oneilWebAPI Introduction

January 10, 2017

oneilWebAPI

This document attempts to describe basic oneilWebAPI use and conventions. This is not formal documentation.

The oneilWebAPI is based on REST concepts, so the URL templates, the HTTP request/response, and the JSON format should be somewhat familiar. The oneilWebAPI conventions are defined, and we will elaborate these now. This is an API in progress, and hence this is a document in progress; things could change.

URL, Version, Resource, and Chunk

To help ensure client/server stability across releases each oneilWebAPI URL will require a "version" component, which can basically be considered as part of the "name" of resource. We start at version 1, and if some breaking change needed to be introduced, using a different versioned endpoint might be the best option. The "resource" can be thought of as something to be accessed or manipulated, or as a general category of functionality. For example, the API will expose Account, Item, Web Order, and Web Order Detail resources among others. A simple example of the URL to access a version 1 "Record Center" resource and its "general information" data is shown below.

`https://a.site.com/oneilWebAPI/v1/recordcenter/info`

The URL alone is not enough to essentially use the API; you could not paste this into your browser and expect to see Record Center information pop up. (A proper HTTP GET with proprietary headers carrying credentials would be needed.) We will get to actual use next, but now we should look closer at the URL above.

- HTTPS is required by oneilWebAPI, and the server will not process requests coming in over HTTP. This makes staging and testing a little more formal in that certificates must be in place even on development machines, and HTTPS must function between client and server
- The "//a.site.com" is the URL to a fictitious development machine. There is a certificate installed on that machine for "*.site.com".
- The "/oneilWebAPI" is the site where I chose to host the oneilWebAPI application and expose it on the development machine. This name is arbitrary, and the oneilWebAPI could certainly be installed at the very "top level", the site level, or anywhere down some hierarchy; this is simply a web server configuration detail. As long as clients understand the path to the service, oneilWebAPI does not care.
- The "/v1" directs us to version 1 of the resource. There is only one version today.
- The "/recordcenter" represents the Record Center resource, which is a catch-all resource for many attributes that could exist for a Record Center.
- The "/info" designates the specific call, or action, or whatever, on the RecordCenter controller; in this case, the API returns general name, address, and phone data about the Record Center as a JSON data in the response body.
- This "/v1/recordcenter/info" URL will only accept the HTTP GET verb; the HTTP verbs are a key dimension to the API and will always be included in our descriptions of use.

Some API URL cases allow larger data sets to be retrieved as “chunks” of a specified chunk size, the “count”. In such calls, both the number-per-chunk “count” and chunk “instance” are specified on the URL. This will be described in more detail later. For now, imagine that you could retrieve 1000 Accounts in chunks of 50 Accounts each; chunk “count” would be 50, and chunk “instance” would begin at 1, the next request would be identical but the “instance” would then be 2, and so on until all Account records had been obtained. The HTTP response would contain special headers indicating the count/chunk parameters passed, plus headers indicating the total number of records in the total un-chunked data set, and the total number of records returned in this chunk response. There are more details, but this is enough for now.

https://a.site.com/oneilWebAPI/v1/Accounts/all?count=50&chunk=1 (first 50 records)
https://a.site.com/oneilWebAPI/v1/Accounts/all?count=50&chunk=2 (next 50 records)
https://a.site.com/oneilWebAPI/v1/Accounts/all?count=50&chunk=3 (next 50 records)

Hopefully that breakdown of the URL template provides enough of an introduction so that we can examine the simplest actual request that can be made.

API Request Basics

Almost every API URL requires credentials to be included in HTTP headers. (Of course, unless validated, only an HTTP 401 “Unauthorized” result will ever be returned.) That said, there are a few URLs that are allowed prior to authentication, basically available to all callers, and might be needed as a bootstrap step for clients. Our very first sample request will use one of these; the API Version number request. The “/v1/recordcenter/apiversion” URL will be supported from epoch to eternity.

https://a.site.com/oneilWebAPI/v1/recordcenter/apiversion

The complete HTTP request and the server response are below.

```
GET https://a.site.com/oneilWebAPI/v1/recordcenter/apiversion  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Fri, 12 Feb 2016 19:52:59 GMT
Content-Length: 16

{"apiVersion":1}
```

The GET request is simple, and the response is what a typical HTTP response might look like. The response body is a JSON representation of an “apiVersion” value of 1. (This is one of very few API calls that can be made directly by pasting the URL into a browser; it can be useful for rudimentary server connectivity/configuration troubleshooting from a browser.)

This next example is more typical, and requires credentials in order to complete successfully. The required credentials for oneilWebAPI consist of a Record Center ID, a User, and a Password. The ID uniquely identifies an

RS-SQL Record Center. (It also determines which Record Center to serve when multiples are hosted using a single oneilWebAPI.) All credentials are passed using HTTP headers; X-RS-SQL-RCID, X-RS-SQL-User, and X-RS-SQL-Password. An optional fourth header, X-RS-SQL-CallerData, is captured for future use when identification of API callers might be useful to clients and custom API implementations. The HTTP credential headers are described below.

X-RS-SQL-RCID	The Record Center GUID. This would be provided by client by the Record Center.
X-RS-SQL-User	The RS-SQL User Code. This is like the logon name.
X-RS-SQL-Password	This is the password. (This is one reason we insist on HTTS/SSL.)
X-RS-SQL-CallerData	This is for future use, and represents data specific to the "caller" that is not necessarily tied to the User. One example might by some device identifier. This may come in handy when an custom API plug-in is written.

We will retrieve basic Record Center information. The Record Center administrators have provided us with their base URL, their RC GUID, and our user name and password.

```
GET https://a.site.com/oneilWebAPI/v1/recordcenter/info  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Fri, 12 Feb 2016 21:35:04 GMT
Content-Length: 244

{
    "Name": "James Records Services",
    "AddressLine1": "5120 Rover Avenue",
    "AddressLine2": "Fontana, CA 93707-2131",
    "AddressLine3": "",
    "Phone": "312-363-1876",
    "Fax": "312-363-1874",
    "Email": null,
    "ReceiptText1": "Store It All",
    "ReceiptText2": "And Be Secure"
}
```

The GET request is simple, and the response is what a typical HTTP response might look like. The response body is a JSON representation of an anonymous data class containing the Record Center information. Had the User or Password been incorrect, an HTTP "401 Unauthorized" response would be returned. Had the Record Center ID been invalid, or one of the required X-RSSQL-* HTTP headers been missing, an HTTP "400 Bad Request" would be returned, and may have details in the response body. Below, the X-RSSQL-RCID header was misspelled as Z-RZSQL-RZID in the request, and a clue may be found in the response.

```
HTTP/1.1 400 Bad Request
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/7.5
Date: Fri, 12 Feb 2016 22:04:08 GMT
Content-Length: 365

{
    "Message": "The request is invalid.",
    "ModelState": {
        "$type": "System.Web.Http.HttpError, System.Web.Http",
        "A required HTTP Header tag is missing:": {
            "$type": "System.String[], mscorlib",
            "$values": ["X-RS-SQL-RCID"]
        }
    }
}
```

Often, in error and exception cases, the response body will contain JSON with “Message” and “ModelState” entries in which details are provided. The nature of the response in an error condition can vary, but will often contain useful details.

