Building Multi-language Reports in Power BI

Power BI provides Internationalization and localization features which make it possible to build multi-language reports. For example, you can design a Power BI report that renders in English for some users while rendering in Spanish, French, German or Dutch for other users. If a company or organization has the requirement of building Power BI reports that support multiple languages, it's no longer necessary to clone and maintain a separate PBIX project file for each language. Instead, they can increase reuse and lower report maintenance by designing and implementing multi-language reports.

Updated: July 7, 2021

This document is a work in progress and should be considered a draft and not a final version

Contents

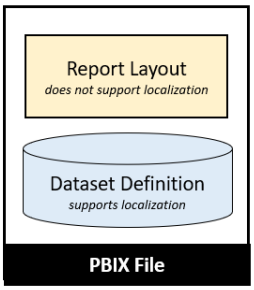
Overview of Multi-language Report Design

Power BI provides the features required to design and implement multi-language reports. However, the path to success is not overly intuitive. The purpose of this article is to explain how to use the Power BI features for Internationalization and localization from the ground up and to provide the guidance for building reports that support multiple languages.

The primary feature in Power BI used to build multi-language reports is known as **metadata translations**. Power BI inherited this feature from its predecessor, Analysis Services, which introduced metadata translations to add localization support to the data model associated with a tabular database or a multidimensional database. In Power BI, metadata translations support has been integrated at the dataset level.

A metadata translation represents the property for an object in a data model that's been translated for a specific language. Consider a simple example. If your data model contains a table with an English name of **Products**, you can add translations for the **Caption** property of this table object to provide alternative names for when the report is rendered in a different language. The object types in a Power BI dataset that support metadata translations include tables, columns, measures and hierarchies. In addition to the **Caption** property which tracks an object's display name, dataset objects also support adding metadata translations for two other properties which are **Description** and **DisplayFolder**.

Keep in mind that the Power BI support for metadata translations only applies to dataset objects. The Power BI does not support adding translations for text values stored as part of the report layout. Think about a common scenario where you add a textbox or a button to a Power BI report and then you type in literal text for a string value displayed to the user. That text value is stored in the report layout and cannot be localized. Therefore, you must avoid using textboxes and buttons that include literal text when designing multi-language reports. As a second example, page tabs in a Power BI report are also problematic because their display names cannot be localized. Therefore, you must design multi-language reports so that page tabs are never displayed to the user.



A little later in this article you learn about the nitty-gritty details of building multi-language reports in Power BI Desktop. At this point, however, it's possible to make a high-level observation. For someone experienced with report building in Power BI Desktop, the challenge of learning how to build multi-language reports isn't as much about learning ***what to do*** but more about learning ***what not to do***. There are lots of popular Power BI report design techniques that cannot be localized and therefore cannot be used when building multi-language reports.

Before you dive into the details of building multi-language report, there are two important issues to consider. These issues are important because they could become serious show stoppers if you are attempting to follow the guidance in this article. The first issue is that multi-language reports must run in a dedicated capacity. That means you must be working with either Power BI Premium or the Power BI Embedded service in Microsoft Azure. Multi-language reports will not work correctly when loading into a Power BI workspace in a shared capacity.

The second issue is that Power BI Apps don't support localization and can't be used to distribute multi-language reports. That means you will have to find an alternative approach to making multi-language reports accessible to users. This isn't a problem in a PaaS scenario when you're developing with Power BI embedding. In a SaaS scenario where licensed users are accessing reports using the Power BI service, you can use an alternative to Power BI Apps such as report sharing, direct workspace access or secure embed.

Metadata Translations versus Content Translations

Metadata translations are used to localize the text for dataset object property values such as the names of tables, columns and measures. While metadata translation help to localize the names of tables and columns, they don't offer any assistance when it comes to localizing text values in the data itself. If your dataset has a **Products** table, how do you localize the text-based product names that exist in the individual rows of the **Products** table?

While adding metadata translations to your dataset is an essential first step, it doesn't always provide a complete solution by itself. A complete solution might require not only localizing the names of tables and columns, but also localizing the text-based content stored in table rows such as product names, product categories and country names. The key point is that the use of metadata translations must sometimes be complimented by a data model designed to support **content translation**.

While every multi-language report will require metadata translations, it's is not as clear whether they will also require content translations. Imagine a scenario where you are developing a report template for an application that has a well-known database schema such as Dynamics 365. Now let's say that some customers maintain their Dynamics 365 database instance in English while other maintain theirs in other languages such as Spanish, French and German. The important observation is that for any specific customer, only one language is used in their database instance. Therefore, there is no need to implement content translations when no database instance ever needs to be viewed in multiple language.

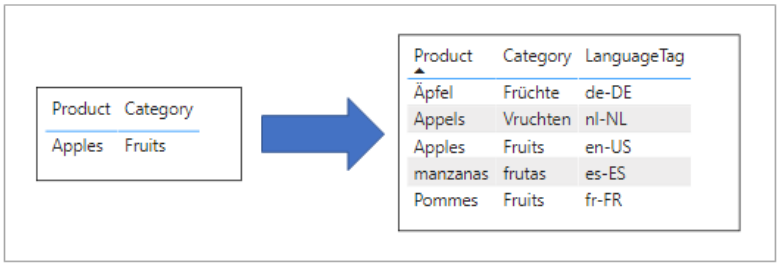
Now think about different scenario on which the Product Sales developer sample was built. The scenario includes a single database instance that contains data about the sales performance across several European countries. The scenario also includes the requirement to display a report in different languages while the data is all coming from a single database instance. This is the typical scenario where you also need to implement content translations.

Long before Microsoft introduced Power BI, software developers around the world have been building multi-language applications that support content translation. After two decades of designing and refining various database designs, several common design patterns have emerged as industry best practices to support content translation. Some of these design patterns involve adding a new table column for each language while other design patterns involve adding a new table row for each language. Each approach has benefits and drawbacks when compared to the other.

Currently, there is a limitation with DAX and the VertiPaq engine which makes it impractical to implement a content translation scheme based on adding a column for each language. The specific limitation is that calculated columns are evaluated at load time and do not yet support dynamic evaluation. While Microsoft has plans to update DAX and the VertiPaq engine to support calculated columns with dynamic evaluation, there is currently no timeline for when this feature will be available in Preview or when it will reach GA.

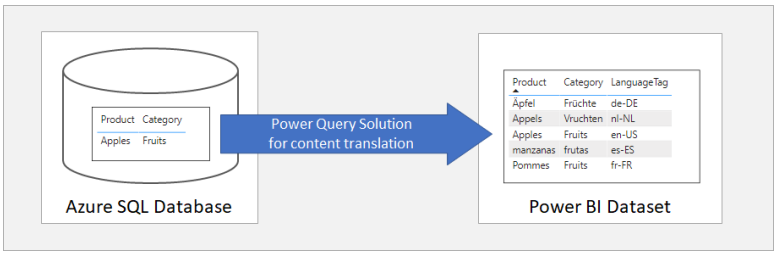
Until dynamic column support is added, it doesn't make sense to implement a content translation scheme based on adding a column for each language. Currently, you would have to use measures instead of columns but that is very limiting because measures do not have row context. For example, measures cannot be used to supply values for the axes in a bar chart or line chart. Furthermore, measures cannot be used in the data roles of a legend or to filter data using slicers.

Currently, the best design pattern to implement content translation in a Power BI solution is row replication. Consider a simple example of using the row replication to implement content translation in a Power BI dataset. Let's say the **Products** table contains two text columns named **Product** and **Category** and you'd like your report to support five different languages including English, Spanish, French, German and Dutch. For each product in the **Products** table, you need to generate 5 records where each record contains the product name and product category translated to a specific language. Whenever the report is loaded, a row filter is applied to the **LanguageTag** column so that users only see the rows for one of the supported languages at a time.

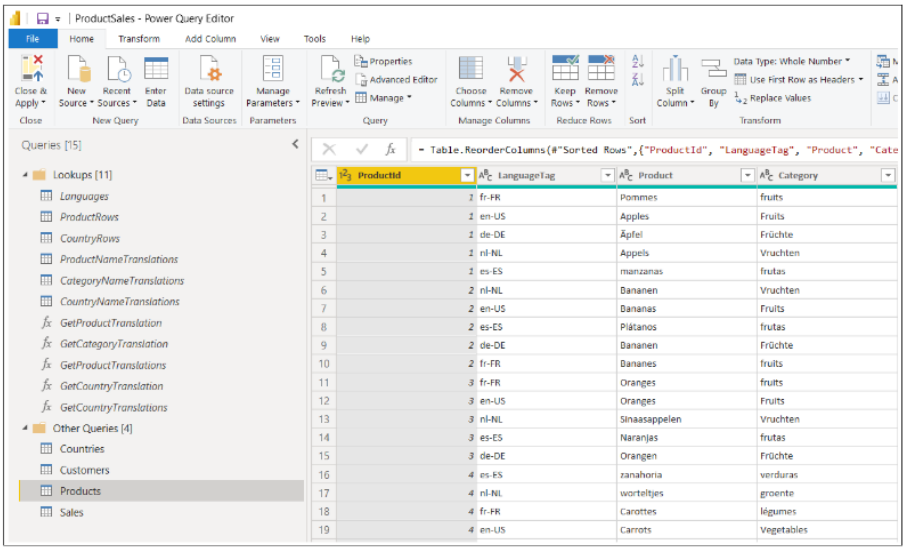


When using the row replication design pattern in Power BI, you must somehow generate the extra rows which add the translated content into each table with one or more text-based columns requiring translation. Many multi-language applications add support for content translation at the database level and use some type of ETL process to generate the extra rows containing the translations. It's relatively easy to build a multi-language report in Power BI if the underling database has already been designed to support content translations.

This article is accompanied by developer sample named **ProductSales.pbix**. This sample demonstrates how to design a content translations solution using Power Query with an import-mode dataset. The solution is implemented using logic written in M which uses translation lookup tables to generate the extra rows when importing data from an Azure SQL database. This type of design allows you to create multi-language reports with content translations without having to make any changes to the underlying database. Instead, you can package all the ETL logic for content translations as query logic inside a PBIX template file.



