Building Multi-language Reports in Power BI

Published: January 2023

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, German, Japanese or Hindi for other users. If a company or organization has the requirement of building Power BI reports that support multiple languages, it's not 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 a strategy for building multi-language reports.

This article has been created to provide guidance and to teach the skills required to build Power BI reports that support multiple languages. You need to learn a few key concepts about how Power BI translations work and how to automate repetitive tasks that would take forever to complete manually. An essential part of this guidance is based on using an external tool named **Translations Builder** that's been designed for content creators using Power BI Desktop. Once you understand how all the pieces fit together, you'll be able to build multi-language reports for Power BI using a strategy that is reliable, predictable and scalable.

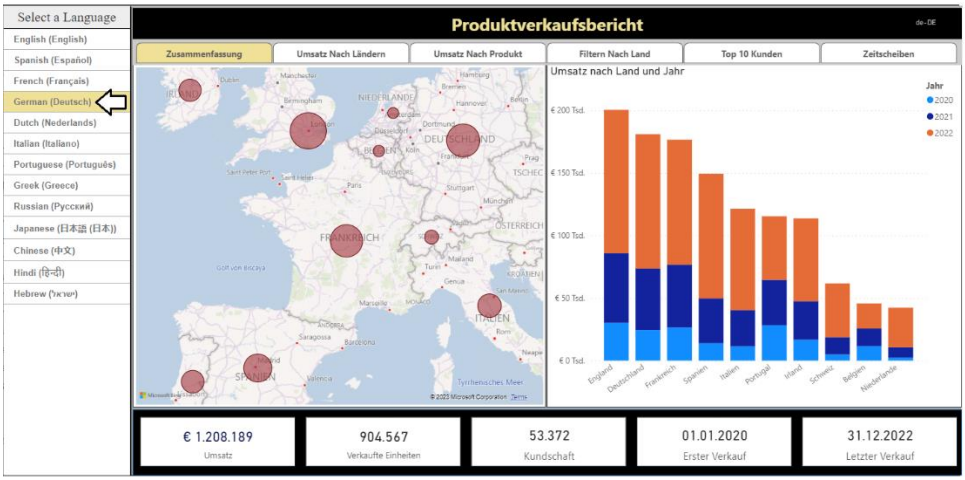
Table of Contents

Multi-language Report Live Demo

This article is accompanied a **live demo** based on a single PBIX file solution named **ProductSalesMultiLanguage.pbix**. This live demo shows the potential of building multi-language reports for Power BI. The report in the live demo can be loaded using English, Spanish, French, German, Dutch, Italian, Portuguese, Greek, Russian, Japanese, Chinese, Hindi and Hebrew. You can test out the live demo and the experience a Power BI report that support over a dozen secondary languages by navigating the following URL.

- **https://multilanguagereportdemo.azurewebsites.net**

When you test out the live demo, experiment by clicking links in the left navigation to reload the report using different langauges. For example, click on the link with the caption of **German (Deutsch)**. When you do, you will see the report load with German translations as shown in the following screenshot.



The live demo is based on a custom web application that uses Power BI embedding. When you click on a link in the left navigation, there is JavaScript behind this web page that responds by explicitly reloading the report using the language of German intead of English. You can see that all the text-based elements for the entire report are now displayed with their German translations instead of with the default English translations.

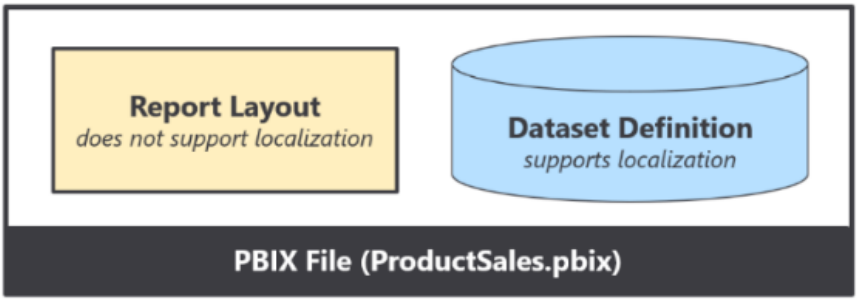
Power BI Support for Metadata Translations

The primary localization 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 a dataset object that's been translated for a specific language. Consider a simple example. If your dataset 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 types of dataset objects that support metadata translations include **Table**, **Column**, **Measure**, **Hierarchy** and **Hierarchy Level**. 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**.

