Real Estate Association APIs and Data Standards: Critical for Future Success

Clareity Consulting
July 20, 2017

Contents

Executive Summary

The preamble to the REALTOR® Code of Ethics begins with the famous phrase, “**Under all is the land.** Upon its wise utilization and widely allocated ownership depend the survival and growth of free institutions and of our civilization.” In similar fashion, organized real estate runs best upon the wise utilization of member information. Understanding what members are interested in, their satisfaction with the association, the services they use, their interests in political advocacy, volunteering, and leadership opportunities, are just a few examples of the kind of information associations need to be able to leverage to create an optimal relationship with the membership. Being able to use data about individual members to show them information they want to see, when and where they like to see it, rather than bombarding them with every possible association communication is important to maintaining that relationship. Perhaps there should be a preamble for association management documents stating, "**Under all is the member.** Upon the wise utilization of member information depend the survival and growth of the real estate association."

There are two association problems related to member information that have needed to be solved for some time:

1. Member information being stuck in silos where it is hard to combine and analyze it to solve business objectives.
2. Association management systems, where much of the member data has been stored, is not (and cannot possibly be) the best tool for every possible software function needed by the association, yet real-time data integration with third party tools was not easy.

The answer to these problems is to create an API (Application Programming Interface) for AMS (association management systems). This would enable programmers to easily move information between AMS and other software packages that have their own APIs. Currently, some of the AMS have APIs, but there are some issues with these APIs:

1. They are not publicly documented or advertised. Most real estate industry software developers don't know they exist.
2. It is "early days" for these APIs – they may or may not support all of the business objectives efficiently, as the industry has never had a formal discussion of those business objectives.
3. Each AMS vendor API has been developed independently, so there's not a data standard or common way to interact with these systems and their data - just like MLS data until fairly recently.
4. There is no common marketplace for sharing or buying / selling software add-ons for AMS.

This paper is a starting point for addressing wider industry awareness of AMS APIs and what can be accomplished by using them if associations show demand and a developer community can be organized. This paper will also serve as a starting point for discussing possible business objectives for the APIs. Finally, this paper should inspire both business and technical experts to engage with the AMS providers at RESO, to see what can be done to ensure that the APIs are designed to meet business objectives and – ideally – to create a data dictionary of field names that should be common to all AMS and create a common mechanism for accessing that member information.

Business Objectives

When designing an API, it must be able to address all of the common business needs an association, its vendors, and its stakeholders (like brokers), may have for accessing its member information. This is not meant as a comprehensive list, but following are examples provided by various industry colleagues[1] meant to inspire additional ideas:

1. Targeting communications to members:
    a. Upcoming license renewal date reminders
    b. Class registration reminders
    c. Messages sent only to new members
    d. Members that have attended most events in time period (likely to attend another)
    e. Members that have attended most classes in time period (likely to attend another)
    f. Members that have attended a specific class (e.g. MLS Basics) but not another (e.g. MLS Advanced Topics).
    g. Message customization based on time in the business
    h. Based on past RPAC Contribution level in a time period
    i. Based on political district of home address
    j. To display upcoming calendar items and notifications specific to one user

2. For data display/use elsewhere:
    a. Pulling class data for display on association website
    b. Brokers pulling languages, designations, course completion, CE hours in a time period for agents

3. Validating user status, licensure, and permission:
    a. For websites and applications to determine permission to log in, see certain content or use certain functions
    b. Pulling user status and key data from NRDS
    c. Pulling licensure status from state licensing boards (if they would use the standard)

4. Pulling data into the AMS or into a third party tool for additional communication segmentation and analytics:
    a. social media interactions
    b. production data (list/sell, buyers represented)

5. Syncing data both to and from a 3rd party:

[1] Many thanks to Dave Conroy, Ryan Castle, Chip Ahlswede, David Arbit, Dan Sale, Michael Seguin, and Charles Willits, Ramco, MMSI, and Rapattoni Corporation.

   a. Communications tool (member data, communications preferences, messages read confirmations, click-through, etc.)
   b. Event management tool (member data, event/class dates, info, registration links, capacity, capacity remaining)
   c. Support software or other software validating member status, viewing call/case history, writing updates back to the AMS

6. To create a management / governance dashboard:
   a. Total # of new members in period of time
   b. Mapping where members live
   c. Mapping where members' offices are

Ideally these ideas could be refined and many additional ideas could be developed. This would both inform API standards development and allow associations to coordinate software development – both on the AMS and third party software developer side to provide the functions that use the API-exposed data to achieve those objectives.

