

Part 9 - Calling Custom APIs from TypeScript

This is part of the course 'Scott's guide to building Power Apps JavaScript Web Resources using TypeScript'.

In this ninth part we will cover how to call a custom API that is build using C# and deployed via a Plugin, using `dataverse-ify` with `dataverse-gen`. The advantage of using `dataverse-ify` is that it will generate types for you so that you can easily provide the correct data and then execute the request, then read the response in a similar way to the `IOrganizationService`.

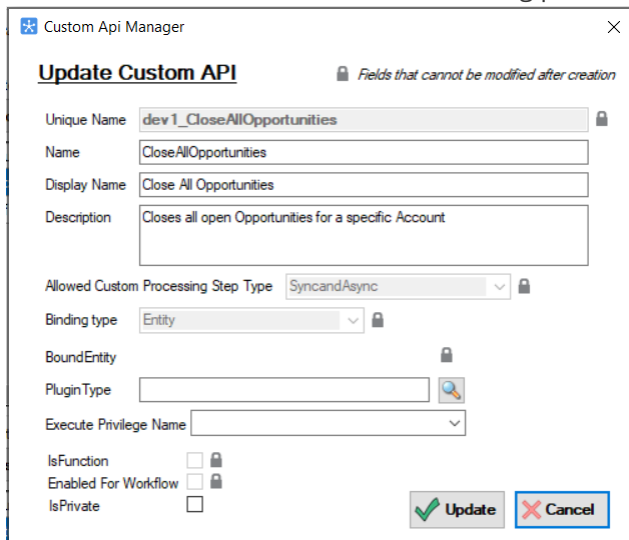
`dataverse-ify` is an open source library that aims to provide an interface to the Dataverse `webApi` from TypeScript that works in a similar way to the C# `IOrganisationService` so that you do not need to code around the complexities of the native `webApi` syntax. See <https://github.com/scottdurow/dataverse-ify/wiki>

Create the Custom API Definition

First we are going to implement a Custom API that does the same as the TypeScript called to close all open opportunities. Before we write the code, we will define the Input and Output parameters using the [Custom API Manager plugin for the XrmToolBox by David Rivard](#). You can also add Custom APIs using the solution explorer.

Firstly install the Plugin into the XrmToolBox and open a connection to your environment.

Create a new Custom API with the following parameters:



The screenshot shows the 'Custom API Manager' dialog box with the 'Update Custom API' tab selected. The form contains the following fields and options:

- Unique Name:** `dev_1_CloseAllOpportunities` (locked)
- Name:** `CloseAllOpportunities`
- Display Name:** `Close All Opportunities`
- Description:** `Closes all open Opportunities for a specific Account`
- Allowed Custom Processing Step Type:** `SyncandAsync` (locked)
- Binding type:** `Entity` (locked)
- BoundEntity:** (locked)
- PluginType:** (empty, with a search icon)
- Execute Privilege Name:** (empty)
- IsFunction:** ☐ (locked)
- Enabled For Workflow:** ☐ (locked)
- IsPrivate:** ☐
- Buttons:** and

Now create the following **input** parameters:

Unique Name: `subject`

- **Name:** `CloseAllOpportunities-In-Subject`
- **Display Name:** `CloseAllOpportunities In Subject`

Note: This name is shown in the maker portal so ensure it is fully descriptive to distinguish from other parameters.

- **Description:** The subject to add to the Close Opportunity Activity
- **Type:** `String`

Unique Name: `whatIf`

- Name: `CloseAllOpportunities-In-whatIf`
- Display Name: `CloseAllOpportunities In whatIf`
- Description: Set to true to just return the opportunities that will be closed
- Type: `Boolean`
- Is Optional : `True`

Also create the following output parameter:

Unique Name: `opportunityCount`

- Name: `CloseAllOpportunities-Out-OpportunityCount`
- Display Name: `CloseAllOpportunities Out OpportunityCount`
- Description: The number of opportunities closed (or that will be closed if the What If input parameter is true)
- Type: `Integer`

It should looks similar to:

The screenshot displays the 'Custom Api' configuration interface. On the left, the 'API' tab is active, showing fields for 'Unique Name' (dev1_CloseAllOpportunities), 'Name' (CloseAllOpportunities), 'Display Name' (Close All Opportunities), 'Description' (Closes all open Opportunities for a specific Account), 'Allowed Custom Processing Step Type' (Sync and Async), 'Binding type' (Entity), 'Bound Entity' (account), 'Plugin Type', 'Execute Privilege Name', 'IsFunction' (No), 'Enabled For Workflow' (No), 'IsPrivate' (No), 'Status' (Unmanaged), and 'IsCustomizable' (True). On the right, the 'Request Parameters (Input)' and 'Response Properties (Output)' tabs are visible. The 'Request Parameters (Input)' tab shows a table with two parameters: 'subject' (String, No) and 'whatIf' (Boolean, Yes). The 'Response Properties (Output)' tab shows a table with one parameter: 'opportunityCount' (Integer). Both tabs have 'Edit' and 'Delete' buttons for each parameter.

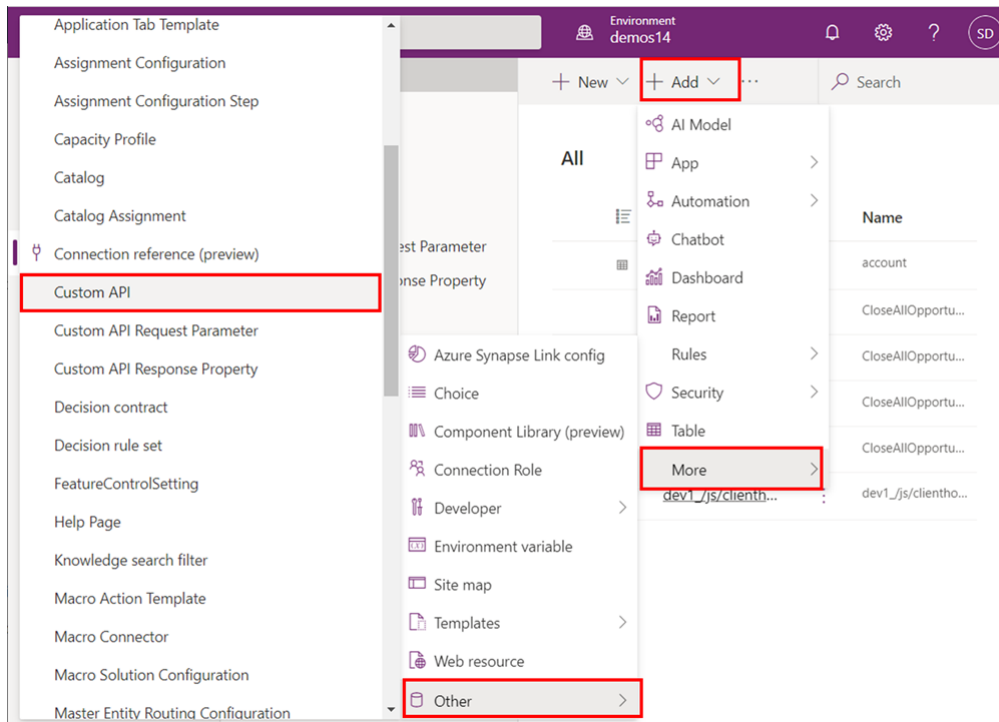
Unique Name	Type	Is Optional
subject	String	No
whatIf	Boolean	Yes

Unique Name	Type
opportunityCount	Integer

Note: The unique name is the name that is exposed to our code. I have chosen to use `camelCase` since we will be calling via TypeScript - but equally `pascalCase` would be acceptable - especially if you are calling via another C# client.