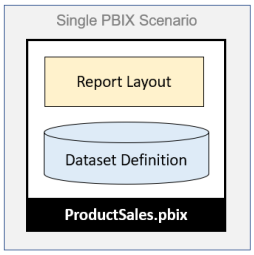
While implementing a content translation strategy with Power Query isn't always the right choice, it's great for scenarios where you don't have either the authority or the time it takes to add content translations support at the database level. If you do decide to use this strategy, you'll find the writing Power Query logic in the M programming language provides developers with a very elegant way to replicate rows with translated content during a dataset refresh operation. You can review the entire Power Query solution in the **ProductSales.pbix** developer sample.



Multi-language Report Development Process

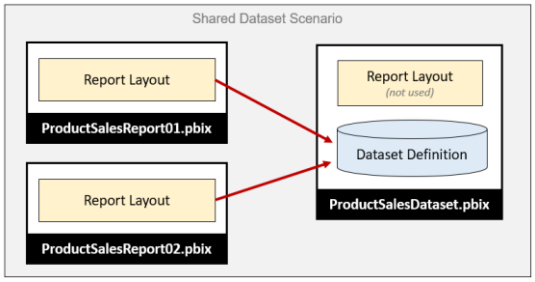
Now that you understand high-level concepts of building multi-language reports, it's time to move forward and discuss how to structure the multi-language report development process. The first step here is to decide how to package your dataset definitions and report layouts for distribution. Let's examine three popular approaches that are commonly used by Power BI customers.

In the first approach, the goal is to keep things simple by creating a single PBIX project file which contains both a dataset definition and a report layout. You can easily deploy this solution by importing the PBIX project file into a Power BI workspace. If you need to update the dataset definition or the report layout after they have been deployed, you can perform an upgrade operation by importing an updated version of the PBIX project file.

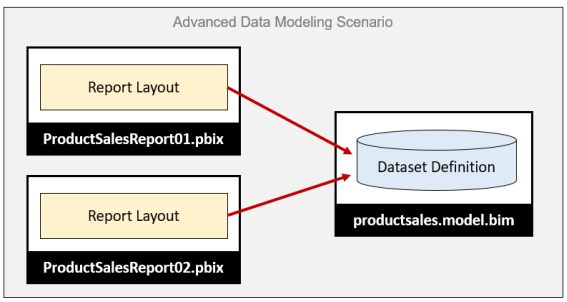


The single PBIX file approach doesn't always provide the flexibility you need. Imagine a scenario where one team is responsible for creating and updating dataset definitions while other teams are responsible for building reports. For a scenario like this, it makes sense to split out dataset definitions and report layouts into separate PBIX project files.

To use the shared dataset approach, you create one PBIX project file with a dataset definition and an empty report which remains unused. Once this dataset has been deployed to the Power BI Service, report builders can connect to it using Power BI Desktop to create report-only PBIX files. This makes it possible for the teams building reports to build PBIX project files with report layouts which can be deployed and updated independently of the underlying dataset.



While many customers use Power BI Desktop to create and maintain Power BI dataset definitions, it's not the only option. There is an alternative approach which makes it possible to take advantage of advanced data modeling features not available through Power BI Desktop. For example, you can use the Tabular Editor to create and maintain dataset definitions which can be saved in a JSON-file format with a .bim extension as shown in the following diagram.



From the perspective of adding multi-language support to a Power BI solution, it really doesn't matter which of these three approaches you choose. The techniques and disciplines used to build multi-language reports remain the same whether you decide to build your solution using a single PBIX project file or with a combination of PBIX project files and a BIM dataset definition file. There are specific tasks you need to perform at the dataset level and other tasks you must perform when building report layouts in Power BI Desktop.

The multi-language report development process can be broken down into a few distinct phases. Each of these phases will be examined in detail in this article.

1. Add a localized label table to the dataset definition
2. Prepare the report layouts for localization
3. Add metadata translations to the dataset definition
4. Design and implement a content translation strategy (if required)

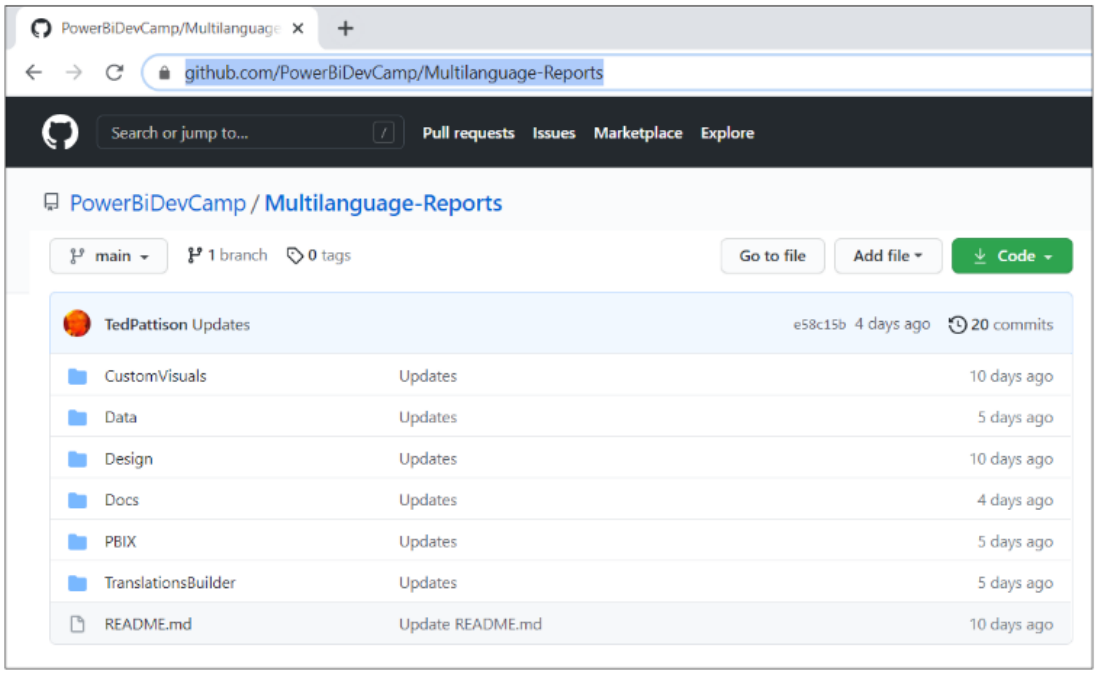
The ProductSales.pbix Developer Sample

This article is accompanied by a developer sample based on a single PBIX file solution named **ProductSales.pbix**. This developer sample demonstrates building a multi-language report for Power BI which supports English, Spanish, French, German and Dutch. While this developer sample is based on a single PBIX project file approach, you should be able to use the same concepts and techniques to build multi-language reports in scenarios where your solution contains multiple PBIX files and optionally a BIM dataset definition file.

The PBIX project files for this developer solution and all the supporting resources used to build them are available to download or view online from a GitHub repository at the following URL:

https://github.com/PowerBiDevCamp/Multilanguage-Reports

The **Multilanguage-Reports** GitHub repository contain several different versions of the **ProductSales.pbix** project file so you can see what the PBIX project looks like at various stages of the multi-language report development process. This GitHub repository also contains a few other development projects including a Power BI custom visual named **LocalizedLabel** and a C# application named **TranslationsBuilder** which demonstrates how to create an external tool which can be used alongside Power BI Desktop to automate the process of adding and updating metadata translations.



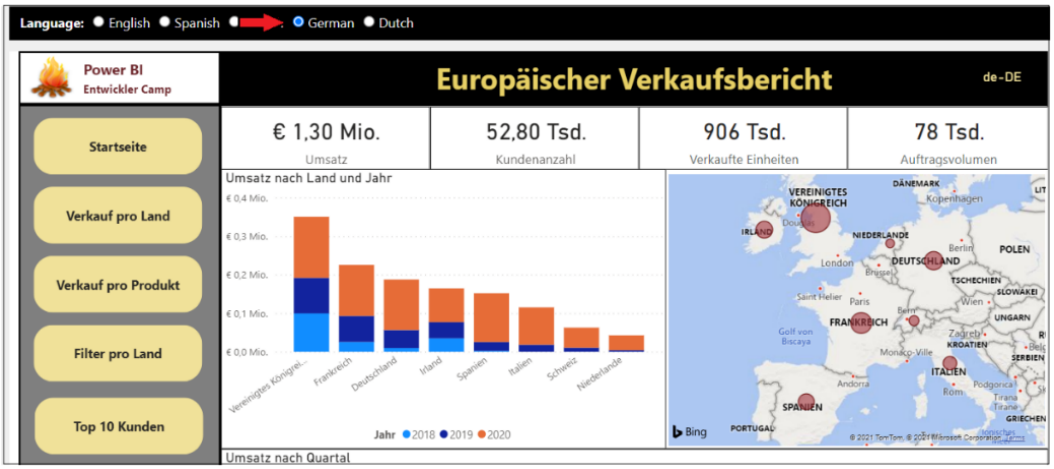
In addition to the project files and resources available in the GitHub repository, this developer sample also includes a live version which you can view to experience the completed multi-language report in action.

https://multilanguagereportdemo.azurewebsites.net

If you navigate to the live version of the report in a browser, you will see the completed solution for **ProductSales.pbix** displayed using it's default language of English. There are buttons in the left-hand navigation with captions such as **Home**, **Sales by Country** and **Sales by Product** which make it possible for the user to navigate from page to page. Experiment by clicking the button in the left-hand navigation to move form page to page.

At the top of the web page above the embedded report, you will see a set of radio buttons that allow you to reload the report using a different language including Spanish, French, German and Dutch.

Experiment by clicking these radio buttons to reload the report in a different langauge. For example, click on the radio button with the caption of **German**. When you do, there is JavaScript behind this page that responds by reloading the report using the language of German intead of English. You can see that all the button captions in the left-hand navigation amd text-based values in the visuals on the page now display their German translations instead of English.