Resource by Identity Requests

Now that some API basics have been shown, we will show some typical requests and response that reveal some of the conventions used by the oneilWebAPI. The “Resource by Identity” pattern can be thought of as a simple lookup based on an ID value, and we’ll explore this now.

RS-SQL represents things such as Items, Accounts, Orders, and Actions within the RS-SQL database. Many things in the database are referenced, at least internally, by a unique identity value generated by the SQL Server engine. It is fast and easy to retrieve a single record if the identity value is known, so the oneilWebAPI exposes retrieval by identity whenever possible. The next example shows retrieval of an Item, (in this case, a Container or Box), using the ItemID value of 453719, which we happened to know. The URL itself is shown below.

https://a.site.com/oneilWebAPI/v1/Items/453719

In this case, the “resource” is specified as “/Items”. In oneilWebAPI it is common to “get some resource using its identity” by specifying the resource followed by the identity value. In this case we specify “/453719”. Almost every oneilWebAPI resource can be retrieved using this technique. The complete HTTP request and the server response are below.

```
GET https://a.site.com/oneilWebAPI/v1/Items/453719  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
```

```
Date: Fri, 12 Feb 2016 22:43:04 GMT
Content-Length: 1079

{
    "ItemID": 453719,
    "LocationID": 67368,
    "ContainerItemID": null,
    "AccountID": 1325,
    "BaseObjectID": 2,
    "ObjectID": 2,
    "ItemSecurityID": 1,
    "CategoryID": null,
    "ContaineeObjectID": null,
    "ItemStatusID": 32,
    "ItemCode": "JHR001",
    "AlternateCode": "ALT001",
    "StatusDtime": "2016-02-09T08:50:07.45",
    "AddDtime": "2016-02-09T08:50:07.45",
    "ItemDesc": "JHR Test oneilBridge first edit",
    "UserDtime": "2011-01-20T00:00:00",
    "ItemSetName": null,
    "AccessCount": 0,
    "UpdtUserID": 1020,
    "UpdtDtime": "2016-02-12T09:13:51.517",
    "LocationCode": "00005",
    "ContainerItemCode": null,
    "AccountCode": "CRANSTON",
    "AccountName": "County of Cranston",
    "BaseObjectDesc": "CONTAINER",
    "ObjectDesc": "CONTAINER",
    "ItemSecurityDesc": "Minimum Security",
    "CategoryCode": null,
    "ContaineeObjectDesc": null,
    "ItemStatusDesc": "In",
}
```

To save space, the JSON representation of the Item record above is being abbreviated. Nevertheless, this does represent Item 453719 from the RS-SQL database. Note that final ten or so fields, starting with "LocationCode" are not part of the actual Item record. These are provided as a convenience by oneilWebAPI, and basically represent a simple lookup from some foreign key value. The "LocationCode" represents the textual "Code" associated with "LocationID" 67368. Similarly, the "Account Name" shows the name associated with "AccountID" 1325. Often, response records will include helpful extras like these that might obviate the need to perform more API calls to obtain common or helpful details.

If the identity value had not been found, an HTTP "404 Not Found" response would be returned. The same is true if Account or Item security did not permit access, even if that resource did exist. The oneilWebAPI enforces RS-SQL security policies regarding Account and Item access.

The next example will retrieve the Account information from the AccountID value, 1325, of the Item above. The technique is identical, specifying a "resource/identity" in the URL, but using "Accounts" this time.

```
GET https://a.site.com/oneilWebAPI/v1/Accounts/1325  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
```

```
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Fri, 12 Feb 2016 23:04:39 GMT
Content-Length: 1017

{
    "AccountID": 1325,
    "Level1AccountID": 1325,
    "Level2AccountID": null,
    "Level3AccountID": null,
    "Level2ParentAccountID": null,
    "Level3ParentAccountID": null,
    "AccountCode": "CRANSTON",
    "AccountName": "County of Cranston",
    "LeaseExpireDtime": null,
    "ForceAuthorization": false,
    "SalesmanCode": "JHR",
    "UpdtUserID": 1,
    "UpdtDtime": "2010-09-24T14:39:58.087",
    "ContentValidation": false,
    "UseParentRetention": false,
    "DefaultPriorityID": null,
    "PriorityRequired": true,
    "WOChargeID": 1,
    "WOPickupReceiveChargeID": 0,
    "AccountReference": null,
    "ThreeLevelAccountCode": "CRANSTON"
}
```

To save space, the JSON representation of the Account record above has been abbreviated, but the concept should be clear at this point. (I didn't paste all the text from the response.) Retrieval using a resource identity is fairly straightforward across the entire oneilWebAPI.

Below is a partial list of resource names that support retrieval of a single record using the identity value specified as part of the URL. Each of these follows the paradigm illustrated for Item and Account above. It is important to note that retrieval by identity is often not the best way to grab information from the oneilWebAPI; often, this method was implemented for completeness, not necessarily for convenience.

v1/Accounts/<identity>	An Account record
v1/Actions/<identity>	An Action record
v1/Addresses/<identity>	An Address record
v1/BarcodeLengths/<identity>	A Barcode Length record
v1/CartEntries/<identity>	A Cart Entry, (pre Web Order Detail), record
v1/ItemActivities/<identity>	An Item Activity record
v1/ItemAliases/<identity>	An Item Alias record
v1/ItemContents/<identity>	An Item Content record
v1/ItemDescriptions/<identity>	An Item Description record
v1/ItemUrgents/<identity>	An Urgent Notification Item record
v1/Locations/<identity>	A Location record
v1/Objects/<identity>	An Object record
v1/Securities/<identity>	An Item Security record
v1/UserFields/<identity>	A User Defined Field Name record
v1/WebOrders/<identity>	A Web Order (header) record
v1/WebOrderDetails/<identity>	A Web Order Detail record
v1/Workorder/<identity>	A Workorder record

Bulk Requests and Chunks

Retrieval by identity results in zero or one resource being returned, but oneilWebAPI also facilitates retrieval of sets of resources. A set of resources could include anywhere from just a few to upwards of thousands of records. The results of simple or complex queries, and requests for "all" records need to be managed by the client, perhaps for performance reasons, or perhaps for pagination tasks.

