Domo Workbench Plugin Development

Workbench Plugin Development Guide

Contents

[2 Overview 4](#_Toc447543930)

[3 Getting Started 4](#_Toc447543931)

[3.1 System Requirements 4](#_Toc447543932)

[3.2 Installing the Plugin Template 4](#_Toc447543933)

[3.3 Starting a New Project 5](#_Toc447543934)

[3.4 Running and Debugging 6](#_Toc447543935)

[3.4.1 Debug Registry Flags 6](#_Toc447543936)

[3.4.2 Registering Your Plugin 7](#_Toc447543937)

[4 Plugin Template Structure 8](#_Toc447543938)

[5 Communicating With Domo Workbench 9](#_Toc447543939)

[5.1 IWorkbenchHost 9](#_Toc447543940)

[5.1.1 Interface Definition 9](#_Toc447543941)

[5.2 IWorkbenchHostImpersonation 11](#_Toc447543942)

[5.2.1 Interface Definition 11](#_Toc447543943)

[5.3 IWorkbenchJob 11](#_Toc447543944)

[5.3.1 Interface Definition 11](#_Toc447543945)

[5.4 IWorkbenchPluginColumn 12](#_Toc447543946)

[5.4.1 Interface Definition 12](#_Toc447543947)

[5.5 IWorkbenchPluginColumnMetadata 12](#_Toc447543948)

[5.5.1 Interface Definition 12](#_Toc447543949)

[5.6 IWorkbenchHostConnectionInfo 12](#_Toc447543950)

[5.6.1 Interface Definition 13](#_Toc447543951)

[5.7 IWorkbenchHostConnectionInfoProxyClient 13](#_Toc447543952)

[5.7.1 Interface Definition 13](#_Toc447543953)

[6 DataReader 13](#_Toc447543954)

[6.1 IWorkbenchDataReaderPlugin 14](#_Toc447543955)

[6.1.1 Interface Definition 14](#_Toc447543956)

[6.2 IWorkbenchDataReaderPluginEditor 15](#_Toc447543957)

[6.2.1 Interface Definition 16](#_Toc447543958)

[6.3 IWorkbenchDataReaderPluginExecutionCharacteristics 16](#_Toc447543959)

[6.3.1 Interface Definition 16](#_Toc447543960)

[6.4 IWorkbenchDataReaderPluginValidationResult 16](#_Toc447543961)

[6.4.1 Interface Definition 16](#_Toc447543962)

[7 DataTransform 16](#_Toc447543963)

[7.1 IWorkbenchDataTransformPlugin 17](#_Toc447543964)

[7.1.1 Interface Definition 17](#_Toc447543965)

[7.2 IWorkbenchDataTransformPluginEditor 17](#_Toc447543966)

[7.2.1 Interface Definition 17](#_Toc447543967)

[8 DataProvider 18](#_Toc447543968)

[8.1 IWorkbenchDataProviderPlugin 18](#_Toc447543969)

[8.1.1 Interface Definition 18](#_Toc447543970)

[8.2 IWorkbenchDataProviderPluginEditor 19](#_Toc447543971)

[8.2.1 Interface Definition 19](#_Toc447543972)

[8.3 IWorkbenchDataProviderPluginDatabaseHandler 20](#_Toc447543973)

[8.3.1 Interface Definition 20](#_Toc447543974)

[9 Distribution 20](#_Toc447543975)

[10 Example 20](#_Toc447543976)

[10.1 Creating the Project 20](#_Toc447543977)

[10.2 Retrieving Data from an Excel Sheet 26](#_Toc447543978)

[10.2.1 Retrieving Column Headers 26](#_Toc447543979)

[10.2.2 Retrieving Row Data 30](#_Toc447543980)

[10.3 Creating a User Interface 32](#_Toc447543981)

[10.4 Finishing Touches on the User Interface 37](#_Toc447543982)

# Overview

Domo Workbench version 4 has been rebuilt from the ground up to support a plugin structure. Workbench currently supports three types of plugins: DataReaders, DataProviders, and DataTransforms. This was done to allow users of Workbench to build their own custom plugins without having to change their system and data structures to conform to what is provided out of the box. This should allow for a greater amount of flexibility when pushing your data up to Domo.

In this document, we will go over at a very high level what each plugin type is for and then at a development level how to implement your first plugin. We will start with creating a new Domo Plugin Project and end up with a fully functional DataReader. All the plugins have a working implementation that can be registered in Domo Workbench and run immediately. This was done to provide a working example of how to save plugin properties from the UI controls and how each plugin type is expected to be used.