Add to the solution

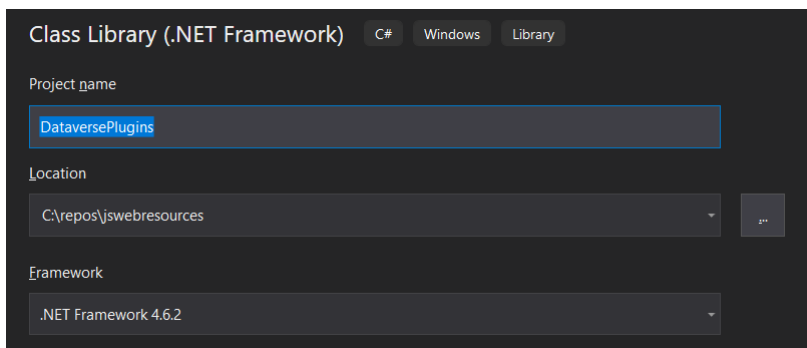
Now you have added your Custom API Definition, you will need to open your solution in make.powerapps.com and add the Custom API:



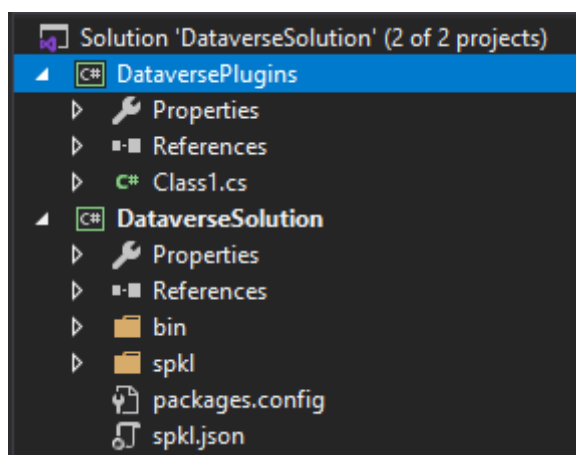
Create c# Plugin

We can now add the implementation of our Custom API using a c# plugin.

1. Create a new Visual Studio Project in the same solution as the `DataverseSolution` project we created in Part 4.
2. Select 'Class Library (.Net Framework)'
3. Name the project `DataversePlugins` and ensure the .Net Framework version is 4.6.2



4. You should end up with a solution that looks like the following:



5. Right click on the project and select Manage NuGet Packages
6. Select the Browse tab and search for spkl as we did in Part 4. Install `spkl` to the Plugin Project

7. Also search for `Microsoft.CrmSdk.XrmTooling.CoreAssembly` and install the latest version into your Plugin Project
8. Rename `Class1.cs` to `CloseAllOpportunitiesPlugin.cs`
9. Update the code to the following:

```
using Microsoft.Xrm.Sdk;
using System;

namespace DataversePlugins
{
    [CrmPluginRegistration("dev1_CloseAllOpportunities")]
    public class CloseAllOpportunitiesPlugin : IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            throw new NotImplementedException();
        }
    }
}
```

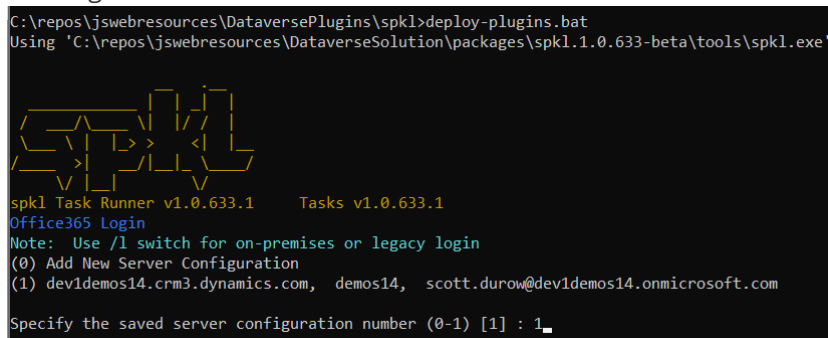
Deploy Plugin

So that we can wire up our Custom API definition to the plugin, we must deploy our plugin. This can be done easily using spkl.