As mentioned earlier, oneilWebAPI recognizes "count" and "chunk" URL query parameters, and when sets of data are returned, these optional parameters can be used to manage the returned results. Consider the RS-SQL "Object" data, which enumerates and describes, (among other things), resources that can be stored in a warehouse. There might be hundreds or thousands of records, and in our first example we would like to retrieve all of them, in chunks of 5. (A count of 3 is artificially tiny, but helps keep this example short. A value of 50 or more seems more appropriate, but the actual value to use can depend on many factors.)

The client might first make the following request to the "v1/objects/all" URL to request the very first chunk of 3 RS-SQL "Object" results. Note the "?count=3&chunk=1" in the URL.

```
GET https://a.site.com/oneilWebAPI/v1/Objects/all?count=3&chunk=1  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
X-RS-SQL-ResultCount: 3
X-RS-SQL-ResultTotal: 47
X-RS-SQL-RequestedCount: 3
X-RS-SQL-RequestedChunk: 1
Date: Mon, 15 Feb 2016 08:47:31 GMT
Content-Length: 803

[{
    "ObjectID": 1,
    "ParentObjectID": null,
    "ObjectCode": "CLIENT",
    "ObjectDesc": "A client",
    "ObjectTypeID": 1,
    "UpdtUserID": 1,
    "UpdtDtime": "2009-02-20T07:52:57.053",
    "ParentObjectDesc": null
},
{
    "ObjectID": 2,
    "ParentObjectID": null,
    "ObjectCode": "CONTAINER",
    "ObjectDesc": "CONTAINER",
    "ObjectTypeID": 2,
    "UpdtUserID": 1020,
    "UpdtDtime": "2015-05-27T17:16:52.653",
    "ParentObjectDesc": null
},
{
    "ObjectID": 3,
```

```
    "ParentObjectID": null,
    "ObjectCode": "FILEFOLDER",
    "ObjectDesc": "A file folder",
    "ObjectTypeID": 2,
    "UpdtUserID": 1,
    "UpdtDtime": "2009-02-20T09:21:40.813",
    "ParentObjectDesc": null
}]
```

We just requested of the "v1/Objects/all", the 3 records be returned in the results, and that those 3 were the very first "chunk of 3" among the total possible result set.

First, note that the JSON serialization represents a collection; our previous samples returned just a single serialized instance.

Next, note the four new response headers returned; these can be used by the caller as appropriate, and may possibly influence future requests. These are described below.

X-RS-SQL-ResultCount	This value represents the number of results returned by this call. Often, this will be identical to the "count" value specified in the URL query string. Typically, it is the very last "chunk" where an uneven multiple of "count" results in fewer than the requested "count" records being returned.
X-RS-SQL-ResultTotal	This value represents the total of results that exist. So, for the "v1/objects/all" request, 47 result records would have be returned had we not specified "count" and "chunk".
X-RS-SQL-RequestedCount	This value simply indicates the "count" value used in the original request.
X-RS-SQL-RequestedChunk	This value simply indicates the "chunk" value used in the original request.

Based on the value of the response headers, we see that to retrieve the 47 "Object" records in chunks of 3 records each, we would need to make 16 requests; the first 15 request should return 3 records each, with the very last request returning just 2 records.

Below, we can see the next logical request, in which we request the second chunk of 3 Object records. Only the "chunk" value, 2, differs from our first request; everything else is identical

```
GET https://a.site.com/oneilWebAPI/v1/Objects/all?count=3&chunk=2  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
X-RS-SQL-ResultCount: 3
X-RS-SQL-ResultTotal: 47
```

```
X-RS-SQL-RequestedCount: 3
X-RS-SQL-RequestedChunk: 2
Date: Mon, 15 Feb 2016 08:57:22 GMT
Content-Length: 832

[{
    "ObjectID": 4,
    "ParentObjectID": null,
    "ObjectCode": "LOCATION",
    "ObjectDesc": "A storage location",  "ObjectTypeID": 3,
    "UpdtUserID": 1,
    "UpdtDtime": "1999-06-15T09:39:17.547",
    "ParentObjectDesc": null
},
{
    "ObjectID": 5,
    "ParentObjectID": null,
    "ObjectCode": "MATERIAL",
    "ObjectDesc": "Things sold to a customer",
    "ObjectTypeID": 4,
    "UpdtUserID": 1,
    "UpdtDtime": "2009-02-20T09:21:48.3",
    "ParentObjectDesc": null
},
{
    "ObjectID": 6,
    "ParentObjectID": null,
    "ObjectCode": "USER",
    "ObjectDesc": "Users of the System",
    "ObjectTypeID": 5,
    "UpdtUserID": 1,
    "UpdtDtime": "2000-11-10T10:25:07.45",
    "ParentObjectDesc": null
}]
```

The response from this second request looks similar to the response from our first request. “Objects” with ObjectID value of 4, 5, and 6 were returned this time, whereas 1, 2, and 3 were returned the first time. Hopefully this brief example illustrates a simple case of how the “count” and “chunk” query parameters can be used to manage large result sets.

By convention, any oneilWebAPI URL that might return a “large” result set will support “count” and “chunk”.

Note that these new response headers are only returned if the optional “count/chunk” convention is used. If it is important that these four response headers are available to you, even if the results need not be “chunked”, specify the “count/chunk” parameters in the request URL. If you expect less than 100 results, perhaps specify “?count=500&chunk=1”; you will get all results, plus you will get the response headers.

Note that various IIS settings, (or possibly other firewall or gateway settings), might limit both request and response sizes. Please be aware that such settings might need to be factored in to decisions on the chunk count. Some mechanism external to oneilWebAPI might ultimately truncate or prohibit a response.

One final point to understand is that the oneilWebAPI is “stateless”, and regardless of “count” or “chunk” parameters, the complete results set must be manifested with every call. Within the oneilWebAPI service itself, once the total set of results has been obtained and ordered, the requested count/chunk can be assembled into a proper response. The implications of this implementation detail should always be under consideration; performance might be a factor, and result “ordering” is certainly a factor. The various “/v1/<resource>/all” APIs all sort based on identity value. When ad-hoc query selection/sorting becomes available, keep in mind that for the count/chunk mechanism to function predictably, all possible results must be consistently ordered such that multiple requests will always result in the same record order.