# Getting Started

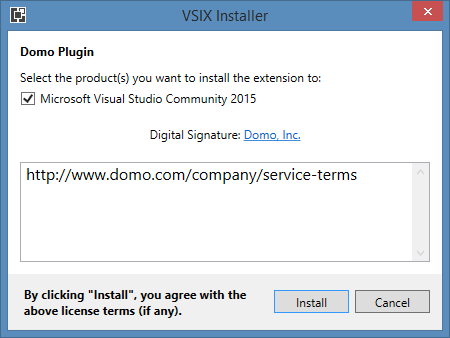
## System Requirements

In order to create, debug, and run a Domo Plugin, your system must meet the following criteria:

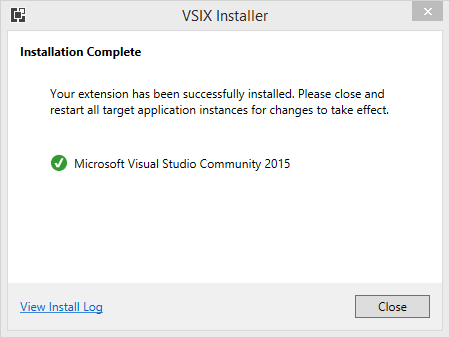
* Support for Microsoft .NET 4.5+ (Windows Vista+, Microsoft Server 2008+)
* Microsoft Visual Studio 2013+ (any edition)
  + Microsoft Visual Studio 2015 Community edition is the version that is being used for the purpose of this document
* Domo Workbench v4 Installed
  + Note: It is possible to create a plugin without having Domo Workbench installed. The plugin template is set up for Domo Workbench being installed in the default installation directory. If it is not installed or if you have a custom installation, you must manually change the library references to point to the correct location. You may not be able to debug or run the plugin if Workbench isn’t installed. The minimum requirements are:
    - DomoApi.dll
    - WorkbenchPlugin.Views.Plugin.v1.dll
  + You should also have a Domo Account added to Workbench. You cannot create new Jobs to test your plugins without having this setup.

## Installing the Plugin Template

The Plugin Template package installer is installed with Domo Workbench. It is placed into an SDK folder in the installation directory (default: C:\Program Files\Domo\Workbench\SDK) and is called DomoPluginTemplate.vsix. When you launch DomoPluginTemplate.vsix, you are presented with an installation screen similar to the picture below:



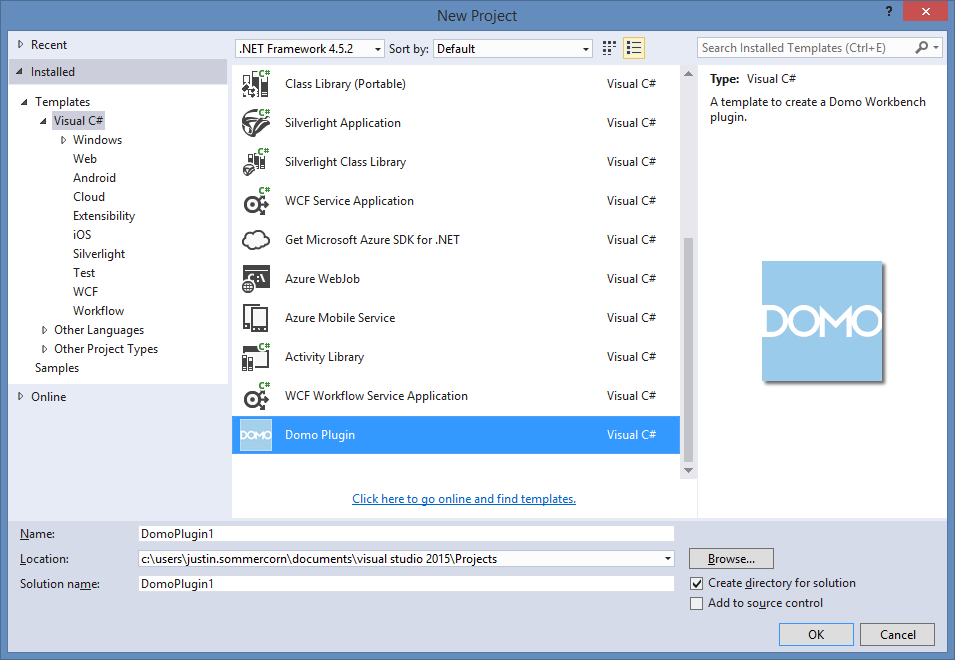
Simply select the version(s) of Microsoft Visual Studio to which you want to add the plugin template and then click **Install**. Upon a successful install, you see a screen similar to the following:



At this point, close any open instances of Microsoft Visual Studio. Upon relaunch, the template is ready to be used.

## Starting a New Project

To begin using the newly installed Domo Plugin Template, open up Microsoft Visual Studio and click **Create a New Project**. In the Visual C# list, select “Domo Plugin” and then enter a name in the **Solution name** field. Click **OK.** Now you have a fully functional set of plugins that you can immediately run and register in Workbench.