Better Business by Technical Design: CRUD and FEoC

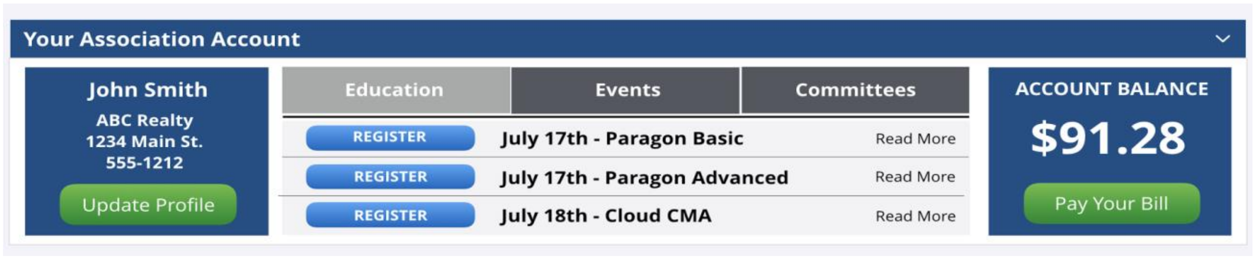
One design consideration of note, based on the objectives listed above, is that the API should not merely be a way to get data out of the AMS (the way RETS was *originally* focused on getting data out of an MLS system) but should rather support "CRUD" – create, read, update, and delete. To quote Michael Seguin, Director of Technology for the Contra Costa Association of REALTORS®:

> "I want a system that handles CRUD, and CRUD only, for the basic member record. Everything else is undoubtedly addressed more efficiently, by an external system. I want it to be open source, and functionality can be contributed via modules to speed that industry specific development of functionality. But at the end of the day, can you imagine a world in which a dedicated, monolithic system does a better job with event creation / display / registration / curriculum? That does a better job of flexible payment options, and integrating with multiple, rapidly evolving payment ecosystems? That does a better job of, well, anything other than industry specific costs (variable dues), ethics and licensing business logic? Everything else is non-core and has no business whatsoever in a real estate AMS, IMHO."

One could argue that a good AMS could at least provide the basic modules and workflows, but what Michael is basically asking for is what we call "front end of choice" (FEoC) – a concept that is gaining some traction in the MLS market but has not seen significant discussion yet in the AMS market. FEoC would allow associations to buy "best in breed" solutions for specific tasks rather than depending solely on an AMS or having to use 3rd parties without a real-time efficient way to move data between systems.

How Clareity is using these APIs

Clareity is leveraging these AMS APIs for customers. In Clareity's SSO Dashboard product, for AMS that provide an API that supports the functionality, Clareity is providing post-login fast display of *relevant* education, events, committee information, and account balance – as well as deep links so members can immediately take action without logging in again. Clareity also provides notifications for classes and events for which the user has already registered.



API Standardization Beginning

According to Jeremy Crawford, CEO of the Real Estate Standards Organization (RESO), "RESO is working with AMS providers Rapattoni, Ramco, MMSI, and LAMPS to move forward with data standardization in a formal manner. Rapattoni has developed an "alpha" membership Data Dictionary for all to start reviewing to allow for the standardized data flow of information from AOR to AOR, broker to broker, and everything in between and hopefully leveraging the RESO Web API as the transport mechanism long term."

NAR's Data Strategy Committee and NAR's Association Executives Committee will be participating in the standardization effort. While RESO is working on the technical aspect of this project, it's the national, state, and local associations that will sort through the political concerns related to data sharing and standardization. This is similar work to what was accomplished for the benefit of MLSs through NAR's Multiple Listing Service committee when NAR helped usher the adoption of the RETS data dictionary.

If you want to help drive real estate data standards, including AMS standards, your organization has to be a member of RESO if it is not already one (https://www.reso.org/join/) and Associations will see the best possible outcome for standards that will improve the future of association management if they actively participate in the standards process, providing whatever insights they can, whether on the business side, technical, or both.

Chip Ahlswede, CEO of the Beverly Hills/Greater Los Angeles Association of Realtors (BHGLAAR), had his association join RESO just for the opportunity to participate in the creation of an AMS API standard. Chip said, "Members expect a unique experience and the key to giving them exactly what they are looking for is in the data. Unlocking the data means unlocking the personalized REALTOR experience our members need to succeed. That is why we joined RESO." Hopefully many associations will follow Chip's leadership example.

The work on the AMS API will primarily be happening as a sub-group under the Data Dictionary workgroup, so if you're already a member of RESO, join that group and you will be kept up to date on upcoming meetings.

APIs and Examples

Following is some background on some of the AMS APIs, some technical examples to demonstrate how some of them work and to illustrate the current differences between them. Thanks to Ramco and Rapattoni Corporation for their cooperation in providing these examples.

Ramco

RAMCO currently serves 110 local associations with 245,070 members – that's 10% of the local AMS market as measured by number of associations and 20% of the AMS market as measured by number of members. In addition, RAMCO is also the AMS of choice for 30 state and territorial associations serving 588,825 members. In total, the system is used by over 875 association and MLS staff nationwide. The system is also used in regional MLS installations: Northstar & Intermountain are live, with MIRealSource undergoing conversion. NAR's Center for Specialized Real Estate Education (CSRE) (100,000+ members) and their governmental affairs operation in DC are also RAMCO-powered. It's also been chosen as the platform for NAR's Master Member Profile System (MMPS) and is now being phased into the Chicago office as their primary membership system on a department by department basis.