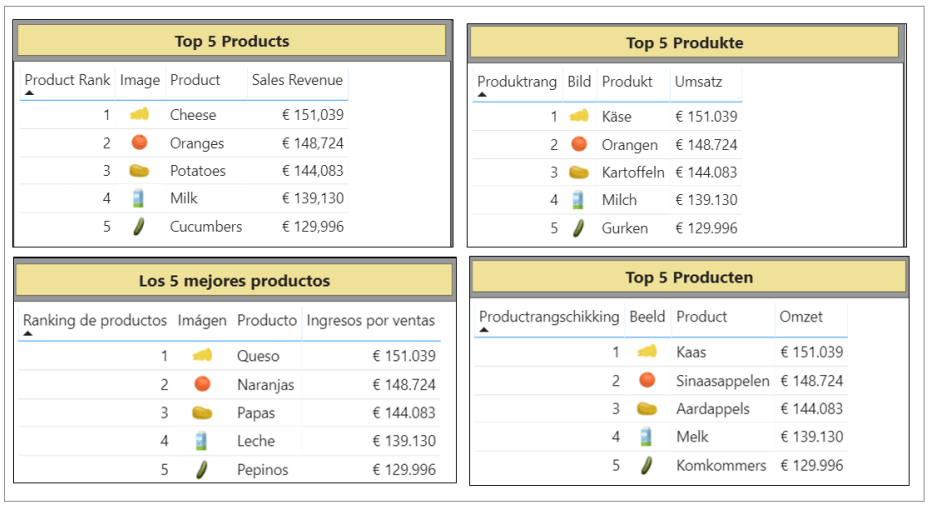
When creating a report for Power BI, it's a common practice to add text-based labels for report elements such as titles, headings and button captions. However, this creates an unexpected bump in the road when building multi-language reports in Power BI Desktop. The problem is that you cannot create labels for a report using the standard approach where you add textboxes and buttons into the report. That's because any text you add for a property value of a textbox or a button is stored in the report layout and, therefore, cannot be localized.

As discussed earlier in this article, the Power BI localization features are supported at the dataset definition level but not at the report layout level. At first you might ask the question "***how can I localize text-based values that are not stored inside the dataset definition***". The answer to this question is that you cannot.

A better question to ask is "***how can I add the text-based values for labels so they become part of the dataset definition***". Once the text-based values for labels become part of the dataset definition, then they can be localized. This leads to an innovative approach of creating a localized labels table. This design technique will be discussed in detail in the next section of this article.

The **ProductSales.pbix** developer sample demonstrates how to create localized labels. As an example, there are button captions in the left navigation that have been localized as well as shown in the following screenshot which shows the left navigation menu for all 5 languages displayed side by side.

The last thing to call out about the **ProductSales.pbix** developer sample is that it demonstrates content translation. With metadata translations, you can see the names of columns and measures change as you switch between languages. Content translation go further to localize the product names in rows of the **Products** table change. The final section of this article will examine how to design a Power BI dataset to support content translations.



Top 5 Products

Product Rank	Image	Product	Sales Revenue
1		Cheese	€ 151,039
2		Oranges	€ 148,724
3		Potatoes	€ 144,083
4		Milk	€ 139,130
5		Cucumbers	€ 129,996

Top 5 Produkte

Produktrang	Bild	Produkt	Umsatz
1		Käse	€ 151.039
2		Orangen	€ 148.724
3		Kartoffeln	€ 144.083
4		Milch	€ 139.130
5		Gurken	€ 129.996

Los 5 mejores productos

Ranking de productos	Imágen	Producto	Ingresos por ventas
1		Queso	€ 151.039
2		Naranjas	€ 148.724
3		Papas	€ 144.083
4		Leche	€ 139.130
5		Pepinos	€ 129.996

Top 5 Producten

Productrangschikking	Beeld	Product	Omzet
1		Kaas	€ 151.039
2		Sinaasappelen	€ 148.724
3		Aardappels	€ 144.083
4		Melk	€ 139.130
5		Komkommers	€ 129.996

Prepare Datasets and Reports for Localization

So far you've learned about essential concepts and background information you'll need to build multi-language reports. Now, it's time to move ahead and discuss the actual development process. We'll start by discussing a few general topics associated with software localization. After that, we'll move on to topics that are specific to Power BI and designing reports in Power BI Desktop.

When it comes to localizing software, there are some universal principals to keep in mind. The first is to plan for localization from the start of any project. It's significantly harder to add localization support to an existing dataset or report that was initially built without any regard for internationalization or localization. This is especially true with Power BI reports because there are so many popular design techniques that do not support localization. You might find that much of the work for adding localization support to existing Power BI reports involves moving backward and undoing the things that do not support localization before you can move forward with design techniques that do support localization.

Another important concept in localization is to plan for growth. A label that's 400 pixels wide when displayed in English could require a much greater width when translated into another language. If you optimize the width of your labels for text in English, you might find that translations in other languages introduce unexpected line breaks or get cut off which, in turn, creates a compromised user experience.

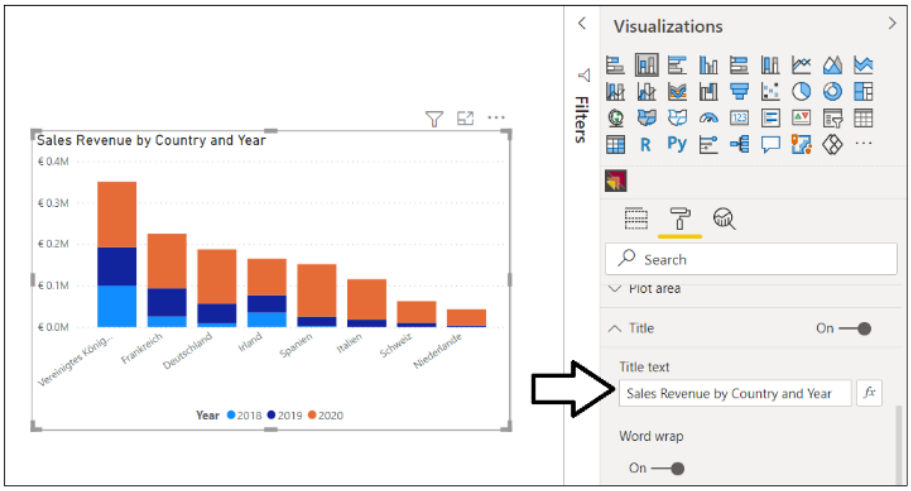
Adding a healthy degree of padding to localized labels is the norm when developing localized software and it's essential that you test your report with each language you plan to support. In essence, you need to ensure your report layout looks the way you expect with any language you have chosen to support.

Avoid Report Design Techniques that do tot Support Localization

If you have experience with Power BI Desktop, it's critical that you learn which report design techniques to avoid when you begin building multi-language reports. Let's begin with the obvious things which cause problems due to a lack of localization support.

- Using textboxes or buttons with literal text
- Adding literal text for the title of a visual
- Displaying page tabs to the user

The key point here is that any literal text that gets added to the report layout cannot be localized. Consider the case where you add a column chart to your report. By default, a Cartesian visual such as a column chart is assigned a dynamic **Title** property which is parsed together using the names of the columns and measures that have been added into the data roles such of **Axis**, **Legend** and **Values**.



There is good news here. The default **Title** property for a Cartesian visual is dynamically parsed together in a fashion that supports localization. As long as you have supplied metadata translations for the names of columns and measures in the underlying data model (e.g. **Sales Revenue**, **Country** and **Year**), the **Title** property of the visual will use the translations for whatever language has been used to load the report. The following table shows how the default **Title** property of this visual is updated from each language supported by the **ProductSales.pbix** developer sample.

Language	Visual Title
English (en-US)	Sales Revenue by Country and Year
Spanish (es-ES)	Ingresos por ventas por país y año
French (fr-FR)	Chiffre d'affaires par pays et année
German (de-DE)	Umsatz nach Land und Jahr
Dutch (nl-NL)	Omzet per land en jaar

Even if you don't like that default visual **Title** property, you must resist replacing it with a literal text value. Any literal text you type into the **Title** property of the visual will be added to the report layout and cannot be localized. Therefore, you should either leave the visual **Title** property with its default value or hide the title so it is not displayed.

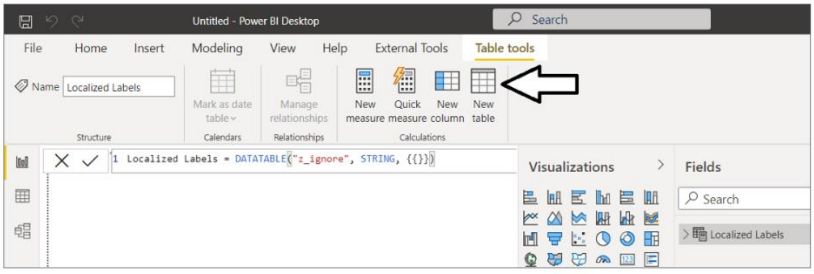
You should also understand that Power BI Apps do not support localization. That's because Power BI Apps are built with a navigation scheme which uses the text captions for page tabs which cannot be localized. Therefore, you must plan to deploy multi-language reports using something other than a Power BI App.

Create the Localized Labels Table

When designing reports, it's a common practice to use text-based labels for report elements such as titles, headings and button captions. You've learned that any text value stored in a report layout cannot be localized. If you want to localize the text-based labels which are displayed on a Power BI report, then those labels must be defined inside the dataset. This leads to the innovative technique of creating a specialized table in the data model for localized labels.

The idea behind the **Localized Labels** table is pretty simple. You can localize the name of any measure inside a dataset. When you need a text label for a report title, you can add a new measure to the **Localized Labels** table and then give the measure a name for the English label such as **European Sales Report**. Since the label is a measure name, you can add translations to supply a localized version of this label for each language.

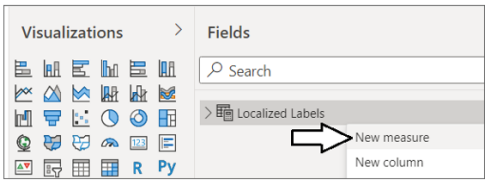
There are several different techniques that can be used in Power BI Desktop to create a new table for localized labels. One quick way to accomplish this is to click on the **New table** button on the **Table tools** tab and then add a DAX expressions with the **DATATABLE** function to create a new table named **Localized Labels**.



Below is the full DAX expressions that creates the **Localized Label** table. The **DATATABLE** function requires that you create a least one column. Therefore the table is created with a single column named **z_ignore** which can be hidden from report view.