Power BI reports and datasets that support multiple languages can only run in workspaces which are associated a dedicated capacity created using Power BI Premium or the Power BI Embedded Service. That means multi-language reports will not load correctly when launched from a workspace in the shared capacity. If you are working in a Power BI workspace that does not display a diamond indicating it's a Premium workspace, you will find that multi-language reports don't work as expected because there is no support for loading translations from secondary languages.

Another critical point to understand is that the Power BI support for metadata translations only applies to datasets. Neither Power BI Desktop nor the Power BI Service provide any support for storing or loading translations for text values stored as part of the report layout.



Think about a scenario where you add a textbox or button to a Power BI report and then you type in a hard-coded text value for a string 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 contain hard-coded text values stored in the report layout. 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 hidden and never displayed to the user.

Implementing Translations Dynamically using Measures and USERCULTURE

A second essential feature to assist with building multi-language reports in Power BI is the DAX **USERCULTURE** function. The **USERCULTURE** function returns a string which includes a lower-case language ID parsed together with an upper-case locale identifier. Here are a few examples of string values retuned by the **USERCULTURE** function which indicate a specific language and locale.

USERCULTURE	Language	Locale
en-US	English	United States
es-ES	Spanish	Spain
fr-FR	French	France
de-DE	German	Germany
Ja-JP	Japanese	Japan

Remember that you can only use the **USERCULTURE** function to implement dynamic translations in measures. When you use **USERCULTURE** in the DAX expression for a measure, it's guaranteed to return the language and locale identifier for the current user. The same is not true if you use the **USERCULTURE** function in the DAX expression for a calculated table or a calculated column which get evaluated at dataset load time. When you use **USERCULTURE** in the DAX expression for a table or calculated column, you don't get the same guaranteed that it uses the language and locale of the current user.

The live demo displays the return value of **USERCULTURE** in the upper right corner of the report banner. You will not typically display a report element like this in a real application, but it's included with the live demo so you can see exactly what language and locale identifier are being used to load the report each time you switch to a new language.

Let's look at a simple example of writing a DAX expression for a measure that implements dynamic translations. You can start by extracting the language ID for the current user using **USERCULTURE** together with **LEFT**.

```
CurrentLanguage = LEFT(USERCULTURE(), 2)
```

Now, you can take things a step further by adding a **SWITCH** statement to form a basic pattern for dynamic translations.

```
Product Sales Report Label = SWITCH(LEFT(USERCULTURE(), 2),
   "es", "Informe De Ventas De Productos",
   "fr", "Rapport Sur Les Ventes De Produits",
   "de", "Produktverkaufsbericht",
   "Product Sales Report"
)
```

OK, maybe it's not as impressive as some of those fancy DAX patterns coming out of Italy. But hey, it's a start.

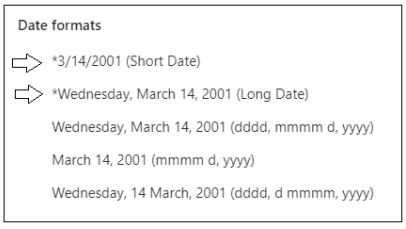
Formatting Dates and Numbers with the Current User's Locale

Every report that loads in the Power BI Service is initialized with a specific language and a specific locale. The default behavior of the Power BI Service is to load each report using the language and regional locale specified by the user's browser settings. However, those settings can be overridden by adding the **language** query string parameter to the end of the report URL. If you're developing with Power BI embedding, you also have complete control to load a report with a specific language and locale as demonstrated by the live demo.

You've already seen that you can implement dynamic translations by writing a DAX expression in a measure with conditional logic based on the user's language. This is a technique that will be used frequently when building reports that support multiple languages. However, you will not be required to write conditional DAX logic based on the user's locale. Why is that?

The short answer is that Power BI visuals automatically handle locale-specific formatting behind the scenes. This makes things so much easier. The long answer is that a Power BI visual inspects the locale of the current user before rendering. During the rendering process, the visual determines what formatting to use for a date or numeric value based on the user's locale and the format string of the source column or measure.