Ramco was built on the Cobalt AMS platform, which in turn was built upon the Microsoft Dynamics CRM platform. The advantage of that Microsoft platform is that it came with a built-in API with full CRUD (create, read, update, and delete) capabilities, and which leverages Odata just like the RESO Web API.

This API has a developer community that goes beyond the real estate industry. The marketplace can be found here: https://appsource.microsoft.com/en-us/marketplace/apps?product=power-bi&page=1
Following are some of the third party apps Ramco customers are using that are integrated into the AMS using the API:

- **Click Dimensions bulk emailing.** Downloads email metrics into their Ramco system which allows them to track opens, click-throughs, and other metrics in the member's record and lots more. There are also Dynamics apps for Constant Contact & Mail Chimp.
- **SavvyCard**. Powers an app that members can give to their customers that allows them to call, email, text their Realtor, check the MLS, etc.
- **Easy Territory** – associations can geomap and heatmap their membership using all kinds of sort criteria – membership density, member office locations, RPAC donors, members' voting precincts, etc.

- **Target Smart.** State associations can download tons of voter and demographic information on their members through NAR's agreement with TargetSmart. All the available Target Smart fields are already installed in Ramco. Illinois and Washington AORs were pioneers in using it this past election cycle.
- **Higher Logic.** Allows associations to stand up a web "community" site. Missouri AOR calls theirs "The Landing Page" and used it to form a number of interest-based communities where members can exchange ideas, opinions, tip, techniques, etc. Ramco also uses it for their User Community. *Note: Clareity integrates The Landing Page into all Missouri-based Dashboard customers.*
- **DotSignal.** Texting service. RA of the Palm Beaches provides it to their brokers to be able to text their agents. The association uses it to text their members for things like class/event reminders, get out the vote, etc.

Following is an example of the core call and response to use the API to request for committee membership records and current/previous year contribution amounts.

Call:

```
$post['key'] = API_KEY;

$post['operation'] = 'GetEntities';

$post['entity'] = 'contact';

$post['attributes'] =
'FirstName,LastName,ramco_nrdsid,EMailAddress1,ramco_CurrentYearContribution,ramco_CurrentYear
MinusOne,cobalt_contact_cobalt_CommitteeMembership/cobalt_CommitteeId,cobalt_contact_cobalt_
CommitteeMembership/cobalt_TermBeginDate,cobalt_contact_cobalt_CommitteeMembership/cobalt_
TermEndDate';
```

Raw JSON Response:

{"ResponseCode":200,"ResponseText":"OK","Data":[{"LastName":"Jones","EMailAddress1":"someone@comcast.net","ramco_currentyearminusone":"120.0000","ramco_nrdsid":"456011638","FirstName":"Lisa","ramco_currentyearcontribution":"0.0000","cobalt_contact_cobalt_committeemembership":[{"cobalt_TermBeginDate":{"Type":"DateTime","Value":1325419200,"Display":"2012-01-01T12:00:00"},"cobalt_TermEndDate":{"Type":"DateTime","Value":1356955200,"Display":"2012-12-31T12:00:00"},"cobalt_CommitteeId":{"Type":"EntityReference: cobalt_committee","Value":"d9e4eda0-2d92-42fd-b430-3f66c488ea0c","Display":"young professional net comm"}},{"cobalt_TermBeginDate":{"Type":"DateTime","Value":1293883200,"Display":"2011-01-01T12:00:00"},"cobalt_TermEndDate":{"Type":"DateTime","Value":1325332800,"Display":"2011-12-31T12:00:00"},"cobalt_CommitteeId":{"Type":"EntityReference: cobalt_committee","Value":"6408f74d-f7c2-4481-912b-4fdbbd7a0c13","Display":"Young Professional Network"}},{"cobalt_TermBeginDate":{"Type":"DateTime","Value":1293883200,"Display":"2011-01-01T12:00:00"},"cobalt_TermEndDate":{"Type":"DateTime","Value":1325332800,"Display":"2011-12-31T12:00:00"},"cobalt_CommitteeId":{"Type":"EntityReference: cobalt_committee","Value":"d9e4eda0-2d92-42fd-b430-3f66c488ea0c","Display":"young professional net comm"}}]}]}

Following is an example of the core call and response to use the API to request members where class registrations where attended = true and it was EITHER the short sales course or the effective RE practices course:

Call:

```
$post['key'] = API_KEY;

$post['operation'] = 'GetEntities';

$post['entity'] = 'cobalt_ClassRegistration';

$post['filter'] = 'cobalt_Attended<eq>true and (cobalt_ClassId<eq>054573dc-652c-4361-af95-f551abd1914e or cobalt_ClassId<eq>a75aacdc-ef6e-4df9-8704-4b2b23108d35)';

$post['attributes'] = 'cobalt_classid,cobalt_Attended,cobalt_contact_cobalt_classregistration/FirstName';
```