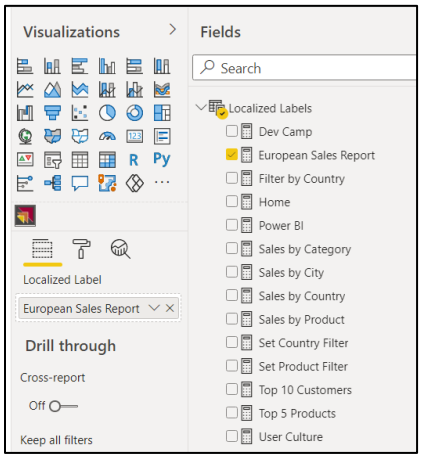
```
Localized Labels = DATATABLE("z_ignore", STRING, {{}})
```

Once you have created the **Localized Labels** table, you can begin to add new measure just as you would in any other Power BI Desktop project.



When creating a measure for a localized label, you can add the label text as the measure name and then set the DAX expression for this measure to a static value of **0**. This value of **0** has no significance and is only added because each measure must be created with a DAX expression.

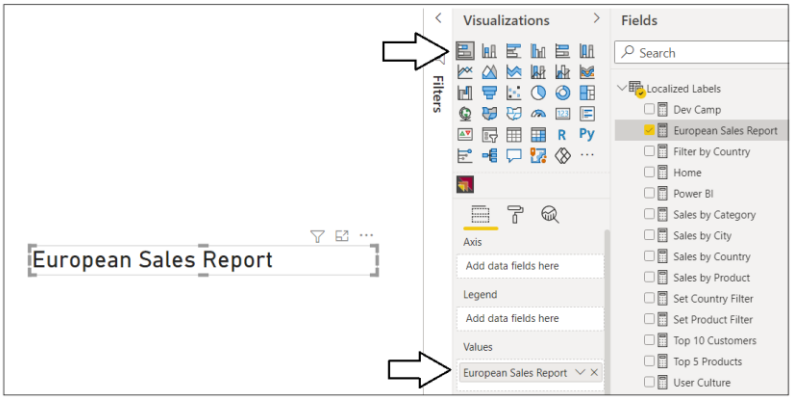
The **ProductSales.pbix** developer sample demonstrates creating a **Localized Labels** table to provide localized labels for all titles, headings and button captions used throughout the report.



Now that you've seen how to create the **Localized Labels** table, it's time to learn how to surface the measure name for a localized label on a Power BI report.

Display Localized Labels using Power BI Core Visuals

The technique used to display measure names from the **Localized Labels** table in a Power BI report is neither straightforward nor intuitive. You can start by adding a Cartesian visual such as the Stacked Barchart visual to a page and then adding the measure for the desired localized label into the **Values** data role. Once you have added the Stacked Barchart visual, you can adjust the visual's width and height so only the visual **Title** property can be seen by the user.



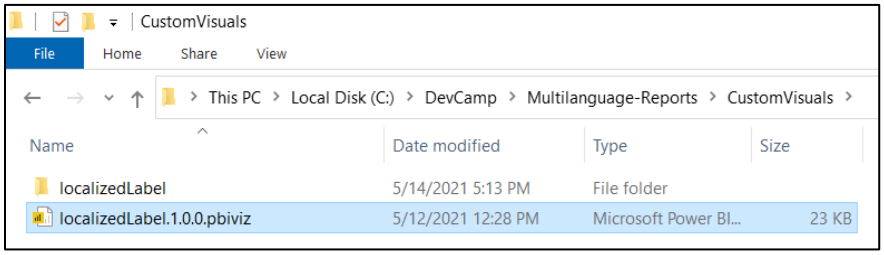
After you have added the visual to a report page to display a localized label, you can adjust the font formatting for the label in the **Title** section of the **Format** pane as shown in the following screenshot.

In summary, you can use the Power BI built-in visuals such as the Stacked Barchart to display localized labels for titles, headings and button captions. However, most report authors find that the design experience is pretty limited when formatting a localized label using any of the Power BI built-in visuals. For example, you cannot configure padding for the label or center its text vertically. In the next section, we will discuss creating a custom visual to provide a much improved design experience for display localized labels on a Power BI report.

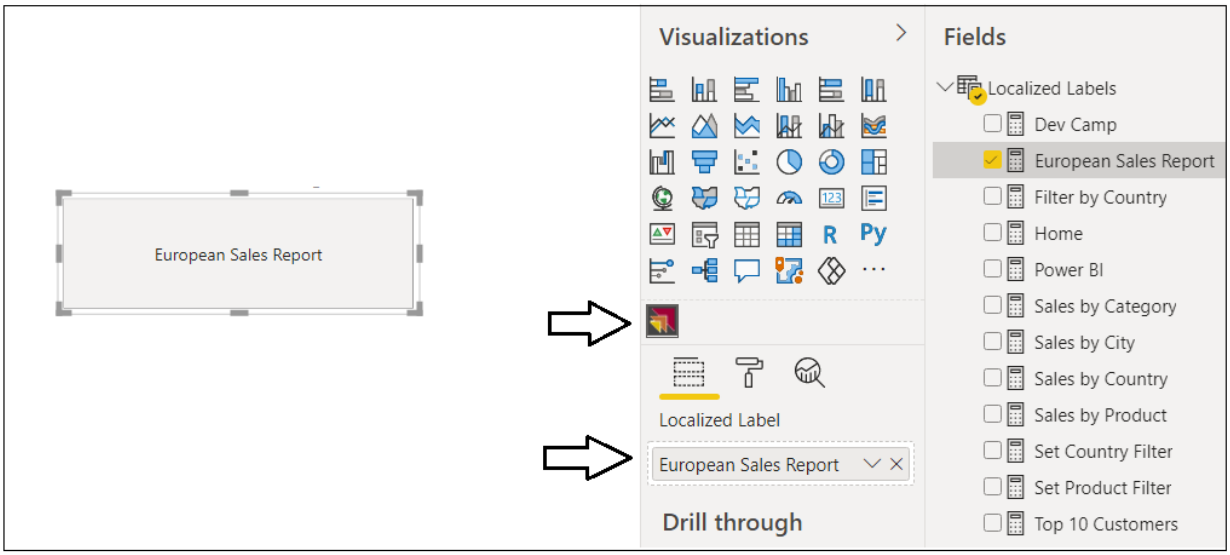
Display Localized Labels using a Custom Visual

The last section discussed using a Stacked Barchart visual to display a localized label. While this technique can be used reliably to create multi-language reports, it is clear that none of the Power BI built-in visuals were designed to support this scenario. The reason this technique works is really a more of a coincidence than a planned outcome. Furthermore, there is extra overhead due to the visual implementation being designed to do far more then just display a title. These issues combine together to provide a motivation to create a custom visual project that is explicitly designed to support the scenario of displaying localized labels in a multi-language report.

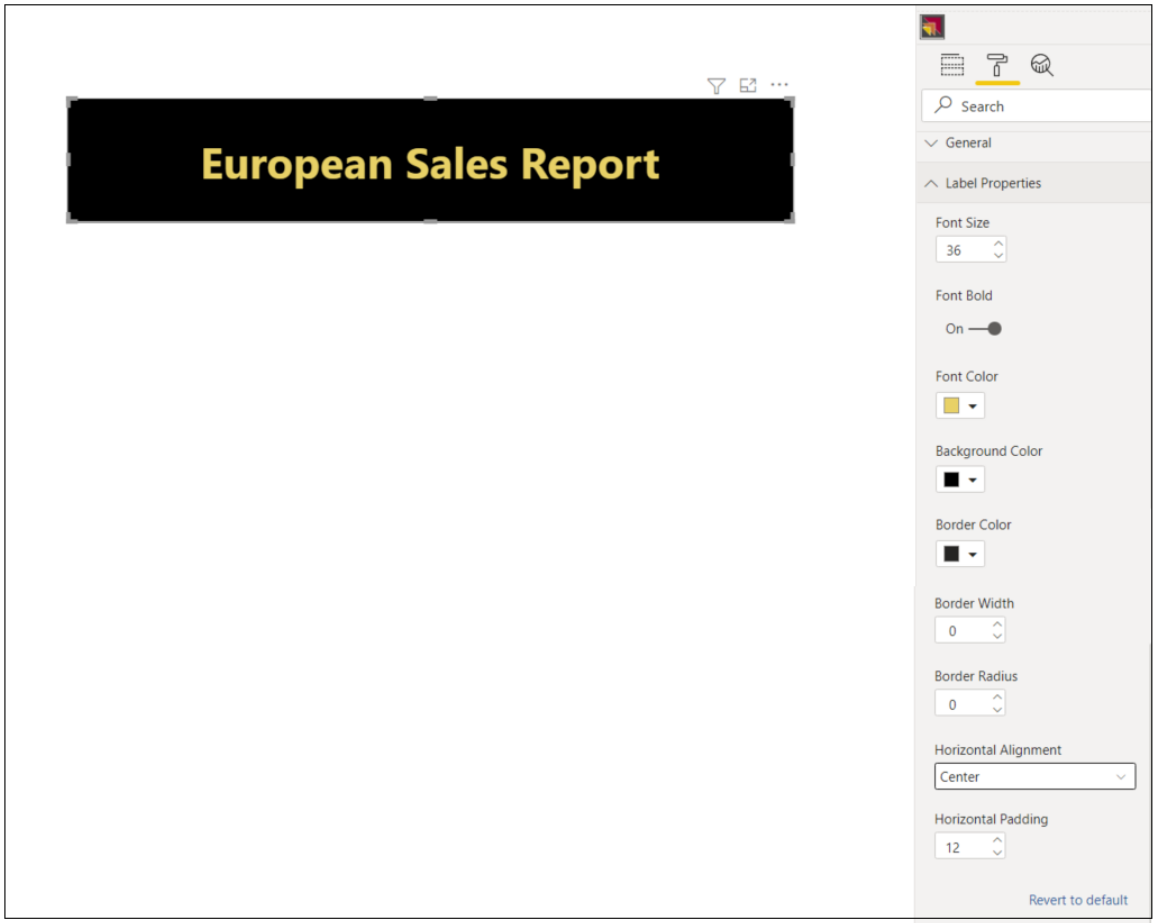
This GitHub repository mentioned earlier contains a CustomVisuals folder. Inside the **CustomVisuals** folder, there is a child folder with a custom visual project named **localizedLabel**. If you have experience with custom visual development, you can open the **localizedLabel** project in Visual Studio Code to see how this custom visual is implemented. The **CustomVisuals** folder also contains a custom visual distribution file for the **localizedLabel** project named localizedLabe.1.0.0.pbiviz. You can import this custom visual distribution file directly into a Power BI Desktop project to begin using this custom visual to display localized labels.



