## **Translations Builder Developers Guide**

Blah Blah Blah

Graphical user interface, application

Description automatically generated

Translations Builder is an external tool for Power BI Desktop that has been developed using C#, .NET 6 and Windows Forms. Translations Builder does its work by reading and writing to a dataset definition that’s been loaded into a session of the Analysis Service engine running in Power BI Desktop. Translations Builder uses the Tabular Object Model (TOM) to perform read and write operations.

TOM allows your code to establish a direct connection to the model for a dataset loaded into Power BI Desktop. This provides the most direct approach for writing custom code to automate the process of managing metadata translations within a Power BI dataset.

Diagram

Description automatically generated

The **TranslationsBuilder** programs against TOM to automate adding secondary cultures and metadata translations to a Power BI dataset. The **TranslationsBuilder** application is similar to the Tabular Editor in that it uses the .NET support for [Windows Forms](https://docs.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-5.0) to provide an interactive user experience. You can open and run the **TranslationsBuilder** project using Visual Studio 2022 or Visual Studio Code if you'd like to examine the code inside or test this application running inside the .NET debugger.

The high-level structure of the **TranslationsBuilder** project is shown in the following screenshot. **TranslationsBuilder** is a desktop application loads its primary form named **FormMain** at application startup. In addition to **FormMain**, there are also a few other forms in the project that are used as dialogs to interact with the application user.

Graphical user interface, application

Description automatically generated

The **TranslationsBuilder** project has been created using the [external tools integration support](https://docs.microsoft.com/en-us/power-bi/transform-model/desktop-external-tools) for Power BI Desktop. Once the **TranslationsBuilder** application has been deploy on a Windows computer, you can launch it directly from Power BI Desktop using the **External Tools** tab in the ribbon.

You can deploy **TranslationsBuilder** as an external tool on a computer where you've already installed Power BI Desktop. You deploy an external tool by copying a JSON file with an extension of **pbitool.json** into a well-known folder location which is inspected by Power BI Desktop at startup. The **TranslationsBuilder** project contains a JSON deployment file named **translationsbuilder.pbitool.json**.

A picture containing graphical user interface

Description automatically generated

Let's walk through the steps to deploy **TranslationsBuilder** as an external tool for Power BI Desktop, First, you must build the project to generate the executable program file named **TranslationsBuilder.exe**. Next, you must update the **path** property in **translationsbuilder.pbitool.json** so it points to the location path for **TranslationsBuilder.exe**. Third, you must copy the file named **translationsbuilder.pbitool.json** to the **External Tools** folder located at the following path.

C:\Program Files (x86)\Common Files\Microsoft Shared\Power BI Desktop\External Tools

The **External Tools** folder might already contain other deployment files for other external tools such as Tabular Editor.

Once you have copied the **translationsbuilder.pbitool.json** file to the **External Tools** folder, the **TranslationsBuilder** tile should appear in Power BI Desktop on the **External Tools** tab in the ribbon. Note that you must restart Power BI Desktop before you can see the effect. After a restart, you should see **TranslationsBuilder** in the **External Tools** tab in the ribbon.

Graphical user interface, text

Description automatically generated

When you launch an external tool like **TranslationsBuilder**, the application is passed startup parameters including a connection string which can be used to establish a connection back to the dataset that's loaded in Power BI Desktop. This allows **TranslationsBuilder** to display dataset information and to provide commands to automate adding metadata translations. **TranslationsBuilder** also provides a table grid down below which displays all the non-hidden dataset objects and their associated metadata translations.

Graphical user interface

Description automatically generated

### Program with TOM in an External Tool

**TranslationsBuilder** is a developer sample created to demonstrate programming metadata translations using TOM. The **TranslationsBuilder** project contains a C# class named [TranslationsManager](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs) which contains the custom C# code which programs with TOM to create secondary cultures and to add metadata translations.

A picture containing diagram

Description automatically generated

Whenever you open a PBIX file with a dataset definition, Power BI Desktop launches a local instance of Analysis Services. The programming model of TOM provides a top-level **Server** object which you can think of as your connection to a local instance of the Analysis Services. The **Server** object provides a **Databases** collection of **Database** objects. Each **Database** object has a **Model** property which provides access to tables, cultures and metadata translations.