Raw JSON Response:

{"ResponseCode":200,"ResponseText":"OK","Data":[{"cobalt_Attended":"true","cobalt_classid":{"Type":"EntityReference: cobalt_class","Value":"054573dc-652c-4361-af95-f551abd1914e","Display":"LI10SSAF - Short Sales & Foreclosures-SFR"},"cobalt_contact_cobalt_classregistration":{"FirstName":"Stacey"}},{"cobalt_Attended":"true","cobalt_classid":{"Type":"EntityReference: cobalt_class","Value":"054573dc-652c-4361-af95-f551abd1914e","Display":"LI10SSAF - Short Sales & Foreclosures-SFR"},"cobalt_contact_cobalt_classregistration":{"FirstName":"John"}},{"cobalt_Attended":"true","cobalt_classid":{"Type":"EntityReference: cobalt_class","Value":"054573dc-652c-4361-af95-f551abd1914e","Display":"LI10SSAF - Short Sales & Foreclosures-SFR"},"cobalt_contact_cobalt_classregistration":{"FirstName":"Nikki"}}]}

See the Full API Documentation toward the end of this paper to more fully understand the capabilities of this API and the level of its documentation.

Rapattoni Corporation

Rapattoni Corporation's association management systems currently serve over 200 local associations representing about 80% of REALTORS®, as well as 14 state associations. Rapattoni built their API utilizing the Odata standard to make it easier for 3rd parties to adopt and pull from their API. In structure, their API is similar to the RESO Web API in how it leverages the existing Oauth2 and Odata standards. The API is read-only at this time, with plans to add write capabilities later this year.

Rapattoni leverages their custom application with 45 years of business experience in the membership space, to help lead the way with RESO in standardizing AMS data. They will be working with 3rd party developers to mutually enhance their product and add value to the AMS space.

Following is an example of the call and response to use the API to request for membership records currently assigned a designation and what those designations are.

The full Odata query is:

Members?$expand=MemberDesignations/Designations&$filter=MemberDesignations/any(c: c/Designation ne null)&$select=First_Name,NRDS_ID,MemberDesignations/Designations/Designation,MemberDesignations/Designations/Description

This breaks down to be:

Entity Set: Members

$expand= MemberDesignations/Designations

$filter=MemberDesignations/any(c: c/Designation ne null)

$select= First_Name,NRDS_ID,Last_Modify_Date,MemberDesignations/Designation,MemberDesignations/Designation_Date,MemberDesignations/Designations/Description

JSON Results:

{"odata.metadata": "http://redacted/odata/$metadata#Members&$select=First_Name,NRDS_ID,Last_Modify_Date,MemberDesignations/Designation,MemberDesignations/Designation_Date,MemberDesignations/Designations/Description","value": [{"MemberDesignations": [{"Designations": {"Description": "E-PRO"},"Designation": "E-PRO","Designation_Date": "20060607"},{"Designations": {"Description": "Graduate Realtors Institute"},"Designation": "GRI","Designation_Date": "20060607"}],"First_Name": "David","NRDS_ID": "166002354","Last_Modify_Date": "20150727"},{"MemberDesignations": [{"Designations": {"Description": "Certified Internation Property"},"Designation": "CIPS","Designation_Date": "20060613"},{"Designations": {"Description": "Certified Residential Speciali"},"Designation": "CRS","Designation_Date": "20060613"},{"Designations": {"Description": "E-PRO"},"Designation": "E-PRO","Designation_Date": "20060613"}],"First_Name": "Roland","NRDS_ID": "166000219","Last_Modify_Date": "20160127"},{"MemberDesignations": [{"Designations": {"Description": "Certified Residential Speciali"},"Designation": "CRS","Designation_Date": "20060614"},{"Designations": {"Description": "E-PRO"},"Designation": "E-PRO","Designation_Date": "20060614"}],"First_Name": "Paul","NRDS_ID": "166002676","Last_Modify_Date": "20130731"},{"MemberDesignations": [{"Designations": {"Description": "E-PRO"},"Designation": "E-PRO","Designation_Date": "20060913"},{"Designations": {"Description": "Graduate Realtors Institute"},"Designation": "GRI","Designation_Date": "20060913"}],"First_Name": "Julia","NRDS_ID": "166001412","Last_Modify_Date": "20081114"},{"MemberDesignations": [{"Designations": {"Description": "Graduate Realtors Institute"},"Designation": "GRI","Designation_Date": "20060807"}],"First_Name": "Shayan","NRDS_ID": "166010099","Last_Modify_Date": "20130808"},{"MemberDesignations": [{"Designations": {"Description": "Graduate Realtors Institute"},"Designation": "GRI","Designation_Date": "20070419"}],"First_Name": "Robb","NRDS_ID": "166013364 ","Last_Modify_Date": "20160224"},{"MemberDesignations": [{"Designations": {"Description": "Certified Residential Speciali"},"Designation": "CRS","Designation_Date": "20060313"}],"First_Name": "Craig   ","NRDS_ID": "166013654 ","Last_Modify_Date": "20160125"},{"MemberDesignations": [{"Designations": {"Description": "Graduate Realtors Institute"},"Designation": "GRI","Designation_Date": "20060201"}],"First_Name": "William","NRDS_ID": "018350354","Last_Modify_Date": "20060201"}]}

See the Full API Documentation toward the end of this paper to more fully understand the capabilities of this API and the level of its documentation.