## Running and Debugging

The Domo Plugin uses the default installation location of Domo Workbench for debugging and the required libraries. If you have Workbench installed in the default installation directory, you don’t need to worry about any configuration changes. If you don’t have Workbench installed or if it’s not installed in the default location, you need to modify your library references and the debug startup program so that these will point to the proper locations. Once set up properly, running the project under Debug launches Domo Workbench and allows you to start stepping through your plugin code.

### Debug Registry Flags

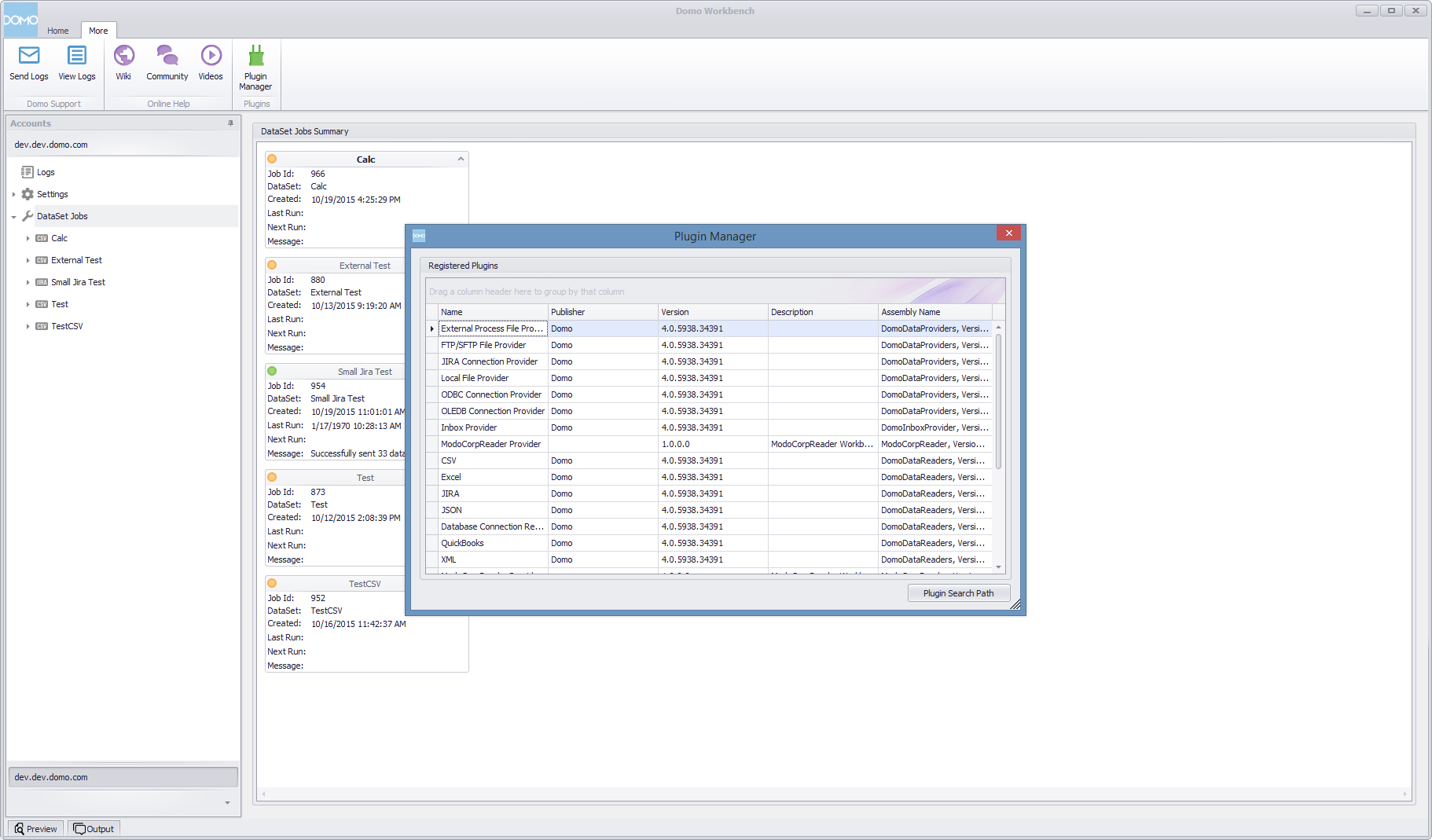
There are a few helpful flags that you can set up in the Windows Registry to help with debugging. Both are REG\_SZ types and may or may not be there by default. If they don’t exist already, you can safely create them at any point. The changes take effect immediately; it shouldn’t be necessary to restart the Workbench Service. It is important to remember that these are global flags on your PC, and if you switch either of these flags, Workbench uses them regardless of whether it has been launched via Microsoft Visual Studio or through the Workbench application. (You will switch a flag in the example provided in Section 9 in this document.)