Below is a partial list of resource names that support the “all” style of, possibly chunked, retrieval. Every implementation of “all” semantics will support chunked results.

v1/Accounts/all	All Account records
v1/Actions/all	All Action records
v1/BarcodeLengths/all	All Barcode Length records
v1/ItemUrgents/all	All Urgent Notification Item records
v1/Objects/all	All Object records
v1/Securities/all	All Item Security records
v1/UserFields/all	All User Defined Field Name records

Simple Related Resources

Now that “Resource by Identity” requests and the count/chunk conventions have been introduced, a brief discussion of “related resource retrieval” seems appropriate. In RS-SQL, as in many business systems, “parent/child” or “master/detail” relationships exist among data elements. For example, in RS-SQL, “Account” records have two or more related “Address” records. Also, “Item” records have multiple related “Alias”, “ItemActivity”, “Contents”, and “Description” records. A “WebOrder” record typically has many related “WebOrderDetail” records.

To make retrieval of related data more convenient and intuitive, oneilWebAPI implements detail retrieval based upon a uniquely identified resource. (This same result could be achieved using a true-ad-hoc query feature in oneilWebAPI as well.) Consider an Account retrieved during an earlier example, “County of Cranston”, having an AccountID value of 1325; we could retrieve all available addresses using the URL below.

https://a.site.com/oneilWebAPI/v1/accounts/1325/addresses

The complete HTTP request and the server response are below. Note that the optional count/chunk convention is supported, but we estimate the number of related address records will be manageable, so it is not used in this example.

```
GET https://a.site.com/oneilWebAPI/v1/accounts/1325/addresses  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Mon, 16 Feb 2016 10:58:09 GMT
Content-Length: 760

[{
    "AddressID": 2731,
    "AccountID": 1325,
    "AddressDesc": "Default Mail Address",
    "AddressTypeID": 1,
```

```
    "Contact": "George Moore",
    "AddressLine1": "100 Civic Center Plaza",
    "AddressLine2": "Cranston CA, 99221",
    "AddressLine3": null,
    "Phone": "(818) 555-2233",
    "Fax": "(818) 555-2234",
    "Route": null,
    "UpdtUserID": 1,
    "UpdtDtime": "2010-09-23T11:08:55.59",
    "StopGroup": null,
    "ThreeLevelAccountCode": "CRANSTON"
},
{
    "AddressID": 2732,
    "AccountID": 1325,
    "AddressDesc": "Default Pick/Del Address",
    "AddressTypeID": 2,
    "Contact": "George Moore",
    "AddressLine1": "100 Civic Center Plaza",
    "AddressLine2": "Cranston CA, 99221",
    "AddressLine3": null,
    "Phone": "(818) 555-2233",
    "Fax": "(818) 555-2234",
    "Route": null,
    "UpdtUserID": 1,
    "UpdtDtime": "2010-09-23T11:08:55.627",
    "StopGroup": null,
    "ThreeLevelAccountCode": "CRANSTON"
}]
```


We see that the response body includes two Address records for this Account. We did not use optional "count/chunk" parameters, so all Addresses were returned.

Below is a partial list of resource names that can be helpful when retrieving related records within a parent/child or master/detail structure. Every implementation will support chunked results. The oneilWebAPI will be extended as appropriate to include as many of these types of convenience URL APIs as needed.

v1/Accounts/<identity>/subaccounts	L2 and L3 Sub-Account records for an Account
v1/Accounts/<identity>/parents	L1 and L2 Parent Account records for an Account
v1/Accounts/<identity>/addresses	Address records for an Account
v1/Actions/<identity>/actions	Child Action records for an Action
v1/Items/<identity>/aliases	Alias records for an Item
v1/Items/<identity>/contents	Content records for an Item
v1/Items/<identity>/descriptions	Description records for an Item
v1/Items/<identity>/keywords	Keyword records for an Item
v1/Items/<identity>/activities	Item Activity records for an Item
v1/Items/<identity>/items	Contained Item records for an Item
v1/Objects/<identity>/objects	Child Object records for an Object
v1/WebOrders/<identity>/details	Web Order Detail records for a Web Order
v1/WorkOrders/<identity>/itemactivities	Item Activity records for a Workorder
v1/WorkOrders/<identity>/itemactivities/access	Item Activity "Item Access" records
v1/WorkOrders/<identity>/itemactivities/delivery	Item Activity "Item Delivery" records
v1/WorkOrders/<identity>/itemactivities/receive	Item Activity "Item Receive" records
v1/WorkOrders/<identity>/itemactivities/pickup	Item Activity "Item Pickup" records
v1/WorkOrders/<identity>/itemactivities/destroy	Item Activity "Item Destroy" records
v1/WorkOrders/<identity>/miscbillings	Miscellaneous Billing records
v1/WorkOrders/<identity>/pickupreceivecounts	Unitemized counts for a Workorder

v1/WorkOrders/<identity>/signatures	Signature image data for a Workorder
v1/WorkOrders/<identity>/workordernotes	Note text lines for a Workorder

Note that an informed, accurate understanding of RS-SQL data representation is required to make good use of these URLs.

Data Transfer Objects

In the previous section, whether access data using an "identity", or accessing sets of data using "all", the data returned, its "shape" if fixed and not under your control. For each data type, be it a Barcode Length, and Account or an Item, the data structure is defined by us, within the oneilWebAPI server. Collectively, we refer to these structures as "Data Transfer Objects", which often abbreviate as DTO.

When you call "v1/accounts/<identity>" we serialize an "Account_DTO" instance and return it in the response body. For "v1/items/<identity>/items" we serialize a collection of "Item_DTO" instances and return the JSON collection in the response body. Since the oneilWebAPI does not presently allow you to request specific data elements, (such as in a "SELECT" statement), the DTO objects that we define within oneilWebAPI provide the structure for all the various data types returned.

Most of the DTO objects correlate closely with the underlying database table structures. To get some idea what an Item_DTO object would look like, the RSITEM table would be a good place to start. For Action_DTO, the RSACTION table my provide clues. Presently, the only "documentation" available for the various DTO classes exists in their source code, the .NET C# classes. These classes are designed to serialize nicely as JSON text. It is our hope that the various DTO data passed in response bodies will be self-explanatory for the most part.; the data can be used without necessarily fully understanding the DTO structure itself.

In the next section, which introduces Adding and Updating data, the DTO concept becomes much more important. An Item is a fairly robust concept in RS-SQL, so to add a new Item could require that you provide quite a bit of detail about that new Item; to do that, a JSON serialized Item_DTO instance will be required. At this point, a detailed definition of Item_DTO becomes necessary to properly use oneilWebAPI.