MMSI

Mark Richburg, Vice President at MMSI emailed Clareity that MMSI's "API is a full featured RESTful API with RESO data dictionary-compliant field names, where applicable." According to a mutual client / customer that just requested access to the API, MMSI indicated that currently the API "is just used for pull agent and office data." MMSI did not provide documentation of its API or examples of API usage for review and, at the time of publication, had not clarified what API functions are available for their customers to take advantage of currently.

Full API Documentation

Ramco API v 2.0

Updated July 9th, 2016

This document is maintained at https://api.ramcoams.com/api/v2/ramco_api_v2_doc.pdf

General Overview

Intent is to provide an API that allows for discovery of what can be accessed (entity types, their properties and metadata), formulation of queries that can support custom entities and properties, and record management functions using an easy to understand syntax with a minimum of custom api calls.

User Authentication

Users are validated and granted access by passing in an assigned key value as a post parameter with every request. A separate authentication process is not required.

API Call Process

All requests are posted to https://api.ramcoams.com/api/v2/ (note https, ssl is required).

Post parameters control which operation is performed and return values.

The user's key must be provided with every request and will tie to a specific user profile which specifies the appropriate Ramco backend and permissions level.

Parameter names are case-insensitive. Parameter values, however, may be case-sensitive.

Operation Overview

- **GetEntityTypes:** Returns a complete list of all entities in the system.
- **GetEntityMetadata:** Returns complete field list for an entity, along with important information about the field (type, length, etc.).
- **GetEntity:** Returns attribute and/or relationship data for a single entity.
- **GetEntities:** Returns attribute and/or relationship data for 0-N entities.
- **GetOptionSet:** Returns the key/value sets for the specified OptionSet.
- **UpdateEntity:** Allows for update of field-level entity data.
- **CreateEntity:** Allows for the creation of new entity records.
- **DeleteEntity:** Allows for the deletion of existing entity records.

Both GetEntity and GetEntities support querying the attributes of the requested entity and related entities (ex: Can return cobalt_membership data with the Contact).

GetEntity and GetEntities differ in how data is requested. GetEntity uses the record guid and returns 0-1 records. GetEntities supports rich query filters and returns 0-N records.

Response Overview

Response format will be a standardized json package across all responses/operations and will contain:

- **ResponseCode:** Numeric code that generally corresponds to http response codes (200 = OK, 204 = Processed, no content, 400 = Bad Request, etc.). This element is always present in the response.
- **ResponseText:** Text that corresponds to responseCode, providing more detail where required. (ex: “UserPhone is invalid field for Contact entity.”). This element is always present in the response.
- **Data:** Contains the response data set, if any.
- **StreamToken:** To prevent very large GetEntities requests from consuming huge amounts of memory, the resultset will be returned once a memory threshold has been reached even if all results haven’t been returned. If this occurs, the StreamToken will be populated with a token that can be used to fetch the next batch of records.

Ramco Dates

All dates – those returned by Ramco and those used in GetEntities queries – are RFC 3339 format and indicate Greenwich Mean Time. Trailing timezone indicator is not supported.

Examples:

- 2013-12-31T18:30:00
- 2013-12-31

Call Overview

GetEntityTypes

Fetch all entities in the system.

Post params:

- Key = your provided authentication key
- Operation = GetEntityTypes

GetEntityMetadata

Fetch metadata on entity type Contact (includes entity description, attributes and relationships).

Post params:

- Key = your provided authentication key
- Operation = GetEntityMetadata
- Entity = Name of entity being queried (ex: Contact)

GetOptionSet

Fetch the valid value/label pairs for the specified OptionSet (what Ramco calls a picklist)

Post params:

- Key = your provided authentication key
- Operation = GetOptionSet
- Entity = Name of entity being queried (ex: Contact)
- Attribute= Name of the attribute being queried (ex: PreferredPhone)

GetEntity

Fetch the attributes and/or relationships of one specific entity using guid.

Post params:

- Key = your provided authentication key
- Operation = GetEntity
- Entity = Type of entity being queried (ex: Contact)
- Guid = guid of entity to return
- Attributes = comma separated list of attributes to return

  Attributes for related entities can be also be specified by including ‘relationship/attribute’ in the attributes parameter. As an example, the metadata for the Contact entity shows a relationship to cobalt_memberships entity named cobalt_contact_cobalt_membership. To have the query return the status of those memberships, include ‘cobalt_contact_cobalt_membership/statuscode’ in the attribute parameter.

