# Module 2 Lab C

# Creating and retrieving entity information using late bound types

Time 45 Minutes

##### Objective: Use the CrmServices client to retrieve, update and create records using late bound types.

**Step 1: Installing the metadata browser solution**

|  |  |
| --- | --- |
| **1** | Navigate to the settings area on Dynamics 365 and click on the solutions link  Import the solution by clicking on the import button and navigating to the tools area of the sdk where you will find the solution ( see below ) |
| **2** | You will be using the Metadata browser to help you later in this lab and other labs later on in this course. |

**Step 2: Add functionality to a windows application to allow the creation of new Account records**

|  |  |
| --- | --- |
| **1** | Open the Solution in the starter folder for Module 2 Lab C  References to the sdk and tooling connector assemblies have been added to this project already. (You may need to reference them again from their original locations)  You should use nuget to reference the package Microsoft.CrmSdk.XrmTooling.CoreAssembly  Modify the connection string in the app.config file so that it will be valid for your tenancy. Copying it from Module 2 LAB B START should work. |
| **2** | Open form1.cs and double click on the “Create Account” button to handle the click event for that button.  Within the cmdCreateAccount\_Click add code to instantiate an Entity class and assign it to a variable named account. Make sure that you set it’s type to “account” in the constructor. |
| **3** | Open up the metadatabrowser in Dynamics 365 and search for information on the attributes of the account entity. Look at the datatypes and requirement level for the name, address1\_city and address1\_line1 attributes.    As you can see the name attribute is required.  Set the name attribute and also the address1\_city and address1\_line1. The values will be supplied from textboxes on the form named txtName, txtAddressLine1 and txtCity. See next page for help |

|  |  |
| --- | --- |
| **4** | Set the name, adress1\_line1 and address1\_city attributes as shown below |
| **5** | Add code to the form constructor above the to set the default security protocol to TLs1.2 as you did in the previous lab. |
| **6** | The project has a file named OrgServiceUtils.cs with a function named createOrgServiceClient. It returns a crmServicesClient object configured with your connection setting in the app.config file. The method is declared as static so you will be able to call it as follows  var crmServiceClient = OrgServiceUtils.createOrgServiceClient(); |
| **7** | Create the CrmServiceClient proxy object and make a call to the create method passing in the account entity you have just initialised. After the call dispose of the proxy object. Your code should look like this. |
| **8** | Check that the code works by running the app and creating a new account. Using a browser navigate to the web site and look for the new account. |

**Step 2: Add functionality to a windows application retrieve multiple account records based on filtering by city**

.

|  |  |
| --- | --- |
| 1 | Search for the comment  //add code here to retrieve accounts filtered by city.  It should be within the btnGetAccounts\_Click method.  Place your cursor with this if statement and then follow the next set of instructions. |
| 2 | Create another instance of a CrmServicesClient using the utilities class as in the previous exercise  var crmServiceClient = OrgServiceUtils.createOrgServiceClient();  Below this declare a variable named cityCondition of type ConditionExpression assign to it a new instance of a ConditionExpression using the default constructor.  ConditionExpression cityCondition = new ConditionExpression();  We will use this to restrict the accounts retrieved based on city.  Set the Condition expression up so that the accounts retrieved will be filtered on the address1\_city attribute  cityCondition.AttributeName = "address1\_city";  Set condition operator us use a like comparison such that the cityname is **like** the value entered in the txtCitySearch textbox. See below for help.  cityCondition.Operator = ConditionOperator.Like;  cityCondition.Values.Add(txtCitySearch.Text + "%"); |
| 3 | Following on add a variable named cityFilter of type FilterExpression and initialise it to a new instance of FilterExpression  FilterExpression cityFilter = new FilterExpression();  Call the add method of the Conditions collection of the CityFilter object  passing the cityCondition we created earlier as a parameter  cityFilter.Conditions.Add(cityCondition); |
| 4 | We’re now ready to prepare an object of type QueryExpression.  Declare a variable named query of type QueryExpression and assign to it a new QueryExpression object passing the value “account” into the constructor.  QueryExpression query = new QueryExpression("account");  This query will be used to retrieve multiple account records filtered by city.  Set the ColumnsSet property of the query object so that only the name, address1\_city and accountid are returned.  query.ColumnSet.AddColumns("name", "address1\_city","accountid");  Now, using the query variable call the criteria Property’s AddFilter method passing the FilterExpression created previously as a parameter.  query.Criteria.AddFilter(cityFilter); |
| 5 | Modify the query object to set the ordering so that the results will be returned sorted by account name in ascending order.  query.Orders.Add(new OrderExpression("name", OrderType.Ascending));  Then pass the query object to the RetrieveMultiple method of the crmServicesClient and accessing the Entities property to a variable named matchedAccounts. See Below for help.  var matchedAccounts = crmServiceClient.RetrieveMultiple(query).Entities; |
| 6 | The completed code should look like this |
| 7 | The project contains as class named Account that we’ll be using to hold instances of the retuned records. The constructor to this class accepts the accountid, name and city.  Immediately below the line  Var matchedAccounts = crmServiceClient.RetrieveMultiple(query).Entities;  add a for each loop to add a new instance of an Account object initialised with the data returned from the query and add the object to the listbox names lstAccounts  foreach (Entity item in matchedAccounts)  {  var acc = new Account(new Guid(item["accountid"].ToString()), item["name"].ToString(), item["address1\_city"].ToString());  lstAccounts.Items.Add(acc);  } |
| 8 | Test your work by running the app and typing Seattle or just Sea into the search text box and then click the Get Accounts button. |