Updating an Item after it exists poses additional challenges; perhaps only one field/element is being updated. One way to accomplish this would be to request the Item, grab the entire Item_DTO instance, change the one field, say the AlternateCode, and pass the entire Item_DTO back in a new request to perform the update. In this case, you could imagine the update as a complete "replacement" of all data fields/elements, though all except AlternateCode would be "replaced" with their original values. This would suffice.

Additionally, we define the "Update Data Transfer Object", the UDTO. Often, the UDTO will closely follow the DTO in structure, except that every field/element will have a corresponding flag indicating whether that field should be updated or not. The "update flag" fields always default to false, and are typically named using a prepended "u"; for example, AlternateCode and uAlternateCode, AccountID and uAccountID. Using the previous example, we would send a request passing an Item_UDTO that included a valid identity for the Item to update, plus the AlternateCode member set to the new value, and with the uAlternateCode flag set true. With just this minimal/essential information, an Item update would succeed. Plus, with the nature of JSON serialization, only a partial Item_UDTO containing the essential elements would need to be passed; the server

will deserialize a complete Item_UDTO, and all “update flags” not explicitly passed would default to false, and only the AlternateCode would be updated.

Below is a sample of C# source code from the BarcodeLength_DTO class, which is one of the more trivial examples. When serialized as JSON text and returned in the HTTP response body, the order of the fields/elements may be ordered alphabetically.

```
[DataContract]
public class BarcodeLength_DTO
{
    [DataMember]
    public Int16        BarcodeLenID        { get; set; }
    [DataMember]
    public byte         Length              { get; set; }
    [DataMember]
    public String       BarcodeLenDesc      { get; set; }
    [DataMember]
    public Int16        ObjectID            { get; set; }
    [DataMember]
    public Int32        UpdtUserID          { get; set; }
    [DataMember]
    public DateTime     UpdtDtime           { get; set; }
    // Below are members included for client usability, and are not
    // part of the data as stored in the RSBARCODELEN table.
    [DataMember] [SupplementalDTO]
    public string       ObjectDesc          { get; set; }
}
```

Note the “ObjectDesc” property; properties marked as “[SupplementalDTO]” represent data not actually belonging to the entity, but rather a “lookup value” provided as a convenience for clients. This introduction to Data Transfer Objects is hardly complete, and rather fragmented, but should be enough that me may proceed with some detailed examples of adding and updating data.

Adding and Updating Data

Moving on from the various retrieval capabilities of oneilWebAPI, the important topics of insert, update, and delete must be introduced. Initially, oneilWebAPI insert/update/delete will be rather limited in scope; we expect new features will be implemented as required.

For any oneilWebAPI add/update, a full or partial JSON “UDTO” is passed in the request body. Remember that the UDTO convention ensures that the “update flag” defaults false; this is important when passing partial UDTO data. Often, only a partially specified UDTO will be sent during add/update, especially when adding or updating more complex records, such as Accounts or Items.

The “false” update flag value indicates “use a default value if available” for the “Add” case, and “retain current value” for the “Update” case. Of course, some fields are required when adding new records, and no default value would be appropriate.

A successful “Add” results in an HTTP “201 Created” response. In keeping with REST conventions, the complete URI to the newly created resource is placed in the HTTP Response Header “Location”. The identity value is also

returned in the response body as a simple JSON name/value pair. A successful “Update” will result in an HTTP “200 OK” response only.

The oneilWebAPI convention is defined so that HTTP verbs PUT and POST are both accepted for new resource creation; we treat them as interchangeable even though POST may be more technically appropriate. For updates, HTTP verbs PUT and PATCH are interchangeable, even though PATCH is emerging as the preferred verb.

In this first example, we will attempt to add a new Container with some minimal information about the Container. Imagine that a law firm has brought a box to the Record Center, and we placed a label, “JHR532”, onto the box, in preparation for storage. We know the customer AccountID value of 1325, and the customer tells us that box contains pleadings for the Jones case. To call the “v1/items/add” API, our PUT request must supply an Item_UDTO in the request body; not all members of the Item_UDTO need be supplied, but those that are supplied must include the value, say “Account”, and the indicator “uAccountID” to indicate that we are supplying a value and that a default value should not be used. The HTTP request below will add the new box to RS-SQL and return the new ItemID value.

```
PUT https://a.site.com/oneilWebAPI/v1/items/add  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 198

{
    "AccountID": 1325,
    "uAccountID": true,
    "BaseObjectID": 2,
    "uBaseObjectID": true,
    "ItemCode": "JHR532",
    "uItemCode": true,
    "ItemDesc": "Legal Pleadings Jones vs. Smith",
    "uItemDesc": true
}
```

```
HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: https://a.site.com/oneilWebAPI/v1/Items/453938
Server: Microsoft-IIS/7.5
Date: Thu, 25 Feb 2016 22:05:51 GMT
Content-Length: 17

{
    "ItemID":453938
}
```

This represents the bare minimum request necessary to add an Item. We chose to accept most Item defaults, but there are many more Item_UDTO data elements that might have been supplied. We had to supply AccountID so that RS-SQL would be aware of ownership and could enforce security appropriately. We had to supply the BaseObjectID to indicate “Container”, as opposed to “Filefolder”, (BaseObjectID := 3), or “Tape”. (BaseObjectID := 8). We chose to a short description of the Item, often a prudent thing to do. For all the many

Item_UDTO elements that we did not provide, the corresponding update flag value defaulted to false, and the API used default values.

You can see that the HTTP response header “Location” provides the full URI used to retrieve the newly created Item record, and the response body contains the identity value, the ItemID. Certain use cases require that the identity be captured and persisted.

If the Container just added contained Filefolders that were to be barcoded and tracked within RS-SQL, we could use the request below to add a new Filefolder, “FJHR500”, which is contained within our newly added Container. In this example, note that we need not supply the AccountID value since that can be obtained from the Container where this new Filefolder resides. Also note the use of the Item_DTO “ContainerItemID” element 453938, which indicates in which Container the Filefolder resides.

```
PUT https://a.site.com/oneilWebAPI/v1/items/add  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 199

{
    "ContainerItemID": 453938,
    "uContainerItemID": true,
    "BaseObjectID": 3,
    "uBaseObjectID": true,
    "ItemCode": "FJHR500",
    "uItemCode": true,
    "ItemDesc": "Pleading 1/7/2015",
    "uItemDesc": true
}
```

```
HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: https://a.site.com/oneilWebAPI/v1/Items/453939
Server: Microsoft-IIS/7.5
Date: Thu, 25 Feb 2016 22:15:51 GMT
Content-Length: 17

{
    "ItemID":453939
}
```

The response indicates the new Filefolder was successfully added, and it is now “contained in” the Container we added in the first example. These are very simple, minimal, examples designed primarily to illustrate how a PUT request should be formed, and how the UDTO structure is used. In truth, Item creation and the Item_UDTO structure is more complex, and its use more nuanced, essentially requiring a solid understand of RS-SQL itself and of the business standards and protocols established by the Record Center. These are topics well outside the scope of the API introduction.