Chart

Description automatically generated

Each time Power BI Desktop starts a local instance of Analysis Service to load a dataset, it assigns what seems like a random port number (e.g. **50001**). You can construct a connection string using a path of **localhost** followed by the port number in a format that follows the pattern of **localhost:50001**. Once you know the port number of the local instance of Analysis Services, you can connect to the dataset loaded in that instance using the following code.

// connect to local Analysis Services engine running in Power BI Desktop session

Server server = new Server();

server.Connect("localhost:50001");

// With Power BI Desktop, server will contain extactly one database

Database dataset = server.Databases[0];

Model model = server.Databases[0].Model;

When using TOM to connect to Power BI Desktop, the **Databases** collection will always contain a single **Database** object. Therefore, you can access the **Database** object representing a dataset loaded into Power BI Desktop using **Datasets[0]**. Things are different when using TOM to connect to a workspace in the Power BI Service. In this case the **Server** object represents a Power BI workspace which can have more than one dataset. In this case, the **Datasets** collection can have more than one dataset so the syntax of **Datasets[0]** cannot be used reliably. Instead, you must access datasets by name.

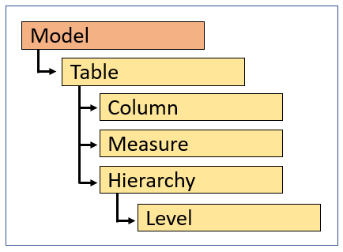
Now let's examine how an external tool like **TranslationsBuilder** is able to obtain the connection string to connect to the **Server** object. When Power BI Desktop launches an external tool, it passes startup parameters which a connection string for the **Server** object and the **Database** name which is passed as a GUID.

Since **TranslationsBuilder** has been deployed as an **External Tool**, the application can examine its startup parameters to dynamically discover the **Server** connect string. The **TranslationsManager** class provides a [static constructor](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs#L19) that uses properties of the **AppSettings** class to initialize the a **Server** object and a **Model** object for the dataset that's open in the current session of Power BI Desktop.

Graphical user interface, text

Description automatically generated

Once you acquire a reference to the **Model** object, it can be used to access dataset objects of type **Table**, **Column**, **Measure**, **Hierarchy** and **Level**. The **Model** contains a **Tables** collection. Each **Table** object contains three collections named **Columns**, **Measures** and **Hierarchies**. A **Hierarchy** object contains a **Levels** collection with **Level** objects.



You can enumerate through the **Tables** collection of a **Model** object to discover what tables exist in a dataset definition. Within a **Table** object, you can further enumerate through three collections named **Columns**, **Measures** and **Hierarchies**. With a **Hierarchy** object, you can enumerate through the **Levels** collection to access **Level** objects.

foreach (Table table in model.Tables) {

// (1) enumerate through tables one by one

Console.WriteLine("Table: " + table.Name);

// (2) enumerate through columns

foreach (Column column in table.Columns) {

Console.WriteLine("Column: " + column.Name);

};

// (3) enumerate through measures

foreach (Measure measure in table.Measures) {

Console.WriteLine("Measure: " + measure.Name);

};

// (4) enumerate through hierarchies

foreach (Hierarchy hierarchy in table.Hierarchies) {

Console.WriteLine("Hierarchy: " + hierarchy.Name);

// (5) enumerate through hierarchy levels

foreach(Level level in hierarchy.Levels) {

Console.WriteLine("Hierarchy: " + level.Name);

}

};

}

It is important that you understand how to enumerate tables, columns, measures, hierarchies and levels in this fashion when you need to add metadata translations. That's because **Table**, **Column**, **Measure**, **Hierarchy** and **Level** objects are the primary types of dataset objects which will require you to add metadata translations.

The **TranslationsBuilder** application has been designed to store and retrieve persistent application settings using the [Application Settings support in Windows Forms](https://docs.microsoft.com/en-us/dotnet/desktop/winforms/advanced/application-settings-for-windows-forms?view=netframeworkdesktop-4.8). The **TranslationsBuilder** application provides a top-level **AppSettings** class with a set of static properties as shown in the following screenshot. If you are curious, you can examine the implementation of these static properties in the [AppSettings](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/AppSettings.cs#L10) class to see how to store and retrieve persistent property settings in a desktop application created using .NET.