Once you have imported the custom visual distribution file named localizedLabe.1.0.0.pbiviz into a Power BI Desktop project, you should be able to begin using it. After you have added a Localized Label visual instance to a report, you can then populate the **Localized Label** data role using one of the measures in the **Localized Labels** table as shown in the following screenshot.



After you have configured the **Localized Label** data role with the correct measure, you can then configure the font and background formatting of the visual in the **Label Properties** section of the Format Pane.

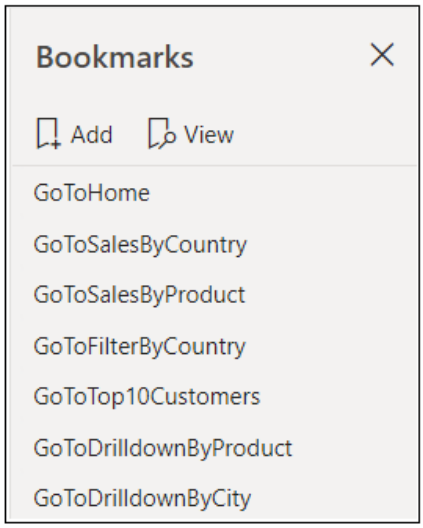


Add Support for Page Navigation

As you recall, you cannot display Power BI report page tabs to the user in a multi-language report because page tabs do not support localization. Therefore, you must provide some other means for users to navigate from page to page. This can be accomplished using a design techniques where you create shapes which act as buttons. When the user clicks on a shape, the shape will apply a bookmark to navigate to another page. Let's step through the process of building a Multi-language navigation scheme using labels from the **Localized Labels** table.

To use this technique, you start by hiding every page in a report except for the first page which acts as the landing page.

Next, create a set of bookmarks. Each bookmark should be created to navigate to specific page. For example, the **ProductSales.pbix** developer sample defines the set of navigation bookmarks shown in the following screenshot.



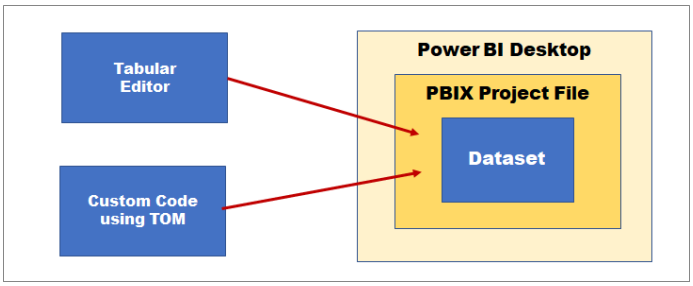
Remember, that you cannot add a button with literal text to a multi-language report. Instead, you must be a bit more creative and use a technique that supports localizing button captions. This can be accomplished by using a shape which is overlaid on top of a localized label visual. The shape should be configured as completely transparent without a border or any background color. The shape should also be configured with an action to trigger one of the bookmarks. The idea is that the user only sees the localized label with the localized button caption but not the shape. When the user clicks the localized label visual, there is a transparent shape on top that applies a bookmark to navigate to another page.

At this point, you've learned how to create the **Localized Labels** table in a dataset and you've learned how to prepare a Power BI report for localization. These are the localization techniques you will continue to use as you create and maintain reports that must support multiple languages. Now it's time to move on and examine the topic of extending a Power BI dataset with metadata translations. Once you learn to do this, you can then add support for whatever languages you need to support in your multi-language reporting solutions.

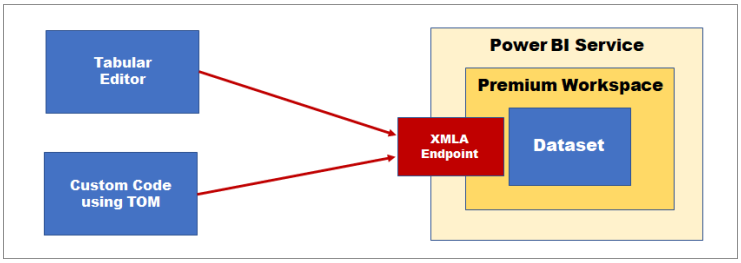
Add Metadata Translations to a Dataset Definition

The previous section discussed how to prepare Power BI datasets and reports for localization. In this section we'll discuss how to extend a dataset definition by adding metadata translations. This is a task you can accomplish by hand using an external tool named **Tabular Editor**. Alternatively, you can write custom code to automate the process of adding metadata translations to Power BI datasets using the **Tabular Object Model (TOM)**. With either approach you can work directly on a PBIX project file that is open in Power BI Desktop. You also have the flexibility to work with datasets defined in .BIM files or live datasets running in Azure Analysis Services (AAS) or the Power BI Service.

While Power BI Desktop doesn't offer direct support for adding and managing dataset translations, it is still often used in the development process. Once you've opened a PBIX project file in Power BI Desktop, the dataset defined inside is loaded into memory and becomes accessible to both the Tabular Editor and to custom code you write using the TOM.



This preceding diagram shows how you can add metadata translations to a dataset loaded into Power BI Desktop. This works well in a scenario where you're building a PBIX project file and preparing to distribute it for the first time. However, You should also understand that it's also possible to access a Power BI dataset directly in the Power BI Service. That means you can connect to a production dataset and manage its metadata translations using the exact same techniques as shown in the following screenshot.



Access to a Power BI dataset running in the Power BI Service is routed through the XMLA endpoint which requires that the hosting workspace is in a dedicated capacity. In other words, you must ensure your workspace has a diamond when viewed in the Power BI Service. This requirement can be met with either Power BI Premium, Power BI Premium Per User or one of the A SKUs for the Power BI Embedded Service in Microsoft Azure.

Install the Tabular Editor

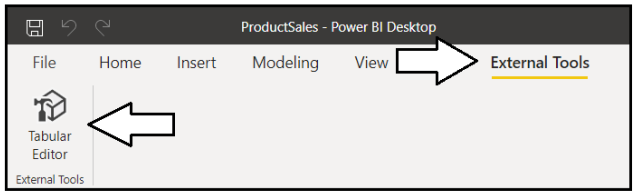
If you plan to work with metadata translations, you should become familiar working with the Tabular Editor. Even if you eventually plan to write your own TOM code to manage translations, working with the Tabular Editor is the best way to get started and the fastest way to learn how translations are stored within the metadata for a dataset. You can download the open source version of Tabular Editor (version 2) for free from the web page at following URL:

https://github.com/otykier/TabularEditor

The Tabular Editor provides first-class support for managing dataset translations, but this is just one areas in which it shines. In addition to its metadata translations support, Tabular Editor provides a comprehensive toolset for advanced data modeling that go far beyond the data modeling support available in Power BI Desktop. Tabular Editor is currently recognized by industry experts as the premiere tool for building and optimizing large-scale Power BI datasets.

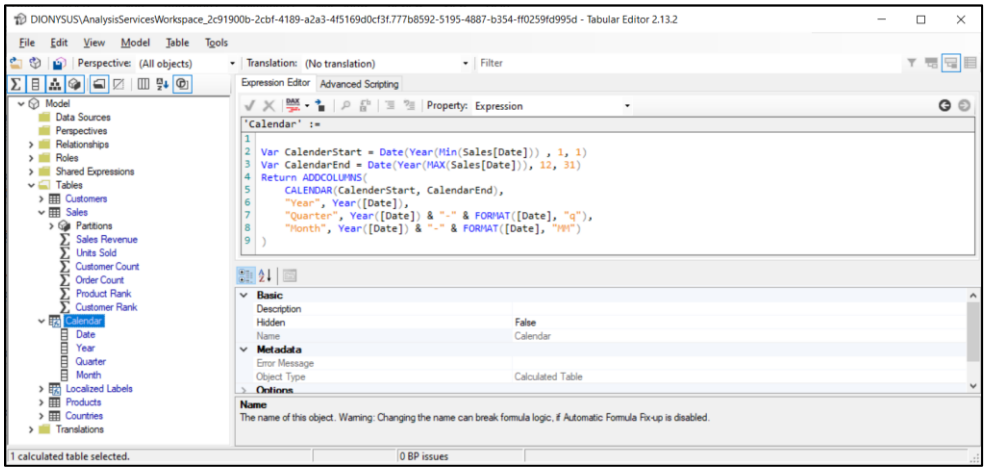
The Tabular Editor was created and continues to be maintained by a very talented developer in the Power BI community named Daniel Otykier. While the open source version of Tabular Editor is available for free, Daniel has recently introduced Tabular Editor version 3 which is based on a paid licensing model. While the free version of Tabular Editor provides all the support you need to add and manage metadata translations, you must purchase Tabular Editor version 3 to take advantage of the newer advanced data modeling and dataset management features that are not available in the free version. You can get more information from the Tabular Editor website at https://tabulareditor.com.

Once you have installed Tabular Editor, Power BI Desktop will display a launch button for it on the **External Tools** tab in the ribbon. Clicking on the **Tabular Editor** button in Power BI Desktop will launch Tabular Editor and automatically open the data model for the current PBIX project file.



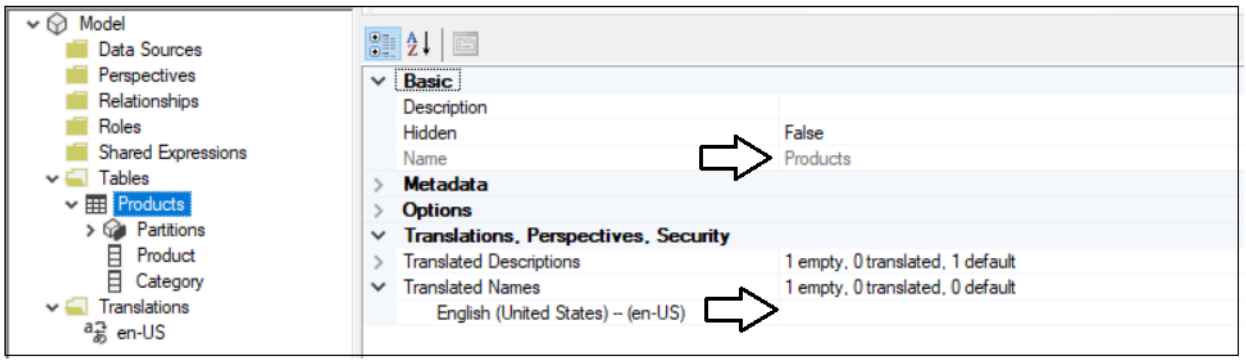
Add Metadata Translations by Hand using Tabular Editor

The Tabular Editor provides an advanced user experience for viewing and modifying dataset objects such as tables, columns and measures. If you expand the **Tables** node in the left navigation, you can select a dataset object and then view its properties in a property sheet on the right. The following screenshot demonstrates selecting a calculated table named **Calendar** in the **ProductSales.pbix** developer sample. Once you have selected an object in the left navigation, you can view or modify any of its properties including the DAX expression used to create the calculated table.