1. Open the Properties on your `DataversePlugins` project, and select the Signing Tab.
2. Check Sign the Assembly and then **<New..>**. Provide a certificate name (e.g. Plugins) and uncheck **Protect my key file with a password**
3. Save the project `Ctrl + S`
4. Build you Plugin Project `Ctrl + Shift + B`
5. Add the command line with the current directory set to be the `DataversePlugins\spkl` folder, type:

```
C:\jswebresources\DataversePlugins\spkl>deploy-plugins.bat
```

7. Login to your Dataverse Environment as we did in Part 4 - or select an existing connection to the target environment.



```
C:\repos\jswebresources\DataversePlugins\spkl>deploy-plugins.bat
Using 'C:\repos\jswebresources\DataverseSolution\packages\spkl.1.0.633-beta\tools\spkl.exe'

spkl Task Runner v1.0.633.1    Tasks v1.0.633.1
Office365 Login
Note: Use /l switch for on-premises or legacy login
(0) Add New Server Configuration
(1) dev1demos14.crm3.dynamics.com, demos14, scott.durow@dev1demos14.onmicrosoft.com

Specify the saved server configuration number (0-1) [1] : 1
```

8. You should see a message that your plugin has been deployed and registered against your custom API.

```

Deploying Plugins
Searching for plugin config in 'C:\repos\jswebresources\DataversePlugins\spk1\..'
Using Config 'C:\repos\jswebresources\DataversePlugins'
Checking assembly 'DataversePlugins.dll' for plugins
1 plugin(s) found!
Registering Plugin 'DataversePlugins' from 'C:\repos\jswebresources\DataversePlugins\bin\Debug\DataversePlugins.dll'
Registering Type 'DataversePlugins.CloseAllOpportunitiesPlugin'
Registered Plugin Type 6050985b-c3bc-eb11-bacc-000d3ae992a3 against Custom Api 'dev1_CloseAllOpportunities'
Checking assembly 'Microsoft.IdentityModel.Clients.ActiveDirectory.Platform.dll' for plugins
Processed 1 config(s)
Press any key to continue . . .

```

Early Bound Types for your Custom API Plugin

1. Edit the `spk1.json` file so that the existing `earlyboundtypes` node has the following:

```

"earlyboundtypes": [
  {
    "entities": "account,opportunity,opportunityclose",
    "actions": "",
    "generateOptionsetEnums": "true",
    "generateStateEnums": "true",
    "generateGlobalOptionsets": false,
    "filename": "EarlyBoundTypes.cs",
    "oneTypePerFile": false,
    "classNameSpace": "DataversePlugins",
    "serviceContextName": "XrmSvc"
  }
]

```

Note: The key here is that we are adding the `account`, `opportunity`, & `opportunityclose` to be generated as early bound types. The `winOpportunityRequest` is already available under the `Microsoft.Crm.Sdk.Messages` namespace

2. At the command line under the `DataversePlugins\spk1` folder, type:

```
C:\jswebresources\DataversePlugins\spk1>earlybound.bat
```

3. You will now get a file generated in your project called `EarlyBoundTypes.cs` which you can add to your project by selecting **Show All Files** on the project context menu, and then **Include in Project** from the file context menu.

Add Integration Test

We can test our plugin logically against Dataverse before we deploy and test so that we can step through our code that would normally be executing on the server.

First we need to create an application user to authenticate against Dataverse. If you don't want to perform integration tests inside Visual Studio, can skip this section.

Open PowerShell ISE as an Administrator



Windows PowerShell ISE

App

Open

Run as administrator

Open file location

Create a using the instructions here [Service Principle](#)

1. Download the script [New-CrmServicePrinciple.ps1](#)
2. Open this in the PowerShell ISE Admin session and press the green **Run Script** or press **F9**
3. You will be asked to login twice - first login with a user that is an Azure AD Administrator and then login with a user that is a Power Platform Administrator - you can use the same account for both logins if you have one!
4. You will be given a set of values that you must make a note of since you will not be able to get them again:

TenantId	ApplicationId	ClientSecret
23b84661-8443-4743-8047-f16a9b433785	4d4713bc-322f-485a-a58c-e9f52d5c218d	8y11edba-17916a427e0276034c-703a