GetEntities

Fetch the attributes and/or relationships of multiple entities of one type using a user-defined filter.

Post params:

- Key = your provided authentication key
- Operation = GetEntities
- Entity = Name of entity being queried (ex: Contact)
- Filter (optional) = User-specified filter string
- Attributes = comma separated list of attributes to return
- StringDelimiter (optional) = User-specified delimiter used to wrap string values (default is #)
- MaxResults (optional) = Maximum number of entities to return

GetEntities Filter syntax

The GetEntities operation supports filtering which allows multiple criteria to be specified using an attribute<operator>value syntax. Multiple filters can be strung together using AND/OR. Nested filters are also supported via parentheses.

Supported operators:

- <eq> = equal to
- <ne> = not equal to
- <gt> = greater than
- <ge> greater than or equal to
- <lt> less than
- <le> less than or equal to
- <sb> string begins with (valid for strings only)
- <sc> string contains (valid for strings only)
- <se> string ends with (valid for strings only)

Strings must be wrapped with a delimiter. The default delimiter is '#' but a user-specified delimiter can be utilized by including the optional StringDelimiter param.

Example filter for records modified on or after Jan 1, 2014, where the LastName field starts with 'S' and the cobalt_EmailVerified is false or null:

'ModifiedOn<ge>2014-01-01 AND LastName<sb>#S# AND (cobalt_EmailVerified<eq>null OR cobalt_EmailVerified<eq>0)'

GetEntities using StreamToken

If the response to a previous GetEntities request includes a StreamToken (see "Response Overview" above), it can be used to continue fetching results. The post params are different in this case.

Post params:

- Key = your provided authentication key
- Operation = GetEntities
- StreamToken = The alphanumeric streamtoken returned from the previous request

UpdateEntity

Allows for modification of attributes of existing Entities.

Post params:

- Key = your provided authentication key
- Operation = UpdateEntity
- Entity = Type of entity being modified (ex: Contact)

- Guid = guid of entity being modified
- AttributeValues = Comma separated attribute=value pairs.
- StringDelimiter (optional) = User-specified delimiter used to wrap string values (default is #)

The AttributeValue parameter syntax

A comma delimited list of attribute value pairs like:

FirstName=#Joe#,Birthday=1980-12-31,EmailVerified=true,NumChildren=3

Strings must be wrapped with a delimiter. The default delimiter is '#' but a user-specified delimiter can be utilized by including the optional StringDelimiter param.

OptionSets must be set to a numeric value valid for that optionset, or zero to clear current existing value. To determine values that are valid for an optionset, use the GetOptionSet api call.

Note that when updating an attribute of type EntityReferene, usually only the guid needs to be specified. In cases where the EntityReference can be of more than one type (ex: OwnerId can be SystemUser or Team) the value must be type:guid (SystemUser:fb50333e-6b9f-e111-8d5d-00155d000140).

In instances where the string delimiter occurs in the string itself (ex: 'Some#Text'), the delimiter must be Base64Encoded (ex: 'SomeIw==Text'). Of course the option to change the delimiter using the StringDelimiter parameter exists as well.

CreateEntity

Allows for creation of a new entity record.

Post params:

- Key = your provided authentication key
- Operation = CreateEntity
- Entity = Type of entity being modified (ex: Contact)
- AttributeValues = Comma separated attribute=value pairs.
- StringDelimiter (optional) = User-specified delimiter used to wrap string values (default is #)

The AttributeValue parameter syntax

The AttributeValue parameter for CreateEntity follows the same format as that for UpdateEntity.

PrimaryIdAttribute

Every entity type has a PrimaryIdAttribute which can be determined with a GetEntityMetadata call. If a specific guid is desired for the entity being created, it can be added to the AttributeValues parameter along with the other specified values. If the PrimaryIdAttribute is not specified, one will be automatically generated.

Example creating a Contact record (ContactId is PrimaryIdAttribute):

FirstName=#John#,LastName=#Doe#,ContactId= 84a4d17f-2e48-4a4b-bb28-26f7c14fc926, Birthday=1980-12-31

DeleteEntity

Allows for the deletion of existing entity records.

Post params:

- Key = your provided authentication key
- Operation = DeleteEntity
- Entity = Type of entity being queried (ex: Contact)
- Guid = guid of entity to delete

Valid Reponse Codes

Code	Description:Short	Desciption:Verbose
200	OK	The request was successfully processed and data is included in the response.
204	OK: No Data	The request was successfully processed but no data is included in the response. This is typical of UpdateEntity requests.
206	OK: Partial Data	The request was successfully processed and partial data is included in the response. This is the expected response when the dataset that Ramco needs to return to the user is too large. A StreamToken will be returned to allow the user to fetch the remaining data.
400	Bad Request	The request was not understood. See the response text for more information.
401	Unauthorized	The request was understood but it will not be fulfilled due to a lack of user permissions. See the response text for more information.
404	Not Found	The request is understood but no matching data is found to return.
500	Server Error	Something isn't working correctly server-side. This is not an issue that can be resolved by modifying query syntax.