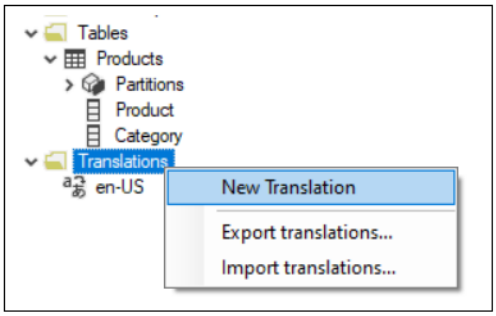


Underneath the **Tables** node in the Tabular Editor, there is another top-level node named **Translations**. This node contains all the **Culture** objects which have been added to the current dataset definition. By default, every new dataset definition contains a single **Culture** object based on a default language and a default locale. The **ProductSales.pbix** developer sample has default **Culture** object based on **English (en)** as its language and the United **States (US)** as its locale. That's why this default culture is tagged with an identifier of **en-US**.

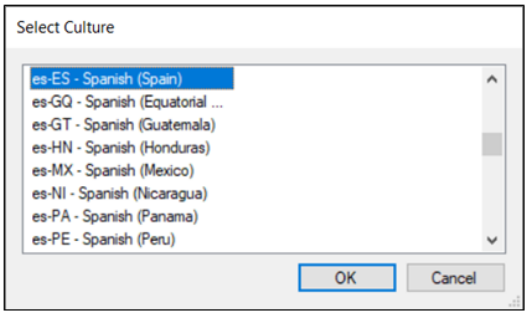
If you examine the property sheet in the Tabular Editor for a dataset object, you will see there is a **Translated Names** section which tracks the value of the **Caption** property. By default, all the translations for the default **Culture** object will have blank values. If you want to add translations manually, you can copy the **Name** property for a dataset object and then paste that text value into the default Culture in the **Translated Names** section.



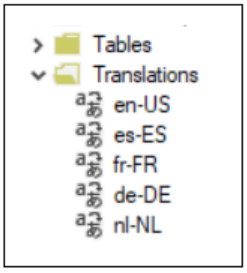
To add support for secondary languages, you must add one or more new **Culture** objects to the dataset definition. This task can be accomplished by right-clicking the **Translations** node and selecting the **New Translation** menu command.



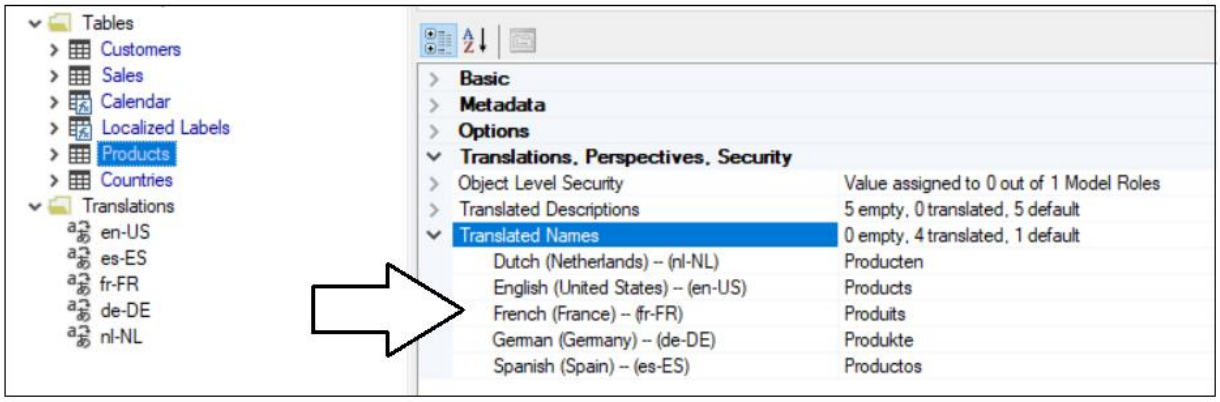
When you invoke the **New Translation** command, you'll be prompted with the **Select Culture** dialog which makes it possible to find and add a new **Culture** object for a specific language and locale.



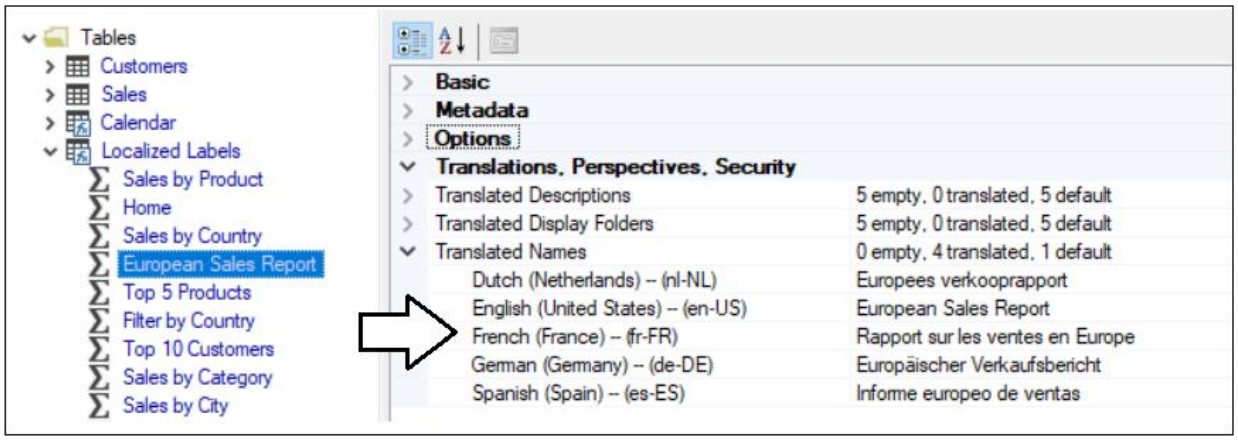
The data model for the **ProductSales.pbix** developer sample has been extended with four secondary **Culture** objects to add metadata translation support for four more languages including Spanish, French, German and Dutch.



Once you have added a **Culture** object for each language you need to support, the **Translated Names** section of the property sheet for each dataset object will provide the ability to add a new translation for each language. The following screenshot shows how the metadata translations have been added to supply translated names for the **Products** table.



The **ProductSales.pbix** developer sample also contains translations for all the measures in the **Localized Labels** table.



Save a Dataset Definition as a BIM File

The Tabular Editor provides a **Save As** command which can be used to save a data model definition in a JSON file format. By convention, a JSON file with a tabular dataset definition is created with a ***.bim** extension. You can build up your understanding of how translations work by saving a dataset definition as a **BIM** file and then inspecting the JSON inside. Let's begin by examining the JSON for a simple a dataset definition which contains a table with two columns.

```
{
  "name": "29ddb796-a33b-40d8-b61b-a2f901c0fcb7",
  "model": {
    "culture": "en-US",
    "tables": [
      { "name": "Products",
        "columns": [
          { "name": "Category", "dataType": "string" },
          { "name": "Product", "dataType": "string" }
        ] }
    ],
    "cultures": [
      { "name": "en-US" }
    ]
  }
}
```

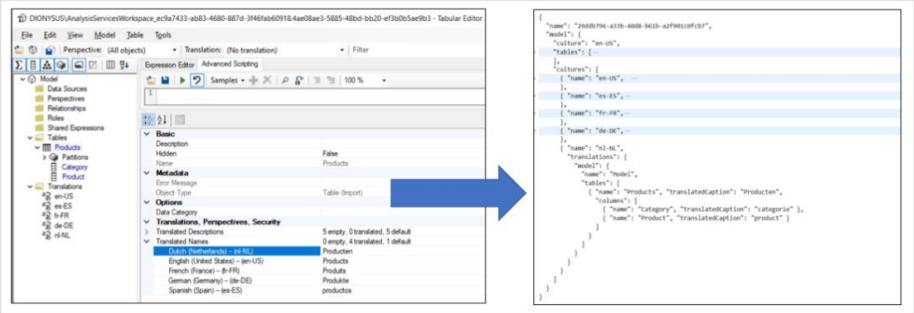
As you can see from the previous screenshot, this new PBIX project file has been created with a default Culture of **en-US**. However, the default Culture object is created without any translations inside. You must populate the default Culture object with translations yourself either by hand or by writing code to automate the process. Once you have populated the default Culture object with translations, you can see these translations are tracked on an object-by-object basis in the **tables** collection using the **translatedCaption** property.


```
{
  "name": "29ddb796-a33b-40d8-b61b-a2f901c0fcb7",
  "model": {
    "culture": "en-US",
    "tables": [
      { "name": "Products",
        "columns": [
          { "name": "Category", "dataType": "string" },
          { "name": "Product", "dataType": "string" }
        ] }
    ],
    "cultures": [
      {
        "name": "en-US",
        "translations": {
          "model": {
            "name": "Model",
            "tables": [
              { "name": "Products", "translatedCaption": "Products",
                "columns": [
                  { "name": "Category", "translatedCaption": "Category" },
                  { "name": "Product", "translatedCaption": "Product" }
                ]
              }
            ]
          }
        }
      }
    ]
  }
}
```