Consider a simple scenario in which you're building a report for an audience of report consumers that live in both New York [**en-US**] and in London [**en-GB**]. All users speak English (**en**), but yet some live in different regions (**US** vs **GB**) where dates and numbers are formatted differently. For example, a user from New York wants to see dates in a **mm/dd/yyyy** format while a user from London wants to see dates in a **dd/mm/yyyy** format. Everything thing works out as long as you configure columns and measures using format strings that support regional formatting. If you are formatting a date, it is recommended you use a format string of **Short Date** or **Long Date** because those format strings support regional formatting.



Here are a few examples of how a date value formatted with **Short Date** appears when loaded under different locales.

en-US	12/31/2022
en-GB	31/12/2022
pt-PT	31-12-2022
de-DE	31.12.2022
ja-JP	2022/12/31

The Japanese formatting is hands-down the winner. It's the only format that automatically sorts chronologically.

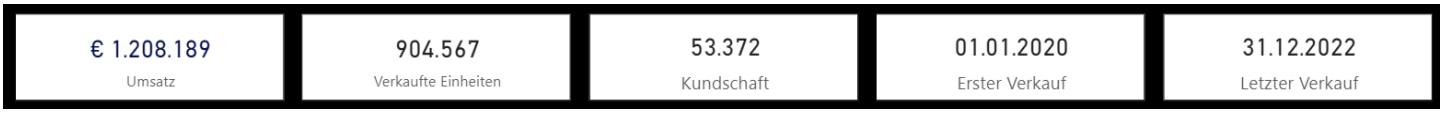
Understanding the Three Types of Translations

When it comes to localizing Power BI artifacts such as datasets and reports, there are three different types of translations and you must be able distinguish between them. These are the three types of translations you should understand.

- **Metadata Translations**
- **Report Label Translations**
- **Data Translations**

Now, let's examine all three types in a little more depth.

Metadata translations provides localized values for dataset object properties. The object types which support metadata translation include tables, columns, measures, hierarchies and hierarchy levels. The following screenshot shows how metadata translations provide German names for the measures displayed in Card visuals.



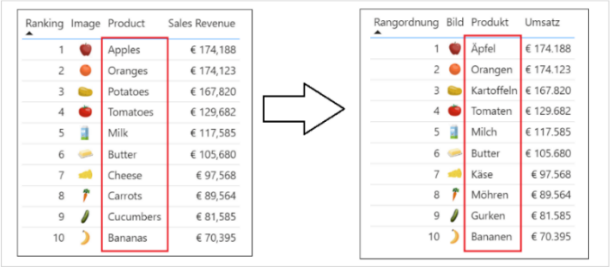
Metadata translations are also used to display column names and measure names in tables and matrices.

Metadata translations are the easiest to create, manage and integrate into a Power BI report. By leveraging the features of Translations Builder to generate machine translations, you can add all the metadata translations you need to build and test a Power BI report in a matter of seconds. As you will discover, adding metadata translations to your dataset is fairly straight-ahead and an essential first step. However, metadata translations rarely provide a complete solution by themselves. A complete solution will typically require going further to localize report labels.

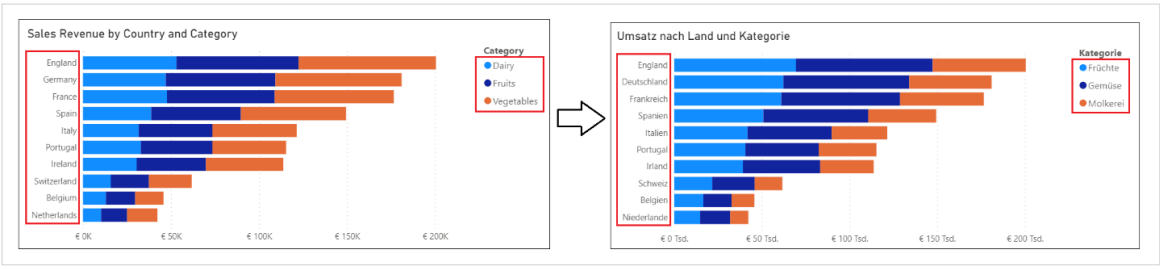
Report label translations provide localized values for text elements on a report that are not directly associated with a dataset object. Examples of report labels include the report title, section headings and button captions. Here are a few examples of report label translations in the live demo with the report title and the captions of navigation buttons.