Graphical user interface, text, application

Description automatically generated

The **Server** property and **Database** property are used to track the target dataset to open. There are two other properties named **TranslationsOutboxFolderPath** and **TranslationsInboxFolderPath** which allow the user to customize where translation files are stored in the file system of the local computer. There are also two addition properties named **AzureTranslatorServiceKey** and **AzureTranslatorServiceLocation** used to store credentials for communicating with the Microsoft Translator service. The purpose of all of these properties in the **AppSettings** class will be explained in later sections of this article.

The **TranslationsBuilder** application makes it possible for a user to modify any of these application settings by invoking the **Configure Settings…** command from the **Settings** menu to display the **Configuration Options** dialog.

Graphical user interface, text, application

Description automatically generated

The **Configuration Options** dialog interacts with the user using a simple form which can be used to view or update any of the application settings. There is also a **Save Changes** button which allows the user to persist their changes locally.

Application

Description automatically generated with medium confidence

### Use Annotations to Store Custom Property Values

Before diving further into programming metadata translations, it's time to examine using the [Annotations feature](https://docs.microsoft.com/en-us/dotnet/api/microsoft.analysisservices.tabular.annotation?view=analysisservices-dotnet) made available through TOM. Let's begin by asking a simple question. ***What is an annotation?***

You can think of an annotation as a custom property or custom tag that you add into a dataset definition. You can add an annotation to a dataset object such as a **Table**, **Column**, **Measure**, **Hierarchy** or **Level**. You can also add an annotation directly to the **Model** object if you'd like to create a custom property that is tracked at the dataset level. Here's an example of creating a new **Annotation** object and adding it to the **Annotations** collection of the **Model** object.

Annotation annotation1 = new Annotation { Name = "MyFirstAnnotation", Value = "Hello Annotations" }

model.Annotations.Add(annotation1);

model.SaveChanges();

As you can see, an annotation is really just a name/value pair that can be added to a dataset object to create a custom property. The use of annotations can lead to innovative designs where you're able to track whatever custom metadata you need within a dataset definition. The **TranslationsBuilder** application demonstrates the use of annotations by adding support to track a friendly dataset name so users do not have to rely upon dataset names in the form of GUIDs.

If you examine the code inside the **TranslationsManager** class, you can see it contains a const string field named [DatasetAnnotationName](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs#L27) and a static property named [DatasetName](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs#L29) as shown in the following code listing.

const string DatasetAnnotationName = "FriendlyDatasetName";

public static string DatasetName {

get {

if (model.Annotations.Contains(DatasetAnnotationName)) {

return model.Annotations[DatasetAnnotationName].Value;

}

else {

return model.Database.Name;

}

}

set {

if (model.Annotations.Contains(DatasetAnnotationName)) {

model.Annotations[DatasetAnnotationName].Value = value;

}

else {

model.Annotations.Add(new Annotation { Name = DatasetAnnotationName, Value = value});

}

model.SaveChanges();

}

}

Now that you have seen the completed implementation of the **DatasetName** property, let's review the code in the **get** block and the **set** block so you can see the general pattern involved when programming annotations with TOM.

The **get** block of the **DatasetName** property checks to see if an annotation with a name of **FriendlyDatasetName** has already been created. If the annotation exists, the property returns the **Value** property of the **Annotation** object. If there is no **Annotation** with the name of **FriendlyDatasetName**, the property implementation falls back on the **Name** property of the **Database** object which is typically tracked using the less-than-friendly GUID value.

get {

if (model.Annotations.Contains(DatasetAnnotationName)) {

return model.Annotations[DatasetAnnotationName].Value;

}

else {

return model.Database.Name;

}

}

The **set** block of the **DatasetName** property has been written to update the annotation value. As you can see from the following code, you must first determine whether the annotation already exists. If the annotation exists, you update its **Value** property. If the annotation does not yet exist, you must create a new **Annotation** object and add it to the **Annotations** collection of the **Model** object.

set {

if (model.Annotations.Contains(DatasetAnnotationName)) {

model.Annotations[DatasetAnnotationName].Value = value;

}

else {

model.Annotations.Add(new Annotation { Name = DatasetAnnotationName, Value = value});

}

model.SaveChanges();

}

**IMPORTANAT**: Calling **SaveChanges** only updates the dataset loaded into memory in a session of Power BI Desktop. You must go to Power BI Desktop and save your work there for the changes to be persisted to the underlying PBIX file.