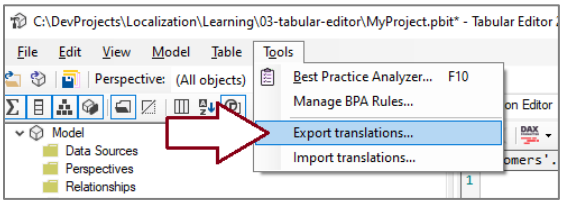

As you begin to add translations for other languages, they're tracked in a similar fashion in a separate **Culture** object.


```
{
  "name": "29ddb796-a33b-40d8-b61b-a2f901c0fcb7",
  "model": {
    "culture": "en-US",
    "tables": [···
    ],
    "cultures": [
      { "name": "en-US",  ···
      },
      { "name": "es-ES",···
      },
      { "name": "fr-FR",···
      },
      { "name": "de-DE",···
      },
      { "name": "nl-NL",
        "translations": {
          "model": {
            "name": "Model",
            "tables": [
              { "name": "Products", "translatedCaption": "Producten",
                "columns": [
                  { "name": "Category", "translatedCaption": "Categorie" },
                  { "name": "Product", "translatedCaption": "Product" }
                ]
              }
            ]
          }
        }
      }
    ]
  }
}
```

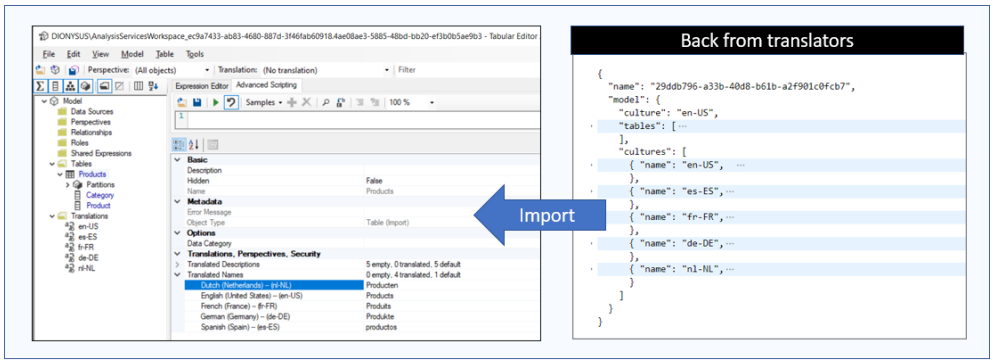
Think about what happens when you add metadata translations by hand in Tabular Editor. Behind the scenes, Tabular Editor adds metadata translations by populating **Culture** objects in the dataset definition.



Tabular Editor supports exporting and importing translations using a JSON file format. This can be a useful feature when you need to integrate human translators into the multi-language report development process. Imagine a scenario where you have just finished creating a dataset in Power BI Desktop by adding the tables, columns and measures. Next, you can open the dataset definition in Tabular Editor and populate the metadata translations for default Culture. After that, you can export the translations in a JSON file formatting using the **Export Translations…** menu command.



The exported JSON file with the metadata translations has the exact same JSON layout as the **Culture** object in a **BIM** dataset definition file. The difference is that the JSON file with exported translations omits any metadata from the dataset definition that does not involve metadata translations. The exported JSON file can then be extended with other **Culture** objects containing the translations for secondary languages. Once the JSON file has been extended with **Culture** objects for each language, the updated file with the translations can then be imported back into Tabular Editor to eliminate the need to add metadata translations by hand.



So far, you've learned the Tabular Editor provides a simple way to add and manage metadata translations by hand. While adding translations by hand with the Tabular Editor is a great way to learn about translations, it can become more tedious as the number of database objects requiring translations increases.

If the number of tables and fields in a data model is small, you can add translations by hand without any problems. But what happens when you're working with a large data model where the dataset contains 30 tables, 500 columns and 250 measures? It can take you 3-4 hours of tedious copy-and-paste operations just to populate the translations for the default Culture object in Tabular Editor. The need to add and managed metadata translations in a more scalable manner provides motivation for learning how to write custom code to automate the process using Tabular Object Model (TOM).

Program with Advanced Scripting in Tabular Editor

Behind the scenes, the Tabular Editor uses the Tabular Object Model (TOM) to create and manage dataset objects and to read and update dataset object properties. The Tabular Editor also provides an Advanced Scripting feature which makes it possible to write and run batches of C# code which program against TOM. The purpose of this section is to get you started with a few quick samples which demonstrate automating the process of creating and managing translations.

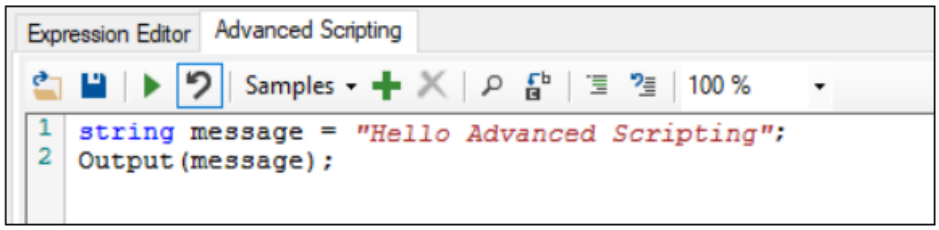
The first thing to understand about the Advanced Scripting features is that you're not really programming against TOM. Instead, Tabular Editor provides a set of TOM wrapper objects. You program against the TOM wrapper objects and these wrapper objects, in turn, make calls into TOM on your behalf. If you plan to become productive with Advanced Scripting, you should start by reading the Advanced Scripting documentation.

The Tabular Editor provides a programming model with a few essential top-level objects. First, there is a **Model** object that provides access to all the objects within the current dataset definition. The **Model** object makes it possible to enumerate through every table in order to examine or modify the columns and measures inside. There is also a **Selected** object which makes it possible to execute code against any object or objects that are selected in the treeview control of the left navigation menu. Finally, there is an **Output** object that makes it possible to display a debugging message in a model dialog.

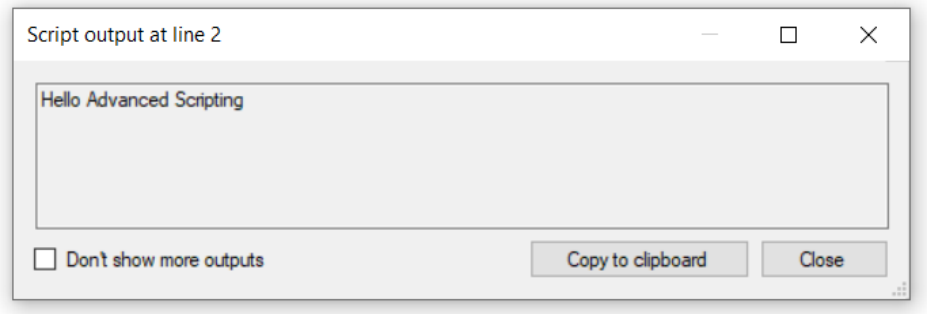
The Advanced Editor is great because it provides instant gratification. Start by typing in a simple batch of C# code.

```
String message = "Hello Advanced Scripting";
Output(message);
```

Click the green arrow button on the Advanced Editor toolbar or the press {F5} key to execute your code.



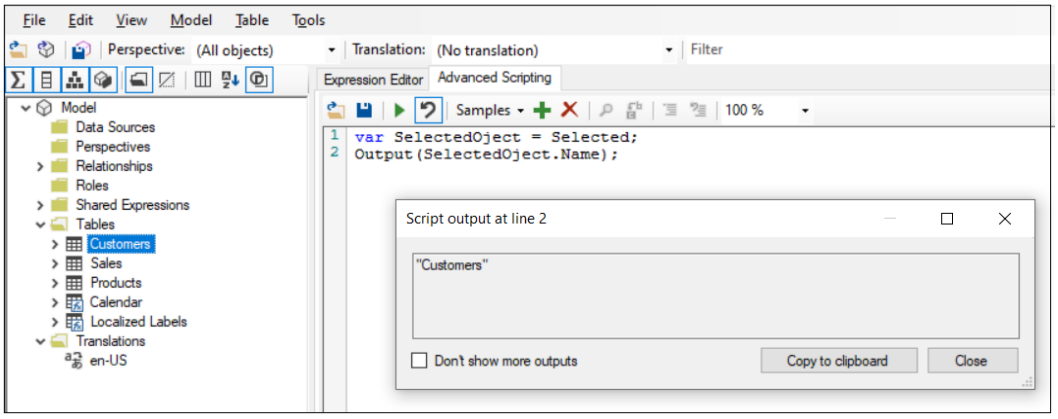
You should now see the model dialog displayed by the Output object with your message. Congratulations. You can now tell all your friends that you use Advanced Scripting in Tabular Editor!



The **Selected** object makes it possible to write generic batch of C# code that program against whatever dataset object is selected in the treeview control of the left-hand navigation menu. For example, you can experiment by writing the following C# code which displays the name of the **Selected** object.

```
var SelectedObject = Selected;
Output(SelectedObject.Name);
```

Now select a table in the treeview and execute the C# code. Your code is able to access the dataset object for whatever table, column or measure is selected in the treeview control and retrieve the value of the **Name** property.



Now let's write some C# code to populate the metadata translations for a dataset object with a translation for the default culture. The first thing you need to determine in this scenario is the default culture name for the current dataset definition. Culture names are stored as a string value which specifies a language and a locale. For example, a culture name of **en-US** specifies the culture language is English and the culture locale is the United States. Your code can dynamically determine the default culture name by examining the **Culture** property of the top-level **Model** object.