5. Download the [AddApplicationUserToEnv.ps1](#)
6. Open the Script in your PowerShell ISE, and replace the `ApplicationId` with the value from step 4

```
Install-Module -Name AzureAD -AllowClobber -Scope CurrentUser
Install-Module -Name Microsoft.Xrm.OnlineManagementAPI -MaximumVersion 1.2.0.1 -Scope CurrentUser
Install-Module -Name Microsoft.Xrm.Data.PowerShell -Scope CurrentUser -AllowClobber

$appId = [Guid]"a2e251bc-322f-485a-a58c-e9f52d5c218d"
```

7. Press the green **Run Script** or press **F9**
8. Accept the request to install the modules.
9. You should get a message that the Application User has been added.

Add a new project of type **Unit Test Project (.Net Framework)** called

`DataversePlugins.IntegrationTest`

Configure your new project

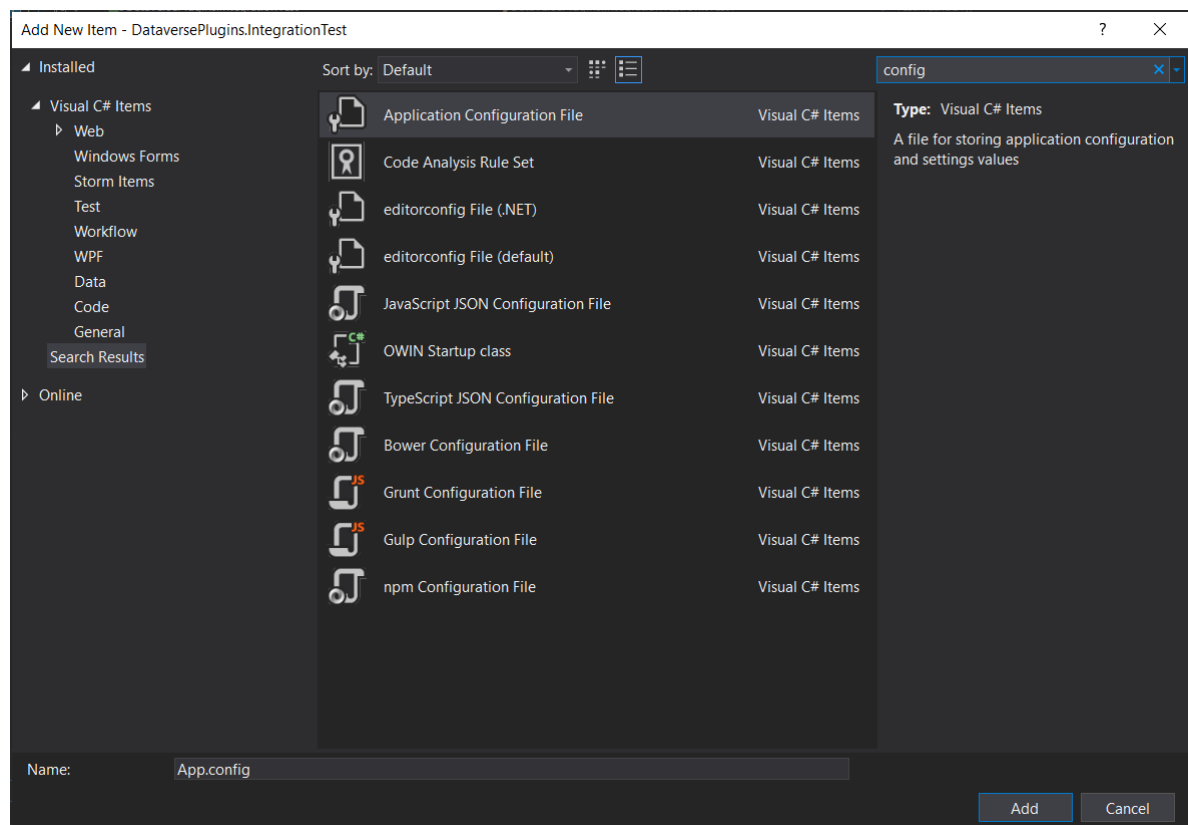
Unit Test Project (.NET Framework) C# Windows Test