Report label translations are harder to create and manage than metadata translations because Power BI provides no built-in feature to track or integrate them. Translations Builder solves this problem using the **Localized Labels** table strategy. This strategy is based on creating a hidden table named **Localized Labels** in the dataset behind a report and adding measures whose sole purpose is to track the required translations for each report label. You will learn more about the **Localized Labels** table strategy later in this article in the section titled **Understanding the Localized Labels Table**.

Data translations provide translated values for text-based columns in the underlying data itself. Think about a scenario where a Power BI report displays product names imported from the rows of the **Products** table in an underlying database. Data translations are used to display product names differently for users who speak different languages. For example, some users see products names in English while other users see product names in secondary languages.



Data translations also appear in the axes of cartesian visuals and in legends as shown in the following screenshot.



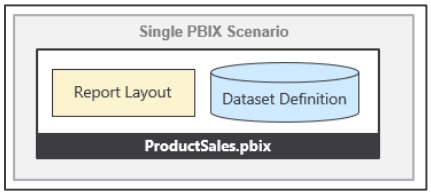
Data translations are harder to design and implement than the other two types of translations. The reason it's harder is that you must typically redesign the underlying datasource with additional text columns for secondary language translations. Once the underlying datasource has been extended with extra text columns for secondary language translations, you can then use a powerful new feature in Power BI Desktop known as ***Field Parameters*** to design a scheme where you can control the loading the data translations for a specific language through filtering.

> While every multi-language report will typically require both metadata translations and report label translations, it is not as clear whether they will also require data translations. Some projects to build a multilanguage report for Power BI will require data translations while others will not. This point will be revisited in more depth later in this article.

Packaging Dataset and Report in PBIX Project Files

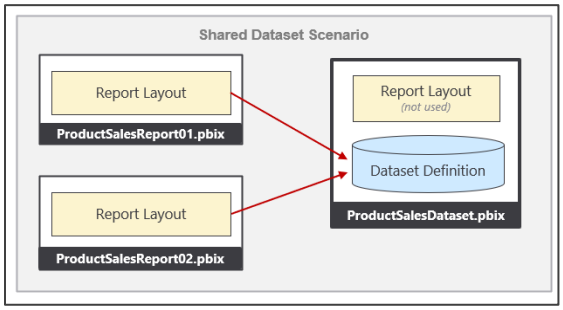
Now that you understand high-level concepts of building multi-language reports with translations, it's time to discuss the multi-language report development process. The first step here is to decide how to package your dataset definitions and report layouts for distribution. Let's examine two popular approaches used by content creators who work with Power BI Desktop.

In the first approach, the goal is to keep things simple and convenient by creating a single PBIX project file which contains both a report layout and its underlying dataset definition. You can easily deploy a reporting solution like this by importing the PBIX project file into a Power BI workspace. If you need to update either the report layout or the dataset definition 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 datasets while other teams are responsible for building reports. For a scenario like this, it makes sense to split out datasets and report layouts into separate PBIX project files.

To use the shared dataset approach, you create one PBIX project file with a dataset 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.