Here are the flags you can set up in Windows Registry, with their default values:

* HKEY\_LOCAL\_MACHINE\SOFTWARE\Domo\Workbench\UseServiceJobExecution
  + Default value: True
  + This flag, when changed to False, causes Domo Workbench to run the Job rather than using the default method of going through the Workbench Service to execute the job. This is the easiest way to debug during development.
* HKEY\_LOCAL\_MACHINE\SOFTWARE\Domo\Workbench\DebugDataCollectorProcess
  + Default value: False
  + This flag, when changed to True, causes the DomoDataCollector process to pause execution until a debugger is attached. This is useful when you want to debug when the Job is launched through the Workbench Service.

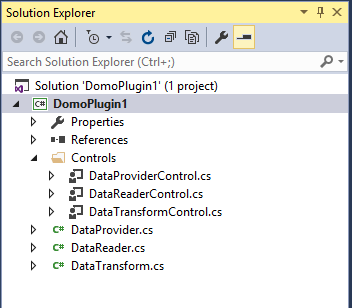
### Registering Your Plugin

Before you can use your new plugin, you must register it within Domo Workbench. There are a few ways to do this:

* You can register your plugin in Workbench by clicking **More** on the ribbon bar and then selecting **Plugin Manager**. Click **Plugin Search Path** and then browse to your plugin’s parent directory. The directory chosen should be a level up from where the DLL is located. It’s important to know which version of the DLL you’re registering, as it won’t automatically switch between the Debug and Release builds. 
* You can also register your plugin in Workbench by modifying the PluginPath REG\_MULTI\_SZ value in the Windows Registry at HKEY\_LOCAL\_MACHINE\SOFTWARE\Domo\Workbench. Add a new row with the plugin’s parent directory path.

# Plugin Template Structure

The Domo Plugin has a very simple structure. There are 3 classes. Each class is named for the type of plugin it corresponds to, and each has a corresponding UI component in the Controls folder. Within each of the plugin classes, there are 2 classes. One is the plugin itself, and the other is a Property class used to hold any values set in the UI.



# Communicating With Domo Workbench

Each plugin type provides an Initialize method that contains an instance of IWorkbenchHost. This interface is used to both push and request certain information directly from Domo Workbench. This information ranges from schema information to UI page navigation.

## IWorkbenchHost

IWorkbenchHost is provided so that your plugin can communicate with Domo Workbench. It is always passed to your plugin via an Initialize method. When you need specific information about a job, such as schema, and your plugin doesn’t know about it already, this would be the way to request that information.

### Interface Definition

* event EventHandler<IList<Dictionary<string, string>>> GetDataProviderSchemaComplete
  + This event is fired once the full GetSchema process from the DataProvider has completed. This is used to help keep the call to the DataProvider asynchronous so that the UI doesn’t lock up while the list is being populated.
* JobExecutionType JobExecutionType { get; set; }
  + This property is used to either get or set the JobExecutionType value. In general, Domo Workbench sets this value, there are only a few rare circumstances where setting the value is necessary.
* void BeginGetDataProviderSchema(string collection, string[] filters)
  + This function begins the process of the DataProvider GetSchema method. It is intended to be asynchronous so as to not lock the UI while the list is being populated.
* string CreateDataSetID()
  + This method is used to tell Domo Workbench that it needs to go through the process of creating a DataSet for the Job if one doesn’t exist. It will return the newly acquired DataSetID value.
* string DecryptString(string encryptedData)
  + This method is used to decrypt an encrypted string for your plugin to use.
* string EncryptString(string data)
  + This method is used to encrypt a sensitive string.
* string ExecuteJobToFile(long jobId)
  + This method is used to execute a Job within your plugin. This is a blocking method and will return the directory of the parsed files of the completed Job. This will not upload the data to Domo.
* IWorkbenchJob GetJob()
  + This method gets information on the currently executing Job.
* IList<IWorkbenchJob> GetJobList()
  + This method gets information on all Jobs setup for the current domain.
* IList<IWorkbenchPluginColumn> GetJobSchemaColumns()
  + This method is used to return the schema saved in the Job. It can be used to compare schema determined from the DataReader to inform the user and Workbench if there are schema changes.
* IList<IWorkbenchPluginColumn> GetJobSchemaColumns(long jobId)
  + This method is used to get the schema of a specific Job.
* IWorkbenchHostConnectionInfo GetLogonDetails()
  + This method is used to get the current domain logon details so that a DomoClient can be setup and calls to DomoApi can be made.
* string GetProviderProperties()
  + This method returns the serialized string value of the DataProvider properties.
* string GetReaderProperties()
  + This method returns the serialized string value of the DataReader properties.
* string GetTransformProperties()
  + This method returns the serialized string value of the DataTransform properties.
* void LogEvent(LogMessageType type, string message, Exception ex = null)
  + This method is used to send logs back to Workbench to both be displayed in the UI (if it’s running) and to be written to the log files.
* void SetJobChange(IWorkbenchJobChange jobChange)
  + This method tells Workbench that there is a job change that may need to be applied to other plugins, such as ODBC credentials being updated.
* void SetJobSchemaColumns(IList<IWorkbenchPluginColumn> columns)
  + This method is used to update the schema in the Job.
* void SetNavigationPage(WorkbenchPages page)
  + This method is used to navigate to a different page in the UI to help the user properly setup the Job. For example, if impersonation needs to be setup, a link can be setup to navigate the user to the Setup page to add impersonation credentials.
* void SetProviderProperties(string json)
  + This method is used to save the serialized string value of the DataProvider properties.
* void SetReaderProperties(string json)
  + This method is used to save the serialized string value of the DataReader properties.
* void SetTransformProperties(string json)
  + This method is used to save the serialized string value of the DataTransform properties.

