OneRoster Dynamics 365 Provider Sample

# 1. Credit

Forked from the [Office 365 OneRoster REST Provider Sample](https://github.com/OfficeDev/O365-EDU-SDS-AspNetMVC-Samples)

# 2. Scope

There are two key components demonstrated by this sample, detailed below.

1. A provider that maps from a Dynamics 365 data model to the OneRoster v1.1 model (Dynamics 🡪 OneRoster). This makes the Dynamics 365 data available to potential consumers using a standardized data model. The following routes have been implemented for this sample:
   1. Schools
      1. /ims/oneroster/v1p1/schools
   2. Academic Sessions
      1. /ims/oneroster/v1p1/academicSessions
   3. Classes for a School
      1. /ims/oneroster/v1p1/{schoolSourcedId}/classes
   4. Students for a School
      1. /ims/oneroster/v1p1/{schoolSourcedId}/students
   5. Teachers for a School
      1. /ims/oneroster/v1p1/{schoolSourcedId}/teachers
   6. Student Enrollments for a School
      1. /ims/oneroster/v1p1/{schoolSourcedId}/enrollments?filter=role=’student’
   7. Teacher Enrollments for a School
      1. /ims/oneroster/v1p1/{schoolSourcedId}/enrollments?filter=role=’teacher’
2. A route (/sync) that triggers a pull of data from another OneRoster API, down into Dynamics 365 (OneRoster 🡪 Dynamics). The following data pulls have been implemented for this sample:
   1. Active Schools
   2. Academic Sessions
   3. Active Classes for a School
   4. Active Students for a School
   5. Active Teachers for a School
   6. Active Student Enrollments for a School
   7. Active Teacher Enrollments for a School