As our next example, we will update the Container we added in the first example, and will change the Item Description and add text to a general purpose User Defined Field. We need to specify the ItemID value to the API for any update, though of course, the ItemID value is never updated.

```
PUT https://a.site.com/oneilWebAPI/v1/items/update  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 159

{
    "ItemID": 453938,
    "ItemDesc": "Pleadings of Jones vs. Smith plus many others",
    "uItemDesc": true,
    "UserField1": "Tagged for RFID",
    "uUserField1": true
}
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: https://a.site.com/oneilWebAPI/v1/Items/453939
Server: Microsoft-IIS/7.5
Date: Thu, 25 Feb 2016 22:43:19 GMT
Content-Length: 0
```

The HTTP “200 OK” response indicates success, and no other information is returned in the response body. Again, this is a trivial example, and more sophisticated updates will likely require detailed knowledge about RS-SQL and the business itself, possibly including “identity” values of other relevant RS-SQL entities.

The next simple example will add a Level2 Account, which represents a new “repossessions” department under the Level1 account representing a large lender. In preparation we have gathered the identity, 1325, of the company Level1 Account record. This new department operates out of a satellite office, so we want to provide accurate address and contact details. If we did not supply that information, values from the parent Account would be used by default.

```
PUT https://a.site.com/oneilWebAPI/v1/accounts/add  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 668

{
    "Level2ParentAccountID": 1325,
    "uLevel2ParentAccountID": "true",
    "AccountCode": "AHB10",
    "uAccountCode": "true",
    "AccountName": "Repossessions",
    "uAccountName": "true",

    "DeliveryContact": "Sheri Rollins",
```

```
    "uDeliveryContact": "true",
    "DeliveryAddressLine1": "114 Cowan Drive",
    "uDeliveryAddressLine1": "true",
    "DeliveryAddressLine2": "Irvine, CA  92618",
    "uDeliveryAddressLine2": "true",
    "DeliveryPhone": "949-555-7265",
    "uDeliveryPhone": "true",

    "MailContact": "Robert Smith",
    "uMailContact": "true",
    "MailAddressLine1": "P.O. Box 1122",
    "uMailAddressLine1": "true",
    "MailAddressLine2": "Stockton, CA  92004",
    "uMailAddressLine2": "true",
}
```

```
HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: https://dtsd7e64012.us.oneilsoft.com/oneilWebAPI/v1/Accounts/4361
Server: Microsoft-IIS/7.5
Date: Fri, 26 Feb 2016 16:40:59 GMT
Content-Length: 18

{
    "AccountID":4361
}
```

This is very similar to the new Item example. Much of this is almost intuitive, but knowing that the Level2ParentAccountID value is to contain that parent AccountID value might not be. In fact, a simple study of the Account_UDTO structure and all of its elements still might not provide the clue. In cases like this, experimentation in a non-production environment is a great way to learn.

API Catalog and Data Catalog

This document introduces oneilWebAPI, but this document is not comprehensive in scope. The oneilWebAPI itself can be called to return a rudimentary catalog of all supported URIs; the server literally looks back at itself and gathers all implemented URI controllers, their supported HTTP verbs, template URI parameters, and request body requirements. This feature was implemented to supplement any documentation, but we already find it quite useful. An example of its use is shown below, with partial output included.

```
GET https://a.site.com/oneilWebAPI/v1/recordcenter/apicatalog  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Thu, 3 Mar 2016 16:22:44 GMT
Content-Length: 30116
```

```
[{
    "verb": "GET",
    "template": "v{apiRequested:int:range(1, 1)}/Accounts/{accountID:int}",
    "method": "AccountsController.GetAccount(System.Int32 accountID)"
},
{
    "verb": "GET",
    "template": "v{apiRequested:int:range(1, 1)}/Accounts/all",
    "method": "AccountsController.GetAccountsAll()"
}
{
    "verb": "PUT,POST",
    "template": "v{apiRequested:int:range(1, 1)}/Accounts/Add",
    "method": "AccountsController.AddAccount([FromBody]oneilWebAPI.Models.Account_UDTO accountUDTO)"
},
{
    "verb": "PUT,PATCH",
    "template": "v{apiRequested:int:range(1, 1)}/Accounts/Update",
    "method": "AccountsController.UpdateAccount([FromBody]oneilWebAPI.Models.Account_UDTO accountUDTO)"
},
{
    "verb": "DELETE",
    "template": "v{apiRequested:int:range(1, 1)}/Accounts/{accountID:int}",
    "method": "AccountsController.DeleteAccount(System.Int32 accountID)"
}]
```

The response provides supported HTTP verbs, the relative URI template, and some detail on the actual controller implementation showing parameter data type and any expected HTTP request body data. The first entry above represents a “resource by identity” URI used to retrieve a single Account. We know from the “verb” member that this is a GET request; only GET is accepted. The “template” should be interpreted as “v1/accounts/123” where the integer 123 is the required AccountID value, internally named “accountID”. The “method” reveals that some internal AccountController.GetAccount() method is called, and the integer “accountID” from the template is passed in as the System.Int32 parameter. Your “developer intuition” will be helpful interpreting this catalog data.

Take a closer look at the third entry above. Here, we see that both PUT and POST verbs are supported; by convention, we know that some resource on the server is to be added or created. The “template” indicates we use “v1/accounts/add” as the relative URI, and no template parameters are used. The “method” is AccountController.AddAccount() and we can see from the parameters that a serialized Account_UDTO instance is expected in the request body. The “[FromBody] tag always indicates that the request body is required, and that its contents must be able to be deserialized into an instance of the stated type.