## IWorkbenchJob

IWorkbenchJob is provided to give specific information about a Job.

### Interface Definition

* string DataSetId { get; }
  + The DataSetId of the Job, this value can be null if the DataSet has not yet been created.
* long Id { get; }
  + The Id of the Job.
* string Name { get; }
  + The name of the Job.

## IWorkbenchJobChange

IWorkbenchJobChange is provided to send a payload to plugins of the same type to perform an update across multiple jobs. It is up to the plugin to determine how and when to use this or if it will be supported at all.

### Interface Definition

* string Description { get; }
  + A description of the change. This will appear in the Workbench user interface to ask the user for confirmation that they wish to apply the change to other plugins.
* string Key { get; }
  + The key value of the change, this allows a plugin to determine how to handle this change when it is called back. It is up to the plugin to determine what changes are supported.
* string Payload { get; }
  + The data that describes what changes should be made. The plugin sets this when it makes the change, and receives it when called back to apply changes from another plugin.

## IWorkbenchPluginColumn

IWorkbenchPluginColumn is provided to give information about a specific column and how it’s to be treated within Domo.

### Interface Definition

* string ColumnId { get; }
  + The Id of the column.
* ColumnType ColumnType { get; }
  + The type of where the column data came from.
* string DestinationColumn { get; }
  + The destination name of the column.
* bool Include { get; }
  + Whether or not the column will be included when uploading to Domo.
* IWorkbenchPluginColumnMetadata Metadata { get; }
  + Any metadata that exists on the column.
* string Name { get; }
  + The name of the column.
* string SourceColumn { get; }
  + The name of the source column that populated the data for this column.
* string TransformId { get; }
  + The Id of a transform that created or modified the column.
* DataType Type { get; }
  + The type of data the column holds.
* bool UpsertKey { get; }
  + Whether or not to use this column as an upsert key.
* bool UserOverride { get; }
  + Whether or not the user has overridden the default values of this column.

## IWorkbenchPluginColumnMetadata

IWorkbenchPluginColumnMetadata is provided to give specific metadata information for a column.

### Interface Definition

* bool Encrypted { get; }
  + Whether or not the data in this column should be treated as if it’s encrypted within Domo.

## IWorkbenchHostConnectionInfo

IWorkbenchHostConnectionInfo is provided to give the plugin enough information to create a DomoConnection object in order to utilize DomoApi if necessary.

### Interface Definition

* string ApiKey { get; }
  + The ApiKey necessary to login to the customer’s instance of Domo.
* string DomoDomain { get; }
  + The customer’s Domo domain name.
* IWorkbenchHostConnectionInfoProxyClient ProxyClient { get; }
  + Any proxy information necessary for reaching the customer’s instance of Domo.

## IWorkbenchHostConnectionInfoProxyClient

IWorkbenchHostConnectionInfoProxyClient is provided to give proxy login and credentials in order to reach out to the customer’s instance of Domo.

### Interface Definition

* string Domain { get; }
  + The domain of the user to login to the proxy.
* string Password { get; }
  + The password of the user to login to the proxy.
* int Port { get; }
  + The port that should be used to get to the proxy.
* bool RequiresAuth { get; }
  + Whether or not the proxy requires authentication.
* string Server { get; }
  + The server that the proxy is on.
* ProxyType Type { get; }
  + The type of how the proxy is setup in Workbench.
* string Username { get; }
  + The username required to login to the proxy.