Project name
DataversePlugins.IntegrationTest

Location
C:\repos\jswebresources

Framework
.NET Framework 4.6.2

To your Test project add a new `app.config`



To the `app.config`, add the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="IntegrationTestConnectionString"
value="AuthType=ClientSecret;url=https://<YOUR_ORG>.
<YOUR_REGION>.dynamics.com;ClientId=<APPLICATION_ID>;ClientSecret=
<CLIENT_SECRET>" />
  </appSettings>
</configuration>
```

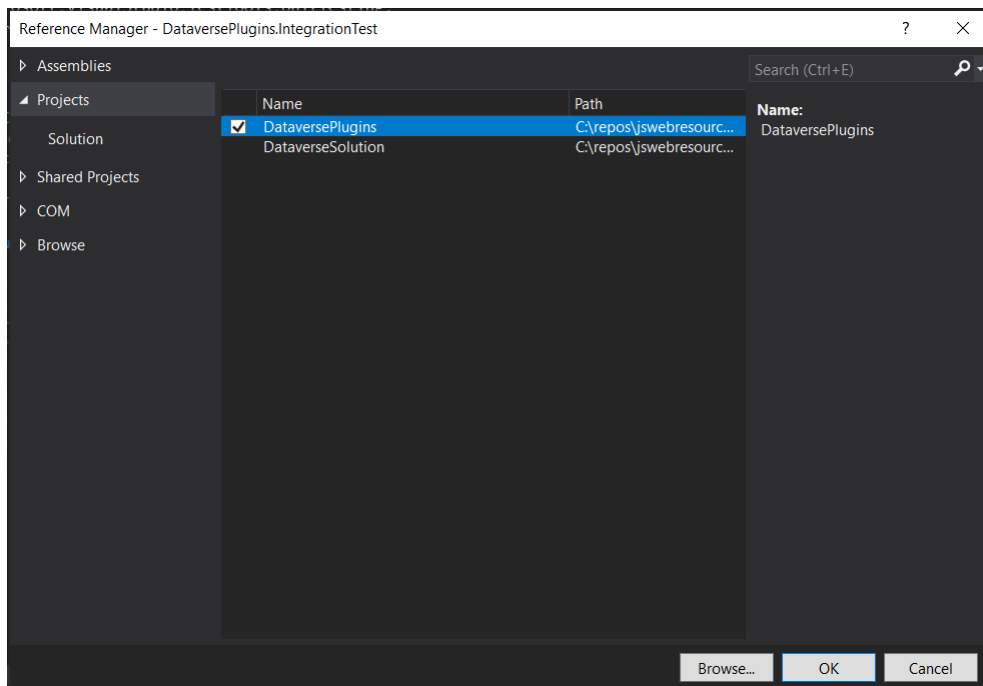
The `ClientId` is the `Application ID` from when you ran the first PowerShell script, as is the `Client Secret`.

Right click on the Test Project and **Manage NuGet Packages**

Install the following packages to the Test Project:

- `spkl.fakes`
- `System.Configuration.ConfigurationManager`
- `Microsoft.Xrm.Tooling.CoreAssembly`

Also, add a reference to your Plugin Project:



Add the following code to the test class:

```
using Microsoft.Crm.Sdk.Fakes;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Xrm.Sdk;
using Microsoft.Xrm.Tooling.Connector;
using System.Configuration;

namespace DataversePlugins.IntegrationTest
{
    [TestClass]
    public class CloseAllOpportunitiesTest : CloseAllOpportunitiesPlugin
    {
        [TestMethod]
        public void CloseAllOpportunities()
        {
            var connectionString =
                ConfigurationManager.AppSettings["IntegrationTestConnectionString"];
            var service = new CrmServiceClient(connectionString);