Rapattoni Corporation

Authentication:

Magic API access is secured via OAuth2; a transaction with our identity server is required to obtain the necessary bearer token. Both Client Credential and Password grant types are supported.

For Client Credential Grant access, the following parameters must be passed in the body of a POST transaction:

- **grant_type**: Client Credential access will always use the client_credentials grant type.
- **client_id**: An id provided by the Association staff.
- **client_secret**: A value provided by the Association staff, associated with the client_id.
- **scope**: Provided by the Association staff. This must match a scope that the API recognizes and that is approved for usage by your credentials. It is part of what defines what you can access in the API.

For Password Grant access, the following parameters must be passed in the body of a POST transaction:

- **username**: A username provided by the Association, often an id known to the end user.
- **password**: The password associated with the above username.
- **grant_type**: Password access will always use the password grant type.
- **client_id**: An id provided by the Association staff.
- **client_secret**: A value provided by the Association staff, associated with the client_id.
- **scope**: Provided by the Association staff. This must match a scope that the API recognizes and that is approved for usage by your credentials. It is part of what defines what you can access in the API.

If the transaction is successful, then the response body will be a JSON object containing the access token, the token expiration date, and the token type. The token type will always be "bearer", and the time till the token expires is set by the association, on a client by client basis, based upon their rules and regulations.

API Transactions:

All transactions must include the Authorization header with the value being the client's current bearer token obtained from the transaction with our identity server.

The Magic API utilizes standard OData URIs and commands (ODatav3).

Responses, unless otherwise specified, will be in the JSON format.

The **Service Document** is available via a get request against the root URL (http://redacted/odata). By default, this will return in the XML format but can be requested in JSON by passing "application/json" in the Accept header. This document is a standard OData resource listing all top-level entity sets exposed by the service. (ODatav3 10.1.1)

The **Metadata Document** is available via a get request against the root URL with "/$metadata" appended (i.e. http://redacted/odata/$metadata). This resource describes the API's data model; including data types, relationships between entity sets, and available fields (ODatav3 10.1.2). This resource is only available in the XML format.

Entity Set Queries: All Entity Set queries use the default OData format; the root URL is appended with "/*EntitySet*" (i.e. to query the Offices Entity Set: http://redacted/odata/Offices)

When Querying against Entity Sets, the Magic API Supports most OData 3 query options, functions and operators. The below list are those query options and some of their associated supported functions and operators. Please see the OData 3 specification for further information in regards to these:

1) **$expand**: allows for the inclusion of entities related to the base entity set queried. The relationships between entity sets can be found in the metadata document. Multiple entities can be included with comma delimiting. If the relationship is further than 1 degree of separation than the full path to the targeted entity must be included. (ODatav3 10.2.3.1.3)
   Examples:
   a) **Members?$expand=MemberDesignations** : this will return Member records and their related MemberDesignations entries. The Metadata will show Members and MemberDesignations are linked via the Member_Number key.
   b) **Members?$expand=MemberDesignations/Designations** : this will return Member records, their related MemberDesignations entries and the related information to describe the Designations. The Members set links to MemberDesignations via the Member_Number key and the MemberDesignations set links to Designations on the Designation key.

2) **$select**: specify fields to be returned, if not included all fields will be returned. For selecting fields from expanded entity set relationships, the syntax is *EntitySet/fieldname*. (ODatav3 10.2.3.2)
   Examples:
   a) Members?$select=Member_Number,First_Name
   b) Members?$expand=MemberDesignations/Designations&$select=Member_Number,First_Name ,MemberDesignations/Designation

3) **$top**: specify the number of entities to return (ODatav3 10.2.3.4)
   a) Members?$top=10

4) **$skip**: skip over the first n entities prior to returning data (ODatav3 10.2.3.5)
   Example:
   a) Members?$skip=10

5) **$orderby**: order the results by a specific field's value; use **asc** or **desc** to specify ascending or descending. To order by multiple criteria use comma delimiting(ODatav3 10.2.3.3)
   Examples:
   a) Members?$orderby=First_Name asc
   b) Members?$orderby=First_Name asc, Member_Office_Name desc