# DataReader

The DataReader does all of the following:

* Opens and parses the values returned from the DataProvider
* Returns the column header values along with each row of data arranged in the same order as the column headers
* Handles the lifespan of the DataProvider
* Handles data of any size and, to this end, it writes temporary files to disk as necessary to avoid crashing the application due to using too much memory and cleans up any temporary files created during parsing to avoid cluttering the disk.

## IWorkbenchDataReaderPlugin

The WorkbenchPlugin.Views.Plugin.v1 library provides an interface class called “IWorkbenchDataReaderPlugin.” This is used by Workbench to determine if your plugin contains a DataReader. This interface inherits the IDisposable interface. It has many properties and methods that control how your reader looks and works.

### Interface Definition

* ExecutionCharacteristics ExecutionCharacteristics { get; }
  + This property is used to set specific characteristics of the DataReader, such as concurrency, which allows only one of this DataReader type to be executed at a time.
* string Name { get; }
  + This property is used to identify your plugin within Workbench. This value is displayed to users in several areas within Workbench. For example, when you create a new Job, the name is displayed in a dropdown.
* byte[] NavigationImage { get; }
  + This property is used to display an image next to the Job node in the list of Jobs within Workbench. The image is automatically resized to fit where it is to be placed.
* string RecommendedDataSetType { get; }
  + This property will auto populate the DataSet type value during the Job creation wizard. If the value is null, the user must select a type.
* bool Requires32Bit { get; }
  + This property is used to force the DataCollector process to run in a 32 bit process. This should be used when the object you’re getting data from cannot be run from a 64 bit process. For example, you would set this override when getting data from QuickBooks, which is only 32 bit.
* bool RequiresInteractiveSession { get; }
  + This property is used to inform the Job execution that the source requires a process with an interactive session to work properly.
* string SupportedSourceTypes { get; }
  + This property is used to tell Domo Workbench what SourceTypes are supported by this DataReader. This value is used to narrow down which DataReaders are displayed to the user when creating a new Job after a DataProvider is selected (which has this property as well). This should be set using a value from WorkbenchPlugin.Views.Plugin.v1.SourceType to properly match a DataProvider filter although it is not required.
* void ApplyJobChanges(IList<IWorkbenchJobChange> jobChanges)
  + This method is used to receive a list of changes to update the Workbench Job. This method is only called when there are changes that can be sent to every job of the same type.
* void Cancel()
  + This method is used to recommend that the reader stop execution of any currently executing process.
* void Dispose()
  + This method is used to clean up any resources used by the DataReader.
* IWorkbenchDataReaderPluginEditor GetDataReaderEditor()
  + This method is used to create the UI component for this DataReader. This is called when the user selects the Source node within the Job to display any values the user can/needs to change.
* IList<string> GetHeaders()
  + This method is used to return the list of header values. The order of the returned list needs to be the same as the row values returned in another method. This method is called immediately after the Open method returns during the execution of the Job.
* IList<object> GetRowData()
  + This method is used to return the data one row at a time. These values need to be returned in the same order as their corresponding headers and should be returned in their expected value types. The schema type per column is based on the value types. This method gets repeatedly called during the Job execution for every row.
* void Initialize(IWorkbenchHost workbenchHost, IWorkbenchDataProviderPlugin dataProvider)
  + This method is used give the DataReader the IWorkbenchHost and IWorkbenchDataProviderPlugin classes in order to callback to Workbench for data and to utilize the DataProvider. This method is called immediately after the DataReader constructor.
* bool MoveNext()
  + This method is used to increment to the next row and return, whether or not there are more rows to be read. It is repeatedly called during the Job execution and determines if GetRowData should be called or not. When this returns False, Workbench knows there are no more rows to be read and finalizes the data. This will be the last method called in the lifecycle of the DataReader (with the exception of Dispose to clean up).
* void Open()
  + This method is used to call the DataProvider to get the value to attempt to open. This method is the first method called in the DataReader during the Job execution.
* IWorkbenchDataReaderPluginValidationResult ValidateProviderSettings()
  + This method is used to validate whether or not the settings provided for this DataReader are valid. For example, the ODBC DataReader uses this to determine if a query can be executed on the server returned by the DataProvider.

## IWorkbenchDataReaderPluginEditor

The IWorkbenchDataReaderPluginEditor is an interface used to define the UI portion of the DataReader and to add some default functionality.

### Interface Definition

* IList<IWorkbenchDataReaderPluginEditorActionButton> EditorRibbonButtons { get; }
  + This property is used to return any ribbon buttons that should be added with the DataReader. They are shown when the Source node is selected in the Job list.
* FrameworkElement GetWpfControl(out int desiredHeight)
  + This method is used to return the actual Control back to Domo Workbench.
* void LoadReaderProperties()
  + This method is used to signal the UI to load any DataReader properties from the Job. It is called when the DataProvider has informed the UI that its properties have been edited.
* public void ReaderPropertiesSaved()
  + This method is used to inform the UI that a save has been successful. It is called after the save method completes.