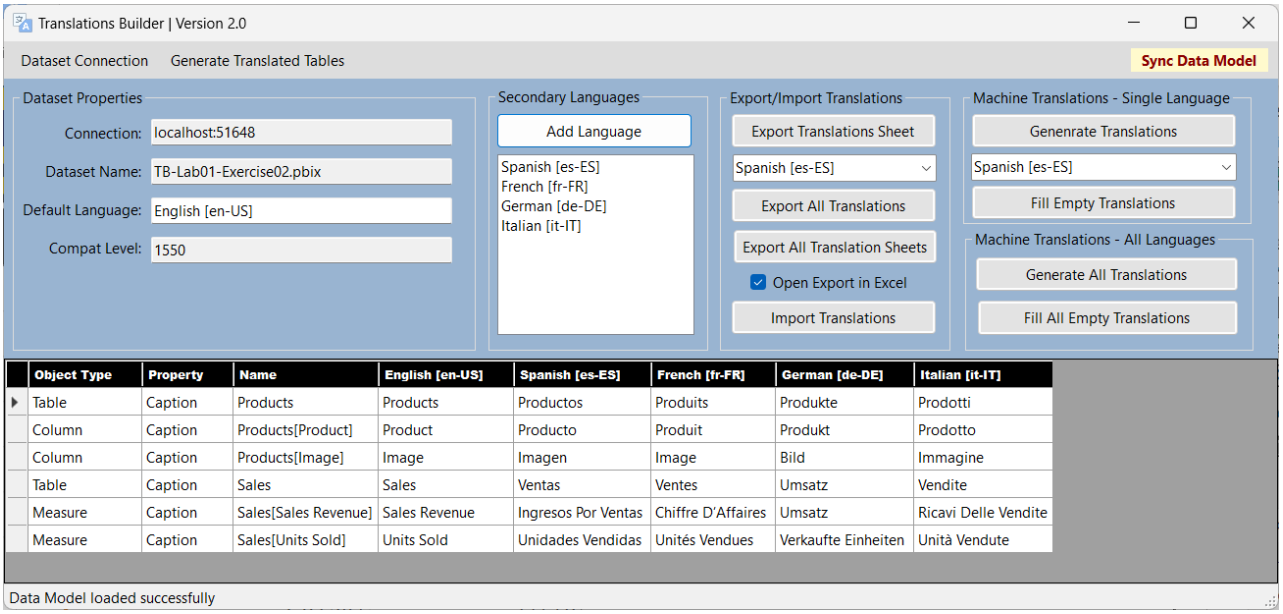


From the perspective of adding multi-language support to a Power BI solution, it really doesn't matter which of these approaches you choose. The techniques and disciplines used to build multi-language reports remain the same whether you decide to build your solution using a single PBIX project file or with a combination of PBIX project files. 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.

> While the solution provided by **ProductSalesMultiLanguage.pbix** demonstrates a single PBIX project file approach where the dataset and report are packaged together for convenience. However, nothing changes if you package and distribute datasets and reports using separate PBIX files. You will use the exact same concepts and techniques to build multi-language reports in scenarios where your solution contains multiple PBIX files.

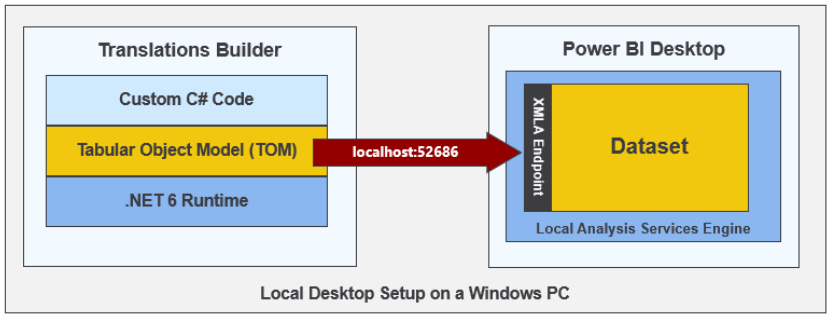
Understanding How Translations Builder Works

Translations Builder is a tool designed for content creators using Power BI Desktop. Content creators can use this tools to add multi-language support to PBIX project files. The following screenshot shows what Translations Builder looks like when working with a simple PBIX project that supports a small number of secondary languages.

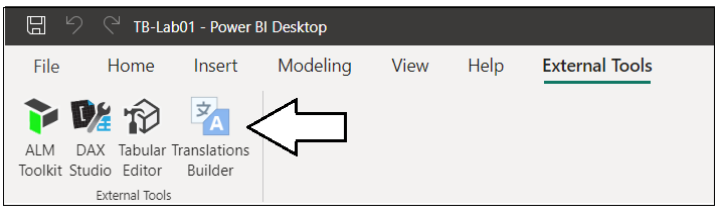


Translations Builder is an external tool developed for Power BI Desktop using C#, .NET 6, and Windows Forms. Translations Builder uses an API known as the ***Tabular Object Model (TOM)*** to update datasets that have been loaded into memory and are running in a session of Power BI Desktop. Translations Builder does most of its work by adding and updating the metadata translations associated with datasets objects including tables, columns and measures. However, there are several scenarios in which Translations Builder will actually create new tables in a dataset to implement strategies to handle various aspects of building multi-language reports.

When you open a PBIX project in Power BI Desktop, the dataset defined inside the PBIX file is loaded into memory in a local session of the Analysis Services engine. Translations Builder uses TOM to establish a direct connection to the dataset of the current PBIX project.



The Translations Builder project has been developed using the [external tools integration support] for Power BI Desktop. You can install Translations Builder on a Windows PC where you've already installed Power BI Desktop using instructions in the [Translations Builder Installation Guide]. Once the Translations Builder application has been installed on a Windows computer, you can launch it directly from Power BI Desktop using the **External Tools** tab in the ribbon.