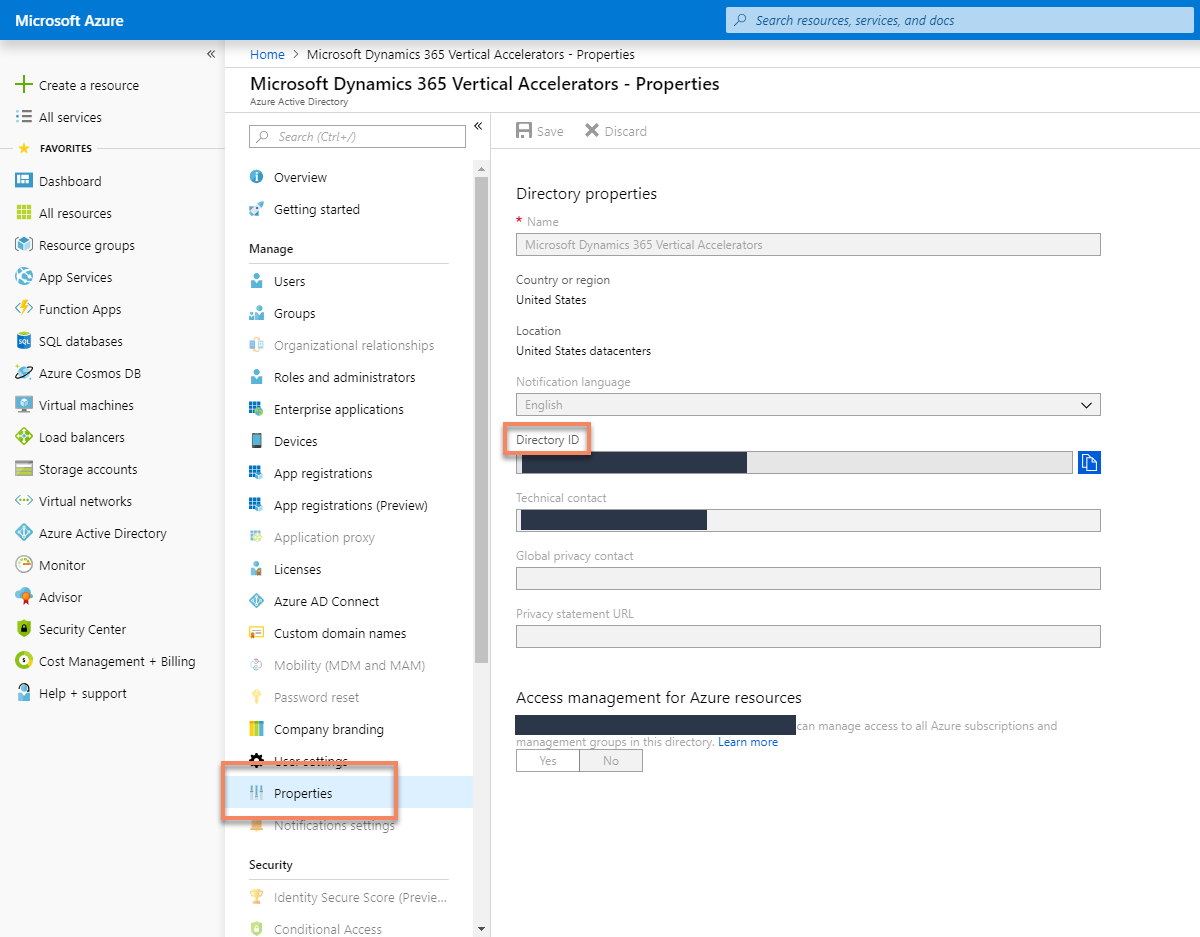
# 3. Setup

## 3.1 Pre-requisites

1. Installation of the Microsoft K-12 Education Accelerator.
2. Azure subscription.

## 3.2 Connection to Dynamics 365

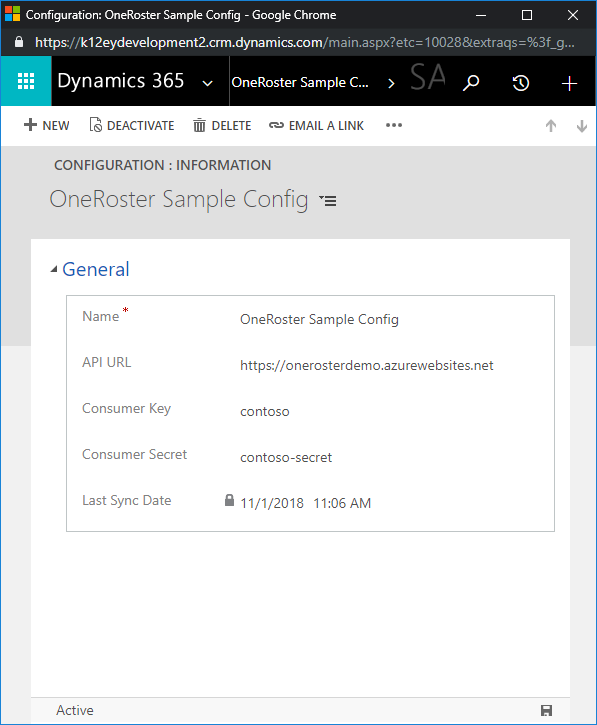
1. Create an Azure Active Directory (Azure AD) app.
   1. Sign in to the Azure management portal by using an account with administrator privileges. You must use an account in the same Office 365 subscription (tentnat) as you intend to register the app with.
   2. Follow these steps to create the app, copy the redirect URL for later use: <https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-v1-add-azure-ad-app#adding-an-application>
   3. Once the app is created in Azure Active Directory, a unique Application ID is generated. Click the app to open the app information page.
   4. On the app information page, hover over Applicaton ID, and select the Click to copy icon to copy the value for later use.
   5. Click Settings.
   6. In the API Access area, select Keys.
   7. Create a new key with the desired description and duration. This will be used for our ClientSecret later on, so copy the value down for later use.
      1. Note that the value will only be displayed once on save, so be sure to copy it down immediately after save.
2. Gather your Tenant ID from Azure AD.
   1. In the Azure management portal, navigate to Azure Active Directory.
   2. Select Properties.
   3. Copy the Directory ID down for later use. This is your Azure Tenant ID that will be used in the Authority configuration setting later.



1. Grant the Azure AD app permissions to access your Dynamics 365 data.
   1. Follow the steps here to create a user and assign a security role to the app: <https://docs.microsoft.com/en-us/dynamics365/customer-engagement/developer/use-multi-tenant-server-server-authentication#create-an-application-user--associated-with-the-registered-application--in->
2. Open the Visual Studio Solution.
3. Open **appsettings.json**.
4. Edit the **CrmConnection** section to have the following details:
   1. **Resource**: Root URL of your Dynamics instance (ex. <https://demo.crm.dynamics.com/>)
   2. **ApiUrl**: URL to the WebAPI endpoint of your Dynamics instance (ex. <https://demo.api.crm.dynamics.com/api/data/v9.0/>)
   3. **ClientId**: ID of your Azure AD app
   4. **ClientSecret**: Key/secret from your Azure AD app
   5. **RedirectUrl**: Redirect URL from your Azure AD app
   6. **Authority**: [https://login.microsoftonline.com/**{your\_azure\_tentatid}**/oauth2/token](https://login.microsoftonline.com/%7byour_azure_tentatid%7d/oauth2/token)

## 3.3 Configuration of Sync (OneRoster 🡪 Dynamics)

1. Login to your Dynamics 365 organization.
2. Open Advanced Find and look for “Configurations”.
3. Create a new Configuration record.
   1. **Name**: any identifying value you would like.
   2. **API URL**: URL to the target OneRoster endpoint.
   3. **Consumer Key**: OAuth 1.0a consumer key
   4. **Consumer Secret:** OAuth 1.0a consumer secret
   5. **Last Sync Date:** Leave this blank. This will be populated after the first successful sync.