            // Arrange
            var account = new Account()
            {
                Name = "Integration Test",
            };
            account.Id = service.Create(account);
            var opportunity = new Opportunity()
            {
                Name = "Test"
            };
            opportunity.CustomerId = account.ToEntityReference();
            opportunity.Id = service.Create(opportunity);

            // Act
            try
            {

```



```

        using (var pipeline = new
PluginPipeline("dev1_CloseAllOpportunities", FakeStages.PostOperation,
account.ToEntity<Entity>(), service))
        {
            pipeline.InputParameters["subject"] = "Test";
            pipeline.Execute(this);
        }

        // Assert
        var closedOp = service.Retrieve(opportunity.LogicalName,
opportunity.Id, new Microsoft.Xrm.Sdk.Query.ColumnSet(new[] { "statecode" })) as
Opportunity;

        Assert.AreEqual(OpportunityState.Won, closedOp.StateCode);
    }
    finally
    {
        service.Delete(opportunity.LogicalName, opportunity.Id);
        service.Delete(account.LogicalName, account.Id);
    }
};

[TestMethod]
public void CloseAllOpportunitiesWhatIf()
{
    var connectionString =
ConfigurationManager.AppSettings["IntegrationTestConnectionString"];
    var service = new CrmServiceClient(connectionString);

    // Arrange
    var account = new Account()
    {
        Name = "Integration Test",
    };

    account.Id = service.Create(account);
    var opportunity = new Opportunity()
    {
        Name = "Test"
    };

    opportunity.CustomerId = account.ToEntityReference();
    opportunity.Id = service.Create(opportunity);

    // Act
    try
    {
        using (var pipeline = new
PluginPipeline("dev1_CloseAllOpportunities", FakeStages.PostOperation,
account.ToEntity<Entity>(), service))
        {
            pipeline.InputParameters["subject"] = "Test";
            pipeline.InputParameters["whatIf"] = true;
            pipeline.Execute(this);
        }

        // Assert
        var closedOp = service.Retrieve(opportunity.LogicalName,
opportunity.Id, new Microsoft.Xrm.Sdk.Query.ColumnSet(new[] { "statecode" })) as
Opportunity;

```

```

        Assert.AreEqual(OpportunityState.Open, closedOp.StateCode);
    }
    finally
    {
        service.Delete(opportunity.LogicalName, opportunity.Id);
        service.Delete(account.LogicalName, account.Id);
    };
}
}
}

```

These two tests do a very similar task to the integration tests that we build in Part 7. We test that the Custom API is working by arranging the records needed, calling the plugin, checking the actions have been carried out and then tidying up.

If you run the tests, they will fail since we have not implemented the logic yet!

Implement the Custom API Logic

Inside the `DataversePlugins` project, add the following code to the Plugin we created above:

```
using Microsoft.Crm.Sdk.Messages;
using Microsoft.Xrm.Sdk;
using Microsoft.Xrm.Sdk.Query;
using System;

namespace DataversePlugins
{
    [CrmPluginRegistration("dev1_CloseAllOpportunities")]
    public class CloseAllOpportunitiesPlugin : IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            ITracingService tracingService =
                (ITracingService)serviceProvider.GetService(typeof(ITracingService));
            IPluginExecutionContext context = (IPluginExecutionContext)
                serviceProvider.GetService(typeof(IPluginExecutionContext));
            IOrganizationServiceFactory factory =
                (IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));
            var service = factory.CreateOrganizationService(context.UserId);
            try
            {
                var accountid = context.PrimaryEntityId;
                var subject = (string)context.InputParameters["subject"];
                var whatIf = context.InputParameters.ContainsKey("whatIf") ?
                    (bool?)context.InputParameters["whatIf"] : null;

                var fetchQuery = string.Format(@"<fetch>
                    <entity name='opportunity' >
                        <attribute name='opportunityid' />
                        <attribute name='name' />
                        <filter>
                            <condition attribute='customerid'
                                operator='eq' value='{0}' />
                        </filter>
                    </entity>
                ");
            }
            catch { }
        }
    }
}
```

