers and the payload of the asynchronous transaction status callback includes the following information:

HTTP headers:

```
{
    'accept-encoding': 'identity',
    'authorization': 'BasicdXNlcm5hbWU6cGFzc3dvcmQ=',
    'content-length': '275',
    'content-type': 'application/json',
    'host': 'localhost: 8080'
}
```

Payload:

```
{
    "external_id": "3x4mpl3-3xtern4l-FF",
    "external_reference": "ExampleExternalReference",
    "status": "Success",
    "transaction": {
        "href":
          "http:
//my.fulfillmentserver/api/tool/Transaction/e6ac7c1e-c63a-11e3-9af5-08002791605b/",
        "id": "e6ac7c1e-c63a-11e3-9af5-08002791605b"
    }
}
```

The following details should be noted here:

- Correlation identifiers (see correlation identifiers) are included in the payload if they are present.
- The status of the transaction is as in the transaction log: Fail or Success.

The transaction status is not affected by the response of the HTTP service published by the requesting system. The transaction log information includes the callback request and the response returned by the callback service published by the external system.

For transactions with multiple sub-transactions, a single transaction status callback request is made upon the completion of the parent transaction. Transaction status callbacks are not supported for the parent transactions tool/BulkLoad and tool/DataImport.

In the event that the transaction status callback is not received by the external system due to for example a network outage, the external system can poll to retrieve the transaction status. For example:

```
GET /api/tool/Transaction/e6ac7c1e-c63a-11e3-9af5-08002791605b
```

# Example of an Asynchronous Mutator Transaction with nowait=true

Request:

```
POST http://172.29.232.238/api/data/Countries/?hierarchy=1c0ffee2c0deab00da595101&nowait=true
Payload of the request:

{'country_name': 'Callback Created Example Country Name',
 'request_meta': {'callback_password': 'password',
                  'callback_url': 'http://localhost:9365',
                  'callback_username': 'username',
                  'external_id': '3x4mpl3-3xt3rn4l-7d',
                  'external_reference': 'External Ref'}}
```

Synchronous response:

```
{
href: "/api/tool/Transaction/e6ac7c1e-c63a-11e3-9af5-08002791605b"
success: true
transaction_id: "e6ac7c1e-c63a-11e3-9af5-08002791605b"
}
HTTP 202 ACCEPTED
```

Asynchronous transaction status callback (console output of the simple http service provided in the separate example section):

```
POST - 2014-04-17 16:16:43.737509


Headers:

{'accept-encoding': 'identity',
 'authorization': 'Basic dXNlcm5hbWU6cGFzc3dvcmQ=',
 'content-length': '275',
 'content-type': 'application/json',
 'host': 'localhost:8080'}


Raw Callback Body:

'{"status": "Fail", "transaction":
{"href":
   "http://django.testserver/api/tool/Transaction/34866060-fd47-11e3-88dd-080027880ca6/",
 "id": "34866060-fd47-11e3-88dd-080027880ca6"},
 "resource": {"hierarchy": "1c0ffee2c0deab00da595101",
              "model_type": "data/Countries",
              "pkid": "53ac3d41c9527062809c0021"},
              "external_reference": "External Ref",
              "external_id": "3x4mpl3-3xt3rn4l-7d"}'

Pretty Callback Body:
```

```
{u'external_id': u'3x4mpl3-3xt3rn4l-7d',
 u'external_reference': u'External Ref',
 u'resource': {u'hierarchy': u'1c0ffee2c0deab00da595101',
               u'model_type': u'data/Countries',
               u'pkid': u'53ac3d41c9527062809c0021'},
 u'status': u'Fail',
 u'transaction': {u'href':
   u'http://django.testserver/api/tool/Transaction/34866060-fd47-11e3-88dd-080027880ca6/',
                  u'id': u'34866060-fd47-11e3-88dd-080027880ca6'}}

localhost - - [17/Apr/2014 16:16:43] "POST / HTTP/1.1" 200 –
```

# Correlation Identifiers

In order to allow an external system use its own identifiers to cross-reference transactions in the system, the Cisco Unified Communications Domain Manager 10.x/11.5(x) API supports two external identifiers for all transactions. This allows the external system to:

1. Tie together multiple transactions in the system (using for example an order number)
2. Track individual requests in the system using the external IDs.

External identifiers are not supported for the parent transactions tool/BulkLoad and tool/DataImport.

The transaction log includes these two IDs and the transaction log, as in the figure below called *An example transaction log showing IDs*.

You can obtain the details of the parent transaction with a given ID by using the following API call:

```
GET http://my.fulfillmentserver/api/v0/tool/Transaction/?hierarchy=1c0ffee2c0deab00da595101&

     filter_condition=contains&
     format=json&
     filter_text=3x4mpl3-3xtern4l-FF&
     filter_field=external.id
```

You can obtain the details of transactions tied together using an external reference number using the following API call:

```
GET http://my.fulfillmentserver/api/v0/tool/Transaction/?hierarchy=1c0ffee2c0deab00da595101&

     filter_condition=contains&
     format=json&
```

```
filter_text=Example%20External%20Reference-FF&
filter_field=external.reference
```