1. **Save** the record.

# 4. Deployment

See the documentation for [Deploying the Office 365 OneRoster REST Provider Sample](https://github.com/OfficeDev/O365-EDU-SDS-AspNetMVC-Samples#deploy-the-sample-to-azure).

# 5. Considerations

## 5.1 Authentication

This sample uses OAuth 1.0a for authentication for both the provider and the sync route. The OneRoster v1.1 specification however also allows for a more modern and secure standard in that of OAuth 2.0. To implement, the general idea would be to leverage an existing library to do the heavy lifting like IdentityServer4. See this documentation for an example on one possible way to implement: <https://github.com/IdentityServer/IdentityServer4/blob/dev/docs/quickstarts/1_client_credentials.rst>.

# 6. Triggering Sync (OneRoster 🡪 Dynamics)

Once configuration is complete, in order to trigger the synchronization process all that is needed is to execute a simple GET request to the route (**/sync**).

# 7. Understand the code

## Introduction

This web application is based on an ASP.NET Core Web project template.

## Authorization

The Middlewares/OAuth.cs file defines a middleware that validates the OAuth 1 signature for each incoming request with a OneRoster route. This file also contains the hard-coded client ID and secret. In order to make a request, first visit the /token endpoint using OAuth1 credentials to get the access token. The Startup.cs file configures the app to use this middleware.

## Data Model

Most of the OneRoster models have a corresponding model class in the Models directory. Due to language naming conventions, some of these have been renamed. The mapping of models is shown in the table below.

|  |  |  |
| --- | --- | --- |
| **OneRoster Model** | **Class File** | **CRM Model** |
| Base | BaseModel.cs | N/A |
| Academic Session | AcademicSession.cs | msk12\_academicsession |
| Class | IMSClass.cs | msk12\_class |
| Course | Course.cs | msk12\_course |
| Demographic Data | Demographic.cs | Not implemented |
| Enrollment | Enrollment.cs | msk12\_enrollment |
| Line Item | LineItem.cs | Not implemented |
| Line Item Category | LineItemCategory.cs | Not implemented |
| Org | Org.cs | account |
| Resource | Resource.cs | Not implemented |
| Result | Result.cs | Not implemented |
| User, Student, Teacher | User.cs | contact |

Enumerations used to map between OneRoster values and the Dynamics equivalent value are provided by the Vocabulary/Vocabulary.cs file.

## Customized Serialization

All response JSON objects are written using the NewtonSoft JSON nuget package, for the purposes of matching the prescribed JSON bindings. There are two methods per model that handle this serialization, AsJson and AsJsonReference.

## Pagination, Filtering, and Sorting

The Controllers/BaseController.cs class file hooks in paging, filtering, and sorting.

Filtering and sorting are implemented in the model classes via reflection in order to suppor the requirement that any data field be usable for filtering or soring.

The actual execution of the pagination, filtering, and sorting is done through FetchXML against the Dynamics 365 Web API endpoint. The FetchXML is built inside the BaseController’s BuildFetch method.

All pagination, filtering, and sorting works according to the OneRoster spec. Note that nested property filtering and field selection are not supported in this sample however.

## Sync Process

As noted in 6. Triggering Sync, the process is initiated by issuing a GET request to /sync. This calls into the Controllers/DynamicsSyncController.cs file, which in turn queries Dynamics 365 for a configuration record (msk12\_configuration). This configuration record contains the host and protocol of the target OneRoster API to pull data from, as well as the consumer key, consumer secret, and the last sync date (which assists with synchronizing deltas).

Once a valid configuration record is retrieved, the controller instantiates an instance of the Helpers/DynamicsSyncAgent class, passing along the necessary configuration details. The Sync Agent authenticates to the target OneRoster API, then begins the actual sync process.

During the sync process, one page of 100 records is retrieved from the target OneRoster API, then that batch is saved to Dynamics before the next batch of records is requested.

The sync process first pulls all active schools modified since the last sync date, then all active sessions. Finally, for each school, the rest of the data for that school is synched.

Upon completion, the configuration record in Dynamics 365 is updated with the last sync date being set to now.