The oneilWebAPI also can return a catalog of all the various DTO/UDTO data types. Similar in intent to the “API Catalog” above, a short sample of the “API Data Catalog” output is below.

```
GET https://a.site.com/oneilWebAPI/v1/recordcenter/apidatacatalog  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
```

```
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
Date: Thu, 3 Mar 2016 16:25:10 GMT
Content-Length: 21693

[{
    "ClassName": "Account_UDTO",
    "PropertyList": [
        "AccountCode (String)",
        "AccountGroup (String)",
        "AccountID (Int32)",
        "AccountName (String)",
        "AdjustCharges (Boolean)",
        "CubicFeetStorage (Boolean)",
        "DefaultPriorityDesc (String) (Supplemental)",
        "DefaultPriorityID (Int16?)",
        "DeliveryAddressLine1 (String)",
        "DeliveryPhone (String)",
        "EnableConnector (Boolean)",
        "EnableUnitVolumeBilling (Boolean)",
        "Level1AccountID (Int32?)",
        "Level2AccountID (Int32?)",
        "Level2ParentAccountID (Int32?)",
        "Level3AccountID (Int32?)",
        "Level3ParentAccountID (Int32?)",
        "MailAddressLine1 (String)",
        "MailPhone (String)",
        "ThreeLevelAccountCode (String) (Supplemental)",
        "UpdtDtime (DateTime)",
        "UpdtUserID (Int32)" ]
},
{
    "ClassName": "BarcodeLength_DTO",
    "PropertyList": [
        "BarcodeLenDesc (String)",
        "BarcodeLenID (Int16)",
        "Length (Byte)",
        "ObjectDesc (String) (Supplemental)",
        "ObjectID (Int16)",
        "UpdtDtime (DateTime)",
        "UpdtUserID (Int32)" ]
}]
```

The response includes class name and a list of each property along with details about the required .NET data type. We can see that the Account_UDTO class has an integer, (a .NET Int32), AccountID and a string AccountName. The Level1AccountID .NET type is Int32?, where the "?" indicates that null values are allowed. All string values are technically nullable, though RS-SQL itself requires certain values to be supplied. The "(Supplemental)" label indicates a property that is simply included for client convenience; often, these represent a "lookup" of some kind. For example, and ObjectID will often be matched with ObjectDescription; the structure only actually contains ObjectID, the ObjectDescription is provided as a convenience on output, and is not required and is ignored on input.

Again, your "developer intuition" will be helpful interpreting this catalog data.

API Prohibit List

Since the oneilWebAPI is designed to be "public facing", the "surface area" for malicious exploitation should be manageable as appropriate for a particular implementation and deployment. We provide a means to disable or "prohibit" those oneilWebAPI URIs not being used. For example, if a site/application makes use of only several API URIs to gather basic RS-SQL information, that site may want to prohibit all attempts to access any DELETE

functionality. Certainly, credentials would be required, but having DELETE capabilities exposed to the world unnecessarily should be avoided if possible.

The oneilWebAPI "Prohibit List" feature makes use of Application Settings entries, essentially key/value pairs, within the web.config file. A basic template web.config file is installed with oneilWebAPI, and it is ready to be hand-edited as appropriate. The Prohibit List entries are read by oneilWebAPI and are enforced for every HTTP request. Prohibited URI requests receive a 400 "Bad Request" response, with details in the response body.

The "key" portion of an entry must conform to names like "prohibit_{0}", where "{0}" is a number from 1 to 99. There must be no gaps in the numbering; they are consecutive. The "value" is an HTTP verb/method, followed by a space, followed by text to be interpreted as a Regular Expression. The entry is essentially a pattern to match one or more URIs to prohibit when accessed using the specified verb/method. A couple examples should help.

```
<add key="prohibit_1" value="GET v1/workorders/.*/miscbillings" />
<add key="prohibit_2" value="PUT v1/accounts/add" />
<add key="prohibit_3" value="DELETE v1/" />
```

The first key will always be named "prohibit_1". The value will prohibit GET requests for any URI that the pattern matches. In this case, we prohibit Workorder Miscellaneous Billing details from being accessed, regardless of the Workorder identity. The second example prohibits new Accounts from being added. If the ability to add anything, (Accounts, Items, Web Orders, etc.), was to be prohibited, the value "PUT v1/.*/add" could be used instead. The third example prohibits all DELETE requests. A caller attempting to delete a record using oneilWebAPI would receive a 400 "Bad Request" response, and the body would indicate that the request has been rejected.

```
DELETE https://a.site.com/oneilWebAPI/v1/accounts/4358  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
```

```
HTTP/1.1 400 Bad Request
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/7.5
Date: Wed, 24 Feb 2016 22:04:22 GMT
Content-Length: 272

{
    "Message": "The request is invalid.",
    "ModelState": {
        "$type": "System.Web.Http.HttpError, System.Web.Http",
        "PROHIBITED": {
            "$type": "System.String[], mscorlib",
            "$values": ["An HTTP DELETE request targeting a prohibited URI,
                /oneilWebAPI/v1/accounts/4358, has been rejected."]
        }
    }
}
```

The response indicates that a DELETE request has been prohibited, and the request URI is provided. This information is also logged so that administrators can watch for unexpected activity. Note that he specific Prohibit List entry and expression is not provided, as a precaution against potential abuse.

To make effective use of the Prohibit List feature requires a clear understanding of the URIs that an application/deployment requires, plus an understanding of the oneilWebAPI and all the URIs it exposes. The regular expression patterns can be very flexible, but to be effective the oneilWebAPI URI conventions must be understood well. The feature is optional, of course, but exists to help reduce the exposed surface area of a deployment.

Finally, since oneilWebAPI can service multiple Record Centers, we added the ability to prohibit/disable an entire Record Center selectively. The oneilWebAPI "Prohibit RCID List" key/value entries specify Record Center GUID values to prohibit.

```
<add key="prohibitRCID_1" value="6c4a74bb-05ac-42e0-b419-6cfd82e95185" />
```

If only one Record Center needs oneilWebAPI support, the surface area should be constrained accordingly. This feature will only be of use at sites where more than one RS-SQL instance exists.

Rudimentary "UDTO" Query Capability

The oneilWebAPI data query/retrieval operations to this point have been rudimentary, retrieving "all" records, a single record based on its identity, or a set of related "detail" records relative to the identity of their parent. All of these are useful and simple to use. In contrast, a fully functional ad-hoc query capability, such as "dynamic query" discussed later, can be complicated and may require intimate understand of the RS-SQL database schema.

The oneilWebAPI implements a simple, possibly useful, query feature where the criteria are specified by providing an appropriate "UDTO" instance in the request body. Basically, a simple WHERE clause is being provided, and the "operation" applies to any/all UDTO members supplied. For many simple "value equality", "string contains", and "date/time is near" queries, the "UDTO" approach may be sufficient. The "UDTO" queries return appropriate "DTO" results only, and there is no SELECT, JOIN, or ORDERBY capability. Basic "AND" or "OR" logic can be applied across supplied values, but AND/OR cannot be mixed, matched, or grouped parenthetically.