```

                                <condition attribute='statecode'
operator='eq' value='0' />
                                </filter>
                                </entity>
                                </fetch>", accountId);
        tracingService.Trace("Executing fetch query:{0}", fetchQuery);

        // Count Open Opportunities
        var openOpportunities = service.RetrieveMultiple(new
FetchExpression(fetchQuery));
        var opportunityCount = openOpportunities.Entities.Count;

        context.OutputParameters["opportunityCount"] = opportunityCount;

        if (whatIf == true)
            return;

        foreach (var opportunity in openOpportunities.Entities)
        {
            var closeRequest = new WinOpportunityRequest
            {
                Status = new
OptionSetValue((int)opportunity_statuscode.Won),
                OpportunityClose = new OpportunityClose
                {
                    Subject = subject,
                    OpportunityId = opportunity.ToEntityReference()
                }
            };
            service.Execute(closeRequest);
        }
        catch (Exception ex)
        {
            throw new InvalidPluginExecutionException("Cannot close
Opportunities:" + ex.ToString(), ex);
        }
    }
}

```

If you now run the integration tests, they should show as passing if everything is configured correctly. You can step through the code to see it executing.

The last thing to do here is deploy your plugin at the command line under the `DataversePlugins\spk1` folder, using:

```
C:\jswebresources\DataversePlugins\spk1>deploy-plugins.bat
```

Generate the TypeScript Custom API Types

Now we have our Custom API working, we need to generate the types to call it using `dataverse-ify` and then change the TypeScript code accordingly.

Open your TypeScript `client.js` project and at the command prompt type:

```
npx dataverse-gen init
```

When you are prompted to select actions, you should now be able to select our custom API `dev1_CloseAllOpportunities`

This will generate a new set of types and metadata for the custom API. Update the TypeScript `AccountRibbon.closeOpportunitiesInternal` function with the following:

```
static async closeOpportunitiesInternal2(serviceClient: CdsServiceClient,
accountid: string): Promise<void> {
  setMetadataCache(metadataCache);
  const closeOpportunityRequest = {
    logicalName: dev1_CloseAllOpportunitiesMetadata.operationName,
    entity: new EntityReference(accountMetadata.logicalName, accountid),
    subject: "Closed",
    whatIf: true,
  } as dev1_CloseAllOpportunitiesRequest;
  try {
    const closeResponseWhatIf = (await serviceClient.execute(
      closeOpportunityRequest,
    )) as dev1_CloseAllOpportunitiesResponse;

    const openOps = closeResponseWhatIf.opportunityCount;

    if (openOps == 0) {
      await Xrm.Navigation.openAlertDialog({ text: `There are no open
opportunities!` });
      return;
    }

    const result = await Xrm.Navigation.openConfirmDialog({
      title: "Close All Open Opportunities?",
      text: `Are you sure you want to close the ${openOps} open opportunities?`,
    });
    if (!result.confirmed) return;
    closeOpportunityRequest.whatIf = false;
    Xrm.Utility.showProgressIndicator("Closing Opportunities, Please Wait...");

    const closeResponse = (await serviceClient.execute(
      closeOpportunityRequest,
      // eslint-disable-next-line camelcase
    )) as dev1_CloseAllOpportunitiesResponse;
    Xrm.Utility.closeProgressIndicator();
    await Xrm.Navigation.openAlertDialog({
      title: "Success",
      text: `${closeResponse.opportunityCount} opportunities closed`,
    });
  } catch (ex) {
    Xrm.Utility.closeProgressIndicator();
    Xrm.Navigation.openErrorDialog({
      details: ex,
      message: `Could not close opportunites:\n${ex.message}\n`,
    });
  }
}
```

You'll see that we are now calling execute with `dev1_CloseAllOpportunitiesResponse` - first with `whatIf` set to `true`, and then with it set to false to actually close the opportunities.

`dataverse-ify` makes it really easy because you don't have to remember all the input and output parameters- they are all generated for you and strongly typed.

This is the end of the first section!

In the next section we will look at managing the Application Lifecycle of our solution with automated builds and deployments of our TypeScript and C# code.