*Figure 1: An example transaction log showing IDs - snippet A.*

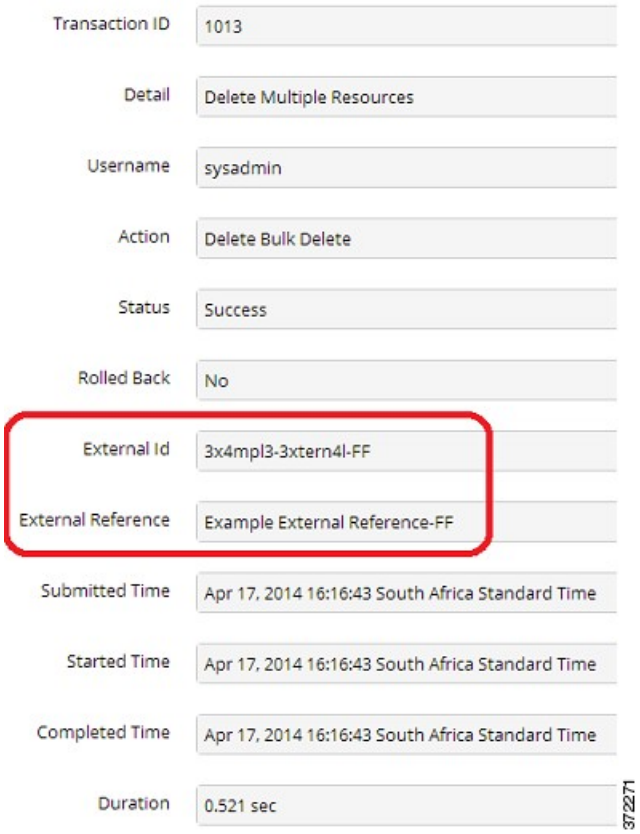
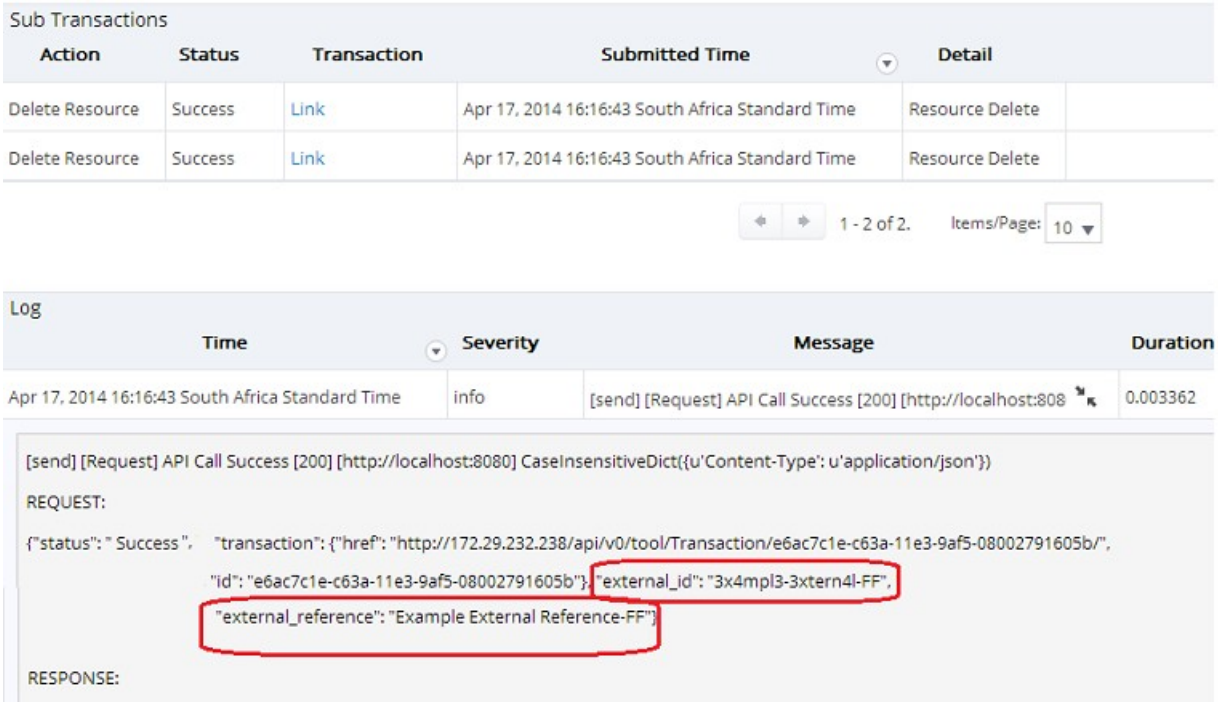



*Figure 2: An example transaction log showing IDs - snippet B.*




# Example Of A Simple HTTP Server

The following code is an example of a simple HTTP server that can be used to test basic async transaction status callback operations. The code is not intended for actual use.

**Note** The HTTP 200 ACK is sent asynchronously *before* internal processing of the callback.

```
#!/usr/bin/env python
from datetime import datetime
import SimpleHTTPServer
import SocketServer
import logging
import cgi
import json
from pprint import pprint
PORT = 8080

class ServerHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        SimpleHTTPServer.SimpleHTTPRequestHandler.do_GET(self)

    def do_POST(self):
        self.send_response(200)
        self.wfile.write("ACK")

        # Insert internal processing here.
        # Below is an example of internal processing that simply prints out the
        # callback request.
        print "\nPOST - {}".format(datetime.now())
        print "Headers:"
        pprint(dict(self.headers))
        print "\nRaw Body:"
```

```
        body = self.rfile.read(int(self.headers['Content-Length'])).decode('utf-8')
        pprint(body)
        print "\nPretty Body:"
        pprint(json.loads(body))


Handler = ServerHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "Serving at port", PORT
httpd.serve_forever()
```