When you add an annotation in this fashion, the annotation becomes part of the metadata for the dataset definition. The following screenshot shows how this annotation is serialized within the **BIM** file for a dataset definition.

Text, letter

Description automatically generated

From this screenshot, you can see that Power BI Desktop uses annotations to track its own custom properties within a dataset definition with annotations such as **PBIDesktopVersion**, **PBI\_QueryOrder** and **\_\_PBI\_TimeIntelligenceEnabled**.

Now, let's examine how using an annotation like this can improve the user experience of an external tool. When you first open a dataset in the **TranslationsBuilder** application, the **Dataset** name will be displayed as a GUID value.

Graphical user interface, application

Description automatically generated

The vast majority of humans have trouble memorizing or recognizing GUID values and would generally prefer more friendly names. Therefore, the **TranslationsBuilder** application provides the ability to replace the GUID for the dataset name with a more friendly name. To change the dataset name, you can invoke the **Set Friendly Dataset Name** command from the **Settings** menu to display the **Set Dataset Name** dialog.

Graphical user interface, text, application

Description automatically generated

When promoted by the **Set Dataset Name** dialog, you can add a friendly dataset name and then click **Save**.

Rectangle

Description automatically generated with low confidence

When you click **Save**, the **TranslationsBuilder** application will save the **Name** value as an annotation and display this friendly name instead of the GUID.

Graphical user interface, application

Description automatically generated

Remember TOM code can update the dataset loaded into memory in Power BI Desktop. However, it cannot persist the changes to the underlying PBIX file. You must save your changes to in Power BI Desktop to update the PBIX file.

As you will see later, the **TranslationsBuilder** application uses the dataset name when it generates file names during the translation export process. This makes it possible to generate files with more meaningful names like this.

ProductSales-Translations-Spanish.csv

You can see why this is better than generating file names with GUIDs that looks like this.

e4287640-d84c-4cb9-abce-443568c5be64-Translations-Spanish.csv.

Let's summarize what you have learned in this section. Annotations provide a great way to track custom properties inside a dataset definition. In the **TranslationsBuilder** application, annotations are used to track a friendly dataset name. However, annotations can also be used to track custom property values for objects of type **Table**, **Column**, **Measure**, **Hierarchy** or **Level**. Your takeaway should be that annotations provide a very powerful design dimension that is really only limited by your imagination.

### Add Secondary Cultures to a Dataset

The set of dataset objects that support adding metadata translations include **Table** objects, **Column** objects, **Measure** objects, **Hierarchy** objects and **Level** objects. With dataset objects, you can add translations using three different properties which include the **Caption** property, the **Description** property and the **DisplayFolder** property. You can think of the **Caption** property of a dataset object as the display name which is seen by users.

Before you can begin to add metadata translations to a dataset definition, you must understand the purpose of **Culture** objects. In a new dataset definition, there will be a single **Culture** object based on a default language and a default locale. In the TOM programming model, the **Model** object contains a text-based **Culture** property with the name of the default culture and a **Cultures** collection which contains a set of one or more **Culture** objects.

A picture containing graphical user interface

Description automatically generated

The **ProductSales.pbix** developer sample has default culture name of **en-US**. If you want to determine the default culture name of a dataset, you can examine the **Culture** property of the **Model** object.

string defaultCultureName = model.Culture;

Let's say you need to obtain a reference to the **Culture** object for the default culture. You can write code to access the default **Culture** object by accessing it through the **Cultures** collection using the **Culture** property value as an index key.

Culture defaultCulture = model.Cultures[model.Culture];

In a new dataset definition, the **Cultures** collection contain a single **Culture** object based on the default culture. You must explicitly add new **Culture** objects to the **Cultures** collection to support secondary languages. Here is an example of adding a secondary culture to support adding metadata translations in Spanish.

model.Cultures.Add( new Culture { Name = "es-ES" } );

model.SaveChanges();

Remember that calling **SaveChanges** doesn't update the PBIX project file. After making changes to a dataset using an external tool such as **TranslationsBuilder**, you still need to return to Power BI Desktop and save your changes there. OK, now you’ve been told several times to save your changes in Power BI Desktop. This is your last warning.

The **TranslationsBuilder** application allows the user to add secondary cultures. To accomplish this, click the **Add Culture** button to display the **Add Culture** dialog. Next, select a culture from the list and click the **Add Culture** button.