6) **$inlinecount**: $inlinecount=allpages specifies that the total count of entities matching the request must be returned along with the results (ODatav3 10.2.3.6)
   Example:
   a) Members?$inlinecount=allpages
7) **$filter**: restrict the returned entities to only those matching the filter criteria (ODatav3 10.2.3.1)
   a) Operators
      i) **eq**: Equal
      ii) **ne**: Not equal
      iii) **gt**: Greater than
      iv) **ge**: Greater than or equal
      v) **lt**: Less than
      vi) **le**: Less than or equal
      vii) **and**: Logical and
      viii) **or**: Logical or
      ix) **not**: Logical negation, used in conjunction with other functions and operators to create a negative comparison.
      x) **add**: add to modify a numeric field and utilize the result for comparisons.
      xi) **sub**: subtract to modify a numeric field and utilize the result for comparisons.
      xii) **mul**: multiply to modify a numeric field and utilize the result for comparisons.
      xiii) **div**: divide to modify a numeric field and utilize the result for comparisons.
      xiv) **()**: Functions and operators can be nested using parentheses for precedence.
   b) Functions
      i) **endswith**: string ends with, syntax is: endswith(FieldName,'string')
      ii) **startswith**: string starts with, syntax is: startswith(FieldName,'string')
      iii) **substringof**: string contains, syntax is: substringof('string',FieldName)
      iv) **tolower**: string to lower casing
      v) **toupper**: string to upper casing
      vi) **trim**: trim all leading and trailing white spaces from a string
      vii) **concat**: append one string value to another
      viii) **day**: return the day component from a DateTime or Date
      ix) **hour**: return the hour component from a DateTime
      x) **minute**: return the minute component from a DateTime
      xi) **second**: return the second component from a DateTime
      xii) **month**: return the month component from a DateTime or Date
      xiii) **year**: return the year component from a DateTime or Date

Complex Filter Example:

Offices?$filter=(endswith(Office_Name,'Realty') and Street_City eq 'Ventura') or Salespersons add 5 gt 10

&: Multiple query options can be included at once, each separated by an "&"
Example: Members?$expand=MemberDesignations**&**$top=50**&**$select=Member_Number, MemberDesignations/Designation

odata.nextLink: In some cases, the number of records matching the query parameters is greater than the number of results the server is capable of returning. If so, the results will include an "odata.nextLink" property that contains the exact URL and query needed to obtain the next set of matching results.

Entity Set queries will by default return in JSON, but they can be obtained in XML by passing "application/atom+xml" in the Accept header.

Single Entity: To return one specific Entity, the format is *EntitySet*(*key*). For example, if pulling back member number 100 from the Members entity set, the root URL would be appended with /Members(100).

HTTP Response Codes

Identity Server:

Code	Text	Description
200	OK	Success
400	Bad Request "error: unsupported_grant_type"	The grant type entered during Authentication is not supported.
400	Bad Request "error: invalid_grant"	The username or password included were invalid
400	Bad Request "error: invalid_client"	The client or client secret included were invalid.
500	Internal Server Error	Most likely occurs due to an issue on the server side.

Magic API:

Code	**Text**	**Description**
200	OK	Success
400	Bad Request “The query parameter 'x' is not supported.”	The given query parameter is not valid.
400	Bad Request "Could not find a property named '*x*' on type '*entityset*'."	The given property doesn’t exist in the entity set being queried.
401	Unauthorized “Authorization has been denied for this request.”	The bearer token passed is expired or invalid.
404	Not Found	An invalid resource was targeted
500	Internal Server Error	Most likely occurs due to an issue on the server side.

About Clareity Consulting

Clareity Consulting brings clients fresh insights and wide perspective gained by serving clients throughout the industry: associations and MLSs, brokerages, franchises, technology vendors, and others. Clareity's services include:

STRATEGIC AND BUSINESS PLANNING

Clareity provides strategic, governance, and product/service business planning that bridges the gaps between strategy, tactics, and the timely activities needed to support your goals. Clareity also facilitates MLS regionalization and data shares.

PUBLIC SPEAKING

Clareity can address leadership or large groups on timely topics in an informative and fun way. Popular topics include MLS trends and system options, information security, and real estate technology trends, such as cloud computing and mobile technologies.

PRODUCT / SERVICE / SOFTWARE REVIEW

Clareity performs customer surveys and market research, develops product strategies and specifications, performs usability and quality assurance, audits security, and facilitates user groups. Clareity also facilitates strategic alliances, mergers, and acquisitions.

SYSTEM SELECTION

From needs assessment and RFP to contract negotiation, for brokerage management systems, transaction management, CRM, and other offerings, Clareity's structured processes help your organization make a good business decision with stakeholder involvement.

SECURITY AUDIT

Clareity is the most experienced security auditing company in the real estate industry, assessing security posture as well as helping organizations put an ongoing organizational security program in place, including both technical and non-technical best practices.

COMPLIANCE AND RISK AUDITS

Providing risk management and business resumption planning, staffing and salary reviews, and VOW / IDX compliance audits, Clareity brings both an independent view and finely-honed technical skills.

RECRUITING

Your business is only as successful as your leaders and employees, and Clareity has discreetly helped recruit some of the brightest minds in our industry for their current positions, both executive and technical.

EXPERT WITNESS

Whether it's a matter of the policies and practice of organized real estate or a more technical software dispute, Clareity can provide an expert witness with integrity and experience to conduct research, write expert opinions, and provide depositions and testimony.

For more information, please contact:

Gregg Larson
602-315-3362
Gregg.Larson@clareity.com

Matt Cohen
612-331-1788
Matt.Cohen@clareity.com

Or visit: http://clareity.com