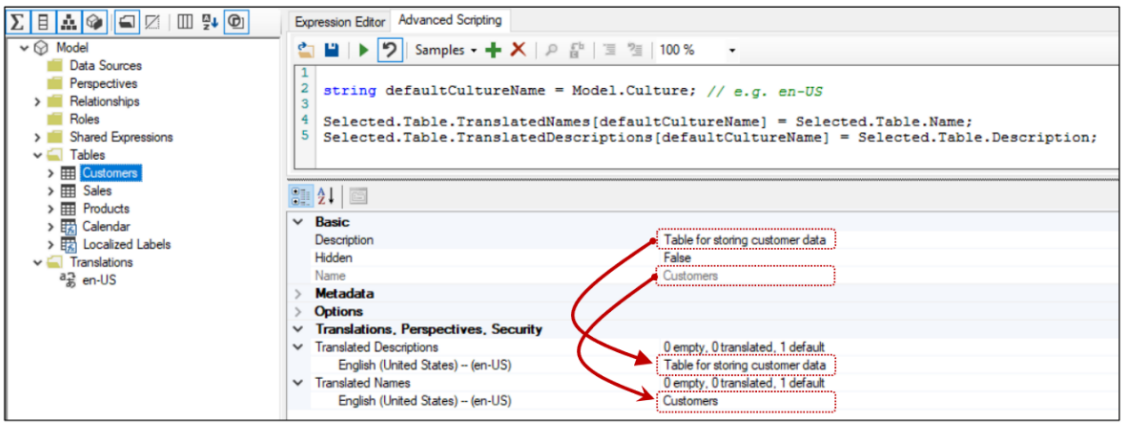
When using Advanced Scripting, the TOM wrapper classes for dataset objects provide two important convenience collections named **TranslatedNames** and **TranslatedDescriptions**. These convenience collections are implemented as dictionaries which make it easy to add or overwrite a translation for a dataset object using the culture name as a key.

```
Selected.Table.TranslatedNames["en-US"] = "Customers";
```

Now, let's say you want to write the C# code program against a table object select and the treeview and add translations for the default culture. First, you can determine the default culture using **Model.Culture**. Next, you can copy the **Name** property and **Description** property of the **Selected.Table** object and add them to the **TranslatedNames** collection and the **TranslatedDescriptions** collection.

```
Selected.Table.TranslatedNames[Model.Culture] = Selected.Table.Name;
Selected.Table.TranslatedDescriptions[Model.Culture] = Selected.Table.Description;
```

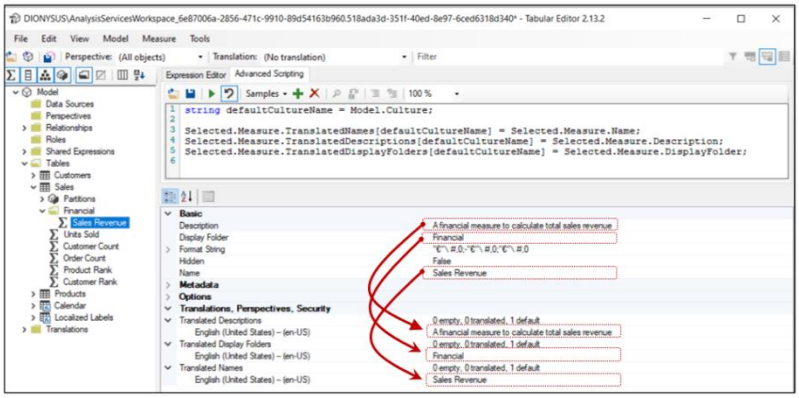
This code demonstrates a simple pattern of programming against a dataset object to populate translations for the default culture. You can use the **Name** and **Description** property values to populate translations in the convenience collections named **TranslatedNames** and **TranslatedDescriptions**.



When using Advanced Scripting, all dataset objects provide the two convenience collections named **TranslatedNames** and **TranslatedDescriptions**. When you program against columns, measures and hierarchies, you have access to a third convenience collection named **TranslatedDisplayFolders** as shown in the following code listing.

```
Selected.Measure.TranslatedNames[Model.Culture] = Selected.Measure.Name;
Selected.Measure.TranslatedDescriptions[Model.Culture] = Selected.Measure.Description;
Selected.Measure.TranslatedDisplayFolders[Model.Culture] = Selected.Measure.DisplayFolder;
```

Once again, this code follows the pattern shown earlier where translations for the default culture are created from dataset object properties which in this scenario are **Name**, **Description** and **DisplayFolder**. While you could copy and paste these property values by hand in the Tabular Editor, automating these tasks with code is a much better approach.



Now let's take this example one step further so you don't have to execute the code on each dataset object individually. You can write an outer **foreach** loop that enumerates through the **Tables** collection of the **Model** object. Inside the outer **foreach** loop which enumerates tables, you can add inner **foreach** loops to enumerate through the columns, measures and hierarchies for each table. You can avoid hidden tables as they do not require localization in most cases.

```
foreach(Table table in Model.Tables) {
    if(!table.IsHidden){
        // generate translation for table name
        table.TranslatedNames[Model.Culture] = table.Name;

        foreach (Column column in table.Columns) {
            // generate translation for column name
            column.TranslatedNames[Model.Culture] = column.Name;
        };

        foreach (Measure measure in table.Measures) {
            // generate translation for measure name
            measure.TranslatedNames[Model.Culture] = measure.Name;
        };

        foreach (Hierarchy hierarchy in table.Hierarchies) {
            // generate translation for hierarchy name
            hierarchy.TranslatedNames[Model.Culture] = hierarchy.Name;
        };
    }
}
```

If you'd like to use the Advanced Scripting features to automate tasks associated with localization, your next step is to learn how to save your C# code as custom actions. This will make it possible for you to create a library of reusable code snippets that can be shared across a team and used in the multi-language report development process.

Now, it's time to move beyond the Tabular Editor and examine writing a C# application which can be integrated with Power BI Desktop as an External Tool. This approach will provide greater flexibility then using the Advanced Scripting features in Tabular Editor mainly due to the fact that the code in the next section will program against TOM directly.

Create an External Tool for Managing Translations using TOM

THIS SECTION IS UNDER CONSTRUCTION

In the previous section we examined the Advanced Scripting features of Tabular Editor which makes it possible to automate localization tasks involved in the multi-language report development process.

You can also automate the task of adding translations using the Tabular Object Model (TOM) which is an extension of Analysis Management Object (AMO) client library. You can read through Programming Datasets with the Tabular Object Model (TOM) to learn the fundamentals for getting started with this powerful API. In this article, we'll focus on using TOM to add and manage metadata translations in a Power BI dataset. We'll begin by programming TOM using the Advanced Scripting support in the Tabular Editor. After that, we'll examine how to create a custom C# application in Visual Studio that can be configured to load as an External Tool for Power BI Desktop.

The TranslationsBuilder Developer Sample

THIS SECTION IS UNDER CONSTRUCTION

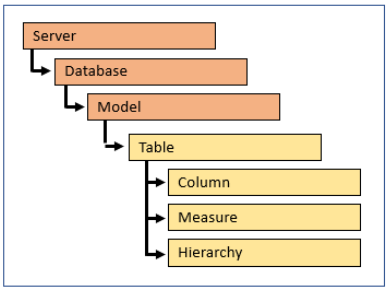
Program TOM to enumerate dataset objects

THIS SECTION IS UNDER CONSTRUCTION

Let's start with a simple TOM programming example which connects to a PBIX project file loaded into Power BI Desktop. After establishing a connection this code enumerates through the tables in a data model and prints the tabke name to the console window.

```
using System;
using Microsoft.AnalysisServices.Tabular;

class Program {

  const string connectString = "localhost:50000"; // update with port number for Power BI Desktop session

  static void Main() {

    Server server = new Server();
    server.Connect(connectString);

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

    foreach(Table table in model.Tables) {
      Console.WriteLine("Table: " + table.Name);
    }

  }
}
```

SSSSSS



Within a table, you can further enumerate through its three main collections which includes

```
foreach (Table table in model.Tables) {
  Console.WriteLine("Table: " + table.Name);

  // enumerate through columns
  foreach (Column column in table.Columns) {
    Console.WriteLine("Column: " + column.Name);
  };

  // enumerate through measures
  foreach (Measure measure in table.Measures) {
    Console.WriteLine("Measure: " + measure.Name);
  };

  // enumerate through hierarchies
  foreach (Hierarchy hierarchy in table.Hierarchies) {
    Console.WriteLine("Hierarchy: " + hierarchy.Name);
  };
}
```

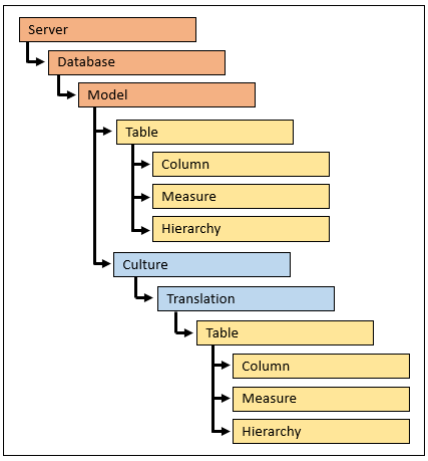
For

Program TOM Support for Adding Translation

THIS SECTION IS UNDER CONSTRUCTION

For each object, you can add translations using three different properties that are supported at the dataset object level. These properties are **Caption**, **Description** and **DisplayFolder**. The **Caption** property is used to add translations for the names of dataset objects. You can think of the **Caption** property of a dataset object as the display name which is seen by users.

You use the **Caption** property to add a translation for the name of an object such as a table, column, measure or hierarchy. If your data model uses display folders to organize columns and measures within tables, you need to add additional translation using the



When you create a new PBIX project, it has a default culture (en-US) but contains no translations. You must add translations for each spoken language.

Enable Human Workflows for Translation using Export and Import

THIS SECTION IS UNDER CONSTRUCTION

Another important consideration when building multi-language reports involves the human aspect of translating text values from one language to another. While it's possible to initially generate the first round of translations using the Microsoft Translation service, you will eventually need to integrate carbon-based life forms (i.e. people) into the development process who can act as translators to generate quality translations. Furthermore, you cannot expect that people who work as language translators will be able to use an advanced data modeling tool like the Tabular Editor.

While it is technically possible to have human translators work on files generated by the Export Translations command of the Tabular Editor, the JSON-based format will likely be rejected by professional translator teams due to it being a non-standard file format that is hard to work with. Once you begin writing custom code with TOM, however, you can generate the files that are sent out to translators using whatever file format required. If you are working with a professional translation team, you might be requires to generate translation files in a standard translation format such as RESX files or XLIFF files. You can also generate translations files in easy-to-use formats such as CSV files or XLSX files.

Load Power BI Reports with Specific Languages and Locales

THIS SECTION IS UNDER CONSTRUCTION

Design and implement a content translation strategy

THIS SECTION IS UNDER CONSTRUCTION

Modify the Data Model Design to Support Content Translation

THIS SECTION IS UNDER CONSTRUCTION

Using Power Query to Generate Content Translation Rows

THIS SECTION IS UNDER CONSTRUCTION

Setting the Language for Current User using RLS and UserCulture

THIS SECTION IS UNDER CONSTRUCTION