The URIs used dictate the type of query performed. Only HTTP POST requests are supported, since the request body is required. Below, the Item query URIs are enumerated. Other URIs that follow the same pattern are shown in an abbreviated form, just to save some space. For "equals" queries, all field types are supported. For "contains" queries, only string fields are supported. For "circa" queries, only date/time fields are supported.

v1/Items/where/equal/and	Exact match of all Item values supplied
v1/Items/where/equal/or	Exact match of any Item value supplied
v1/Items/where/contains/and	Contains match of all Item *text* values supplied
v1/Items/where/contains/or	Contains match of any Item *text* value supplied
v1/Items/where/circa/<seconds>/and	Within seconds of all *date/time* values supplied
v1/Items/where/circa/<seconds>/or	Within seconds of any *date/time* value supplied

v1/Accounts/where/{equal,contains,circa/<seconds>}/{and,or}/
v1/WebOrders/where/{equal,contains,circa/<seconds>}/{and,or}/
v1/WebOrderDetails/where/{equal,contains,circa/<seconds>}/{and,or}/
v1/Workorders/where/{equal,contains,circa/<seconds>}/{and,or}/

Our first example will query for an Item based on Item Code. We will use the "where equal and" URI, which requires that all specified criteria match exactly; we only provide Item Code, so we could have used the "where equal or" URI to achieve the same result.

```
POST https://a.site.com/oneilWebAPI/v1/items/where/equal/and  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 38

{
    "ItemCode": "JHR001",
    "uItemCode": true
}
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/7.5
Date: Fri, 25 Mar 2016 12:05:01 GMT
Content-Length: 1081

{
    "ItemID": 453719,
    "LocationID": 67368,
    "ContainerItemID": null,
    "AccountID": 1325,
    "BaseObjectID": 2,
    "ObjectID": 2,
    "ItemSecurityID": 1,
    "CategoryID": null,
    "ContaineeObjectID": null,
    "ItemStatusID": 32,
    "ItemCode": "JHR001",
    "AlternateCode": "ALT001",
    "StatusDtime": "2016-02-09T08:50:07.45",
    "AddDtime": "2016-02-09T08:50:07.45",
    "ItemDesc": "JHR Test oneilBridge first edit",
    "UserDtime": "2011-01-20T00:00:00",
    Etc…
}
```

The HTTP "200 OK" response indicates success, and the response body contains the single Item_DTO for of the single-record result. Next, we show an example using the "where contains or" URI. Again, we will supply Item_UDTO data representing the criteria; all Items that contain "JHR" in the Alternate Code or "James Brown" in the Item Description. Only the first chunk of 25 records is to be returned, and I omit the actual results in the response body to save space.

```
POST https://a.site.com/oneilWebAPI/v1/items/where/contains/or?count=25&chunk=1  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
```

```
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 107

{
    "AlternateCode": "JHR",
    "uAlternateCode": true,
    "ItemDesc": "James Brown",
    "uItemDesc": true
}
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/7.5
X-RS-SQL-ResultCount: 25
X-RS-SQL-ResultTotal: 46
X-RS-SQL-RequestedCount: 25
X-RS-SQL-RequestedChunk: 1
Date: Fri, 25 Mar 2016 14:11:22 GMT
Content-Length: 1081

[{
    Etc. 1…
},
{
    Etc. N…
}]
```

We can see from the X-RS-SQL response headers that a total of 46 Item records are available, the first chunk of 25 Item records was returned to us. I tried this exact query but changed the “or” in the URI to “and”, and I received an HTTP “204 No Content”. As expected, the “and” query was more restrictive, and I had nothing in my database to satisfy that query. The next UDTO query example will illustrate the “circa” form, where date/time values must be within a specified number of seconds of the supplied values. I’ll keep this simple and request only Workorders created within a week, (604800 seconds), of 1/7/2016.

```
POST https://a.site.com/oneilWebAPI/v1/workorders/where/circa/604800/and?count=25&chunk=1  HTTP/1.1
User-Agent: Fiddler
Host: a.site.com
Accept-Encoding: gzip
X-RS-SQL-RCID: 6c4a74bb-05ac-42e0-b419-6cfd82e95185
X-RS-SQL-User: jhr
X-RS-SQL-Password: jhrP@ssw0rd
X-RS-SQL-CallerData: LinqPad-987654321|1|4
Content-Type: application/json
Content-Length: 107

{
    "CreateDtime": "2016-01-07",
    "uCreateDtime": true
}
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/7.5
X-RS-SQL-ResultCount: 2
```

```
X-RS-SQL-ResultTotal: 2
X-RS-SQL-RequestedCount: 25
X-RS-SQL-RequestedChunk: 1
Date: Fri, 25 Mar 2016 14:52:45 GMT
Content-Length: 712

[{
    Etc. 1…
},
{
    Etc. N…
}]
```

Only two Workorder records were returned, so it must have been a slow week. I should note that the query includes 7 days before and after the specified 1/7/2016 date. I hope that the simple capabilities of the UDTO queries have at least been introduced. Ultimately, there can be no substitute for experimentation with an actual known data set.

The Dynamic Query Capability

This section may seem as somewhat an afterthought, but the "dynamic" query capability is important to at least be aware of. Where the oneilWebAPI retrieval allows, at best, a WHERE clause to be specified, the "dynamic" capabilities allow SELECT, WHERE, and ORDER BY, including the ability to JOIN out to related entities to include those attributes in the SELECT, WHERE, and ORDER BY. The SQL concept of DISTINCT is supported as well.

For any "dynamic" query executed, a SHA256 checksum of the result can be requested instead of an actual result set. This feature may be useful when the client retains a previous checksum result, and wishes to see if the query results have changed, (hence should be retrieved), by comparing the saved/current hash values.

Almost any valid .NET LINQ-to-SQL expression tree could be serialized by the client and processed by the server. The "count/chunk" convention applies to all "dynamic" queries, and if they are used, an ORDER BY is required, of course. Though not presently documented for general use in the oneilWebAPI, the "dynamic" feature could provide power and flexibility, under client control, for special and unforeseen application needs.

We retain "dynamic" capabilities not so that integrators can use them at will, but so that we, O'Neil Software, Inc., have this power within the API should we need to use it to help our integrators. If necessary, we could provide a JSON serialized .NET LINQ expression tree. The JSON text would be passed in a request body to a "dynamic" URI on, say "v1/Items/dynamic", allowing a quick and dirty custom query capability. The JSON text that we send to the integrator could have "placeholders" for one or more "parameters" that could be altered at run time prior to submission in the request body.

Future versions of the oneilWebAPI will support a similar but friendlier "dynamic" capability, named "ad-hoc", where the submission of SELECT, WHERE, and ORDER BY, (and perhaps even JOIN), are available for general use. The expressions to submit in the request body would be less proprietary, and easier to serialize.