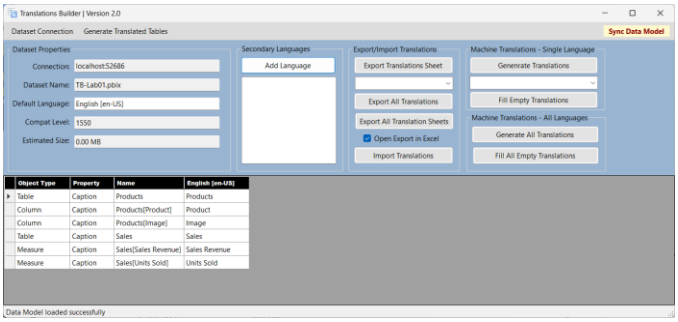
When you launch an external tool like Translations Builder, the application is passed startup parameters including a connection string which can be used to establish a connection back to a dataset that's loaded in Power BI Desktop. This allows Translations Builder to display dataset information and to provide commands to automate adding metadata translations. You can read [Translations Builder Developers Guide] if you want to learn more about the details of working with Translations Builder as a developer. The content in this article will focus on teaching concepts and localization skills to content creators building datasets and reports with Power BI Desktop.

The key value proposition of Translations Builder is that is allows a content creator to view, add and update metadata translations using a two-dimensional grid. This ***translations grid*** simplifies the user experience because it abstracts aways the low-level details or reading and writing metadata translation associated with a dataset definition. Users work with the translation grid to view, add and edit metadata translations in a manner that is similar to working with data inside an Excel spreadsheet.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	French [fr-FR]	German [de-DE]	Italian [it-IT]
Table	Caption	Products	Products	Productos	Produits	Produkte	Prodotti
Column	Caption	Products[Product]	Product	Producto	Produit	Produkt	Prodotto
Column	Caption	Products[Image]	Image	Imagen	Image	Bild	Immagine
Table	Caption	Sales	Sales	Ventas	Ventes	Umsatz	Vendite
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Chiffre D'Affaires	Umsatz	Ricavi Delle Vendite
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Unités Vendues	Verkaufte Einheiten	Unità Vendute

Adding Secondary Languages and Translations

When you launch Translations Builder with a PBIX project for the first time, the translation grid will display a row in for each non-hidden table, measure and column in the project's underlying data model. The translation grid does not display rows for dataset objects in the data model that are hidden from report view. The reason for this is that hidden objects will not be displayed on a report and, therefore, do not require translations. The following screenshot shows the starting point for a simple data model before it's been modified to support secondary languages.

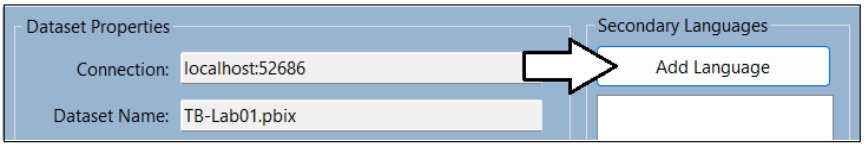


If you examine the translation grid for this PBIX project more closely, you can see the first three columns contain read-only columns used to identity each metadata translation. Each metadata translation has an **Object Type**, a **Property** and a **Name**. Translations for the **Caption** property will always be used while translations for the **Description** property and the **DisplayFolder** property can be added if required. The fourth column in the translation grid always displays the translations for the dataset's default language and locale which in this case is **English [en-US]**.

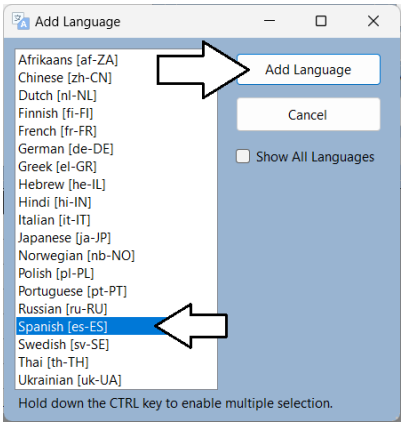
Object Type	Property	Name	English [en-US]
Table	Caption	Products	Products
Column	Caption	Products[Product]	Product
Column	Caption	Products[Image]	Image
Table	Caption	Sales	Sales
Measure	Caption	Sales[Sales Revenue]	Sales Revenue
Measure	Caption	Sales[Units Sold]	Units Sold

> While Translations Builder makes it possible to update the translations for the default language, you should do it sparingly. Doing so can be confusing because translations for the default language will not load in Power BI Desktop.