**Step 3: Retrieving and Adding related records. You will add a letter to and associate it to an existing account.**

|  |  |
| --- | --- |
| 1 | A second form will be used to display an account’s related letters and allow the user to add further letters with that account.  Handle the selectedIndexChanged event of the lstAccounts object using the visual studio designer by double clicking on the event in the list displayed in the properties window see below for help      This function is called when someone clicks on an item in the lstbox and we will then show a form that displays related letters.  The project has a second form that uses a query expression to return and display letters associated with an account. The accountId passed to the constructor will be used to perform the filtering. |
| 2 | Within the lstAccounts\_SelectedIndexChanged function declare a variable of type Account named selectedAccount and assign to it the lstAccounts.SelectedItem property as follows.  Account selectedAccount = lstAccounts.SelectedItem as Account;  Immediately below the line you have just added declare a variable named frmLetter of type FrmAddLetter and initialise it to a new instance of the FrmAddLetter class passing in the accounted property of the selectedAccount object.  FrmAddLetter frmLetter = new FrmAddLetter(selectedAccount.AccountId); |
| 3 | Now just display the form by calling the ShowDialog method of the frmLetter object. Your code should look like this. |
| 4 | Put a break point in the first line of code within the lstAccount\_SelectedIndexChanged function and run the project in debug mode.  Select an account from the list after performing a search and then step through the code using F11 once the break point has been hit.  Place your cursor over variables such as the accounted to see their values. Hit f5 to continue. The list of letters for an account will probably be empty at this stage unless you log on to the dynamics 365 portal and create one or two for an account. we will now add functionality to create letters from the client. |
| 5 | Double click on the FrmLetterAdd.cs file in the Soultion explorer window to open up the form in visual studio and then double click on the Add Letter button.  In the btnAddLetter\_Click function create another instance of the crmServicesClient object as we have done previously  var crmServiceClient = OrgServiceUtils.createOrgServiceClient(); |
| 6 | Use the metadata browser in to look up the datatypes and requirement level of the Subject and Activityid attributes of the Letter entity |
| 7 | Immediately following the crmServiceClient initialisation create a try catch finally block and within the try block create a new instance of a letter entity.  Assign to it’s subject attribute the value in the textbox named txtSubject and generate a new Guid and assign it to the activityid attribute. |
| 8 | Now use the crmServiceClient to create the new letter by passing the letter entity to the create method of the client. Make sure to store the id return by that method in a variable named letterId  Guid letterId = crmServiceClient.Create(letter); |
| 9 | Use the metadata browser to search for information related to the accounts one to many relationships and use the search to look for a relationship with “Letter” in the name.  We will now associate the newly created letter with the account the user selected from the list. First, using the accounted stored in the acid field retrieve the Account record using the retrieve method of the crmServicesClient object and assign it to a variable named selectedAccount.  var selectedAccount = crmServiceClient.Retrieve("account", accId, new ColumnSet("name")); |
| 10 | Now create a generic list of entities called letters. Add the letter entity to the list.  List<Entity> letters = new List<Entity>();  letters.Add(letter);  Now declare a Variable named letterRelationship of type RelationShip and initialise it to a new instance passing to the constructor the name of the relationship you obtained from the Metadata browser in step 6 above.  Relationship letterRelationship = new Relationship("Account\_Letters"); |
| 11 | Create a variable of type EntityCollection named relatedletters and assign to it a new instance of an EntityCollectionObject passing the letters variable in the constructor. |
| 12 | Following that call the Add method of the relatedEntities property of the selectAccount object passing to the first parameter the letterRelationship object and to the second parameter the related letters object. |
| 13 | Now use the crmServicesClient update method passing the selectedAccount  object as the parameter. |
| 14 | The completed code should look like this |
| 15 | Following the call to the Update method add a call to the function populateLettersList();  Modify the Catch block to catch exceptions of type  FaultException<OrganizationServiceFault>.  Display the Message property using MessageBox.Show(ex.Message)  In the finally block Dispose of the crmServicesClient |
| 16 | Finally run the App and test that you can add letters and associate them with the existing account. |