## IWorkbenchDataReaderPluginEditorActionButton

IWorkbenchDataReaderPluginEditorActionButton is provided to give Workbench information about buttons that are to be added to the Workbench Ribbon bar when the plugin is loaded.

### Interface Definition

* string ButtonCaption { get; }
  + The text to be displayed with the ribbon button.
* byte[] ButtonImage { get; }
  + The bytes of the image to be displayed on the button.
* string ButtonTooltip { get; }
  + The text to be displayed when the user hovers over the button.
* void ButtonClicked()
  + The action taken when the button is pressed by the user.

## IWorkbenchDataReaderPluginExecutionCharacteristics

IWorkbenchDataReaderPluginExecutionCharacteristics is provided to give information on any special circumstances as to how Workbench should run the Job.

### Interface Definition

* int ConcurrentExecutionLimit { get; }
  + The count of the number of Jobs that can be run concurrently.
* string QueueName { get; }
  + Overrides the default queue name of the Job.

## IWorkbenchDataReaderPluginValidationResult

IWorkbenchDataReaderPluginValidationResult is provided to send a validation result with any error message as well.

### Interface Definition

* string Error { get; }
  + An error message describing what went wrong.
* bool Valid { get; }
  + Whether or not the validation was successful.

# DataTransform

A DataTransform manipulates data values before sending them to Domo. It can manipulate data values in nearly any way by either changing the values directly or by adding new column(s) to the schema and then adding the value(s). DataTransforms are called just after the DataReader calls the GetRowData method (once per row). This interface inherits the IDisposable interface. Transforms are executed in the order they are created in the Job.

## IWorkbenchDataTransformPlugin

The WorkbenchPlugin.Views.Plugin.v1 library provides an interface called “IWorkbenchDataTransformPlugin” that is used by Workbench to determine if your plugin contains a DataTransform. It has many properties and methods that need to be implemented that Workbench calls in order to execute your transform logic.

### Interface Definition

* IList<IWorkbenchPluginColumn> ExecutionSchema { get; set; }
  + This property is used to identify expose the job schema to and from your plugin. It is provided to your plugin prior to the ExecuteDataTransform call and then if requiresSchemaUpdate is set to true, Workbench will update the job schema.
* string Name { get; }
  + This property is used to identify your plugin in Workbench. It is displayed in both the Job’s list of transforms and when creating/deleting a transform on the Job.
* void ApplyJobChanges(IList<IWorkbenchJobChange> jobChanges)
  + This method is used to receive a list of changes to update the Workbench Job. This method is only called when there are changes that can be sent to every job of the same type.
* void Cancel()
  + This method is used to handle a cancellation request from a user.
* IList<object> ExecuteDataTransform(IList<object> rowData, out bool requiresSchemaUpdate)
  + This method is used to perform the transform on the rowData. It can manipulate the columns and the rowData as necessary. This method is called after the DataReader calls GetRowData during the Job execution.
* IWorkbenchDataTransformPluginEditor GetDomoTransformEditor()
  + This method is used to create the UI component for this DataTransform. This is called when the user adds/edits the transform.
* void Initialize(IWorkbenchHost hostObj, IWorkbenchPluginColumn transformColumn)
  + This method is used give the DataTransform the IWorkbenchHost and IWorkbenchPluginColumn classes in order to callback to Workbench for data and to utilize the column setup to be transformed. This method is called immediately after the DataTransform constructor.
* void TransformRemoved()
  + This method is used to clean up any manipulations done to the Job when the transform was added. It is called when the transform is deleted from the Job.

## IWorkbenchDataTransformPluginEditor

The IWorkbenchDataTransformPluginEditor is an interface used to define the UI portion of the DataTransform and add some default functionality.

### Interface Definition

* event EventHandle<string> TransformPropertiesChanging
  + This event is used to inform Workbench that the properties of the DataTransform have changed. It is used to help update the values displayed to the user when viewing the transform list.
* FrameworkElement GetWpfControl(out int desiredHeight)
  + This method is used to return the actual Control back to Domo Workbench.
* void TransformPropertiesSaved()
  + This method is used to inform the UI that a save has been successful. It is called after the save method completes.

# DataProvider

The DataProvider tells the DataReader how to get to the data. The data can be a file or a database connection. It is up to the DataReader to parse the data provided by the DataProvider. DataProviders should not do any parsing or manipulation of data, only pass the information on.

## IWorkbenchDataProviderPlugin

The WorkbenchPlugin.Views.Plugin.v1 library provides an interface called “IWorkbenchDataProviderPlugin” that is used by Workbench to determine if your plugin contains a DataProvider. This interface inherits the IDisposable interface. It has many properties and methods that need to be implemented that Domo Workbench calls in order to pull data.

### Interface Definition

* bool AllowFileCopy { get; set; }
  + Whether or not the file can be copied to the local machine or if it needs to remain and used from the current location. This should generally be set to true.
* string DataProviderType { get; }
  + The type of data served by this DataProvider. This should always be a value from WorkbenchPlugin.Views.Plugin.v1.DataSetTypes.
* string Description { get; }
  + This property is used to help identify the DataProvider in the log files.
* string Name { get; }
  + This property is used to identify your plugin in Workbench. It is displayed in the Job settings and when creating a new Job in a dropdown list.
* string ProviderSourceType { get; }
  + This property is used to tell Domo Workbench what SourceTypes are supported by this DataProvider. This value is used to narrow down the DataReaders (which have this property as well) that are displayed to the user when creating a new Job after a DataProvider is selected. This should be set using a value from WorkbenchPlugin.Views.Plugin.v1.SourceType to properly filter available DataReaders although it is not required.
* bool Requires32Bit { get; }
  + This property is used to force the DataCollector process to run in a 32 bit process. This should be used when the object you’re getting data from cannot be run from a 64 bit process. For example, you would set this override when getting data from QuickBooks, which is only 32 bit.
* bool RequiresInteractiveSession { get; }
  + This property is used to inform the Job execution that the source requires a process with an interactive session to work properly.
* bool RequiresVerification { get; }
  + This property is used to inform Workbench that a verification should be executed on the settings of this DataProvider.
* bool SupportsFileWatcher { get; }
  + Whether or not this data provider can support watching a file for changes. This feature can only be used of the file is local.
* void ApplyJobChanges(IList<IWorkbenchJobChange> jobChanges)
  + This method is used to receive a list of changes to update the Workbench Job. This method is only called when there are changes that can be sent to every job of the same type.
* void Cancel()
  + This method is used to recommend that any running operation be cancelled.
* void Dispose()
  + This method is used to clean up any resources created by the DataProvider.
* IWorkbenchDataProviderPluginDatabaseHandler GetDatabaseResultHandler()
  + This method is used to return how to communicate with the database connection. This method will be called by the DataReader if the DataReader supports a database connection type.
* IWorkbenchDataProviderPluginEditor GetDataProviderEditor()
  + This method is used to create the UI component for this DataProvider. This is called when the user is in the Source node of the Job.
* List<Dictionary<string, string>> GetSchema(string collection, string[] filters)
  + This method is used to return the schema to the editors so that things like the database auto complete data can be populated.
* string GetSourceFilePath()
  + This method returns the file path for DataProviders that return a flat file type.
* void Initialize(IWorkbenchHost hostObj)
  + This method is used give the DataProvider the IWorkbenchHost class in order to callback to Workbench for data. This method is called immediately after the DataProvider constructor.

## IWorkbenchDataProviderPluginEditor

The IWorkbenchDataProviderPluginEditor is an interface used to define the UI portion of the DataProvider and add some default functionality.

### Interface Definition

* string DataProviderDescription { get; }
  + This property is a user friendly description of the DataProvider to be shown in Workbench. It is shown when the Source node of the Job list is selected.
* event OnProviderPropertiesUpdated OnPropertiesUpdated
  + This event is used to inform Workbench that the properties of the DataProvider have been updated. It lets the DataReader know that there have been changes to the underlying DataProvider and to reload the DataReaderProperties.
* FrameworkElement GetWpfControl(out int desiredHeight)
  + This method is used to return the actual Control back to Domo Workbench.
* void ProviderPropertiesSaved()
  + This method is used to inform the UI that a save has been successful. It is called after the save method completes.

## IWorkbenchDataProviderPluginDatabaseHandler

IWorkbenchDataProviderPluginDatabaseHandler is provided to hold the database connection in a way that the DataReader can interact with it without knowing about the connection itself.

### Interface Definition

* int ColumnCount { get; }
  + The column count of the last executed query.
* void Close()
  + Closes the database connection and cleans up any resources.
* void ExecuteQuery(string query, int timeout)
  + Executes the given query under the given timeout.
* string GetColumnName(int index)
  + Gets the column name at the given index.
* IList<object> GetRowData()
  + Returns the query data at the current row index.
* bool MoveNext()
  + Moves to the next row index and returns whether or not there are more rows to be read.

# Distribution

To properly distribute your new plugin, you need to make sure you include all required libraries, which includes the Domo provided libraries. This ensures that your plugin runs on the end user’s computer even if the Workbench libraries get updated.