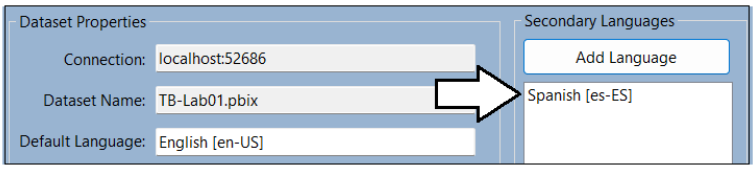
Translations Builder provides an **Add Language** command to add secondary languages to the project's data model.



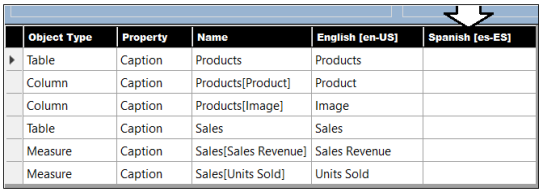
Clicking **Add Language** displays the **Add Language** dialog which allows the user to add one or more secondary languages.



After a new language has been added, the user can see the language in the **Secondary Languages** list.

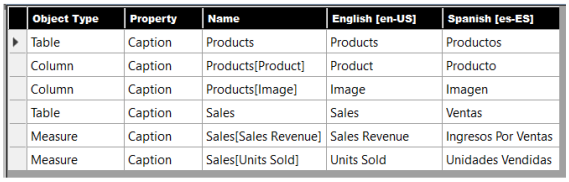


Adding a new language will also add a new column of editable cells to the translations grid.



Object Type	Property	Name	English [en-US]	Spanish [es-ES]
Table	Caption	Products	Products	
Column	Caption	Products[Product]	Product	
Column	Caption	Products[Image]	Image	
Table	Caption	Sales	Sales	
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	
Measure	Caption	Sales[Units Sold]	Units Sold	

In scenarios where content creators know how to speak all the languages involved, they can add and update translations for secondary languages directly in the translation grid with an Excel-like editing experience.

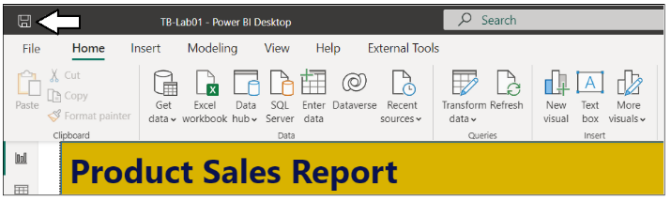


Object Type	Property	Name	English [en-US]	Spanish [es-ES]
Table	Caption	Products	Products	Productos
Column	Caption	Products[Product]	Product	Producto
Column	Caption	Products[Image]	Image	Imagen
Table	Caption	Sales	Sales	Ventas
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas

Technically speaking, Translations Builder isn't just adding a language object to the dataset. Instead, Translations Builder is actually adding a **Culture** object with includes both a language ID and a locale identifier. In Power BI datasets, a **Culture** object is identified using a string-based key which combines a lower-case language ID and an upper-case locale identifier for the geographical region. Note this is the same string-based format returned by the **USERCULTURE** function.

Translations Builder abstracts away the differences between a language and a culture. This has been done to simplify the user experience for content creators who can just think in terms of languages and not worry about the differences between a language and a culture. It's not overly important to distinguish between a language and a culture until you begin programming with TOM and you need to add new **Culture** objects to a Power BI dataset.

Another important aspect of working with Translations Builder has to do with saving your work. While external tools for Power BI Desktop like Translations Builder are able to modify the dataset loaded into memory from a PBIX file, there is no way for an external tool to trigger a command to save the in-memory changes back to the underlying PBIX file. Therefore, you must always return back to Power BI Desktop and click the **Save** command any time you have added languages and any time you have created or updated translations.



Once the changes have been written back to the PBIX file, that file can then be published to the Power BI Service for testing. Once you have tested your work and verified that the translations are working properly, you can also store the PBIX file in a source control system such as GitHub or an Azure DevOps repository. This provides the foundation for an ALM strategy where support for secondary languages and translations can be evolved across versions of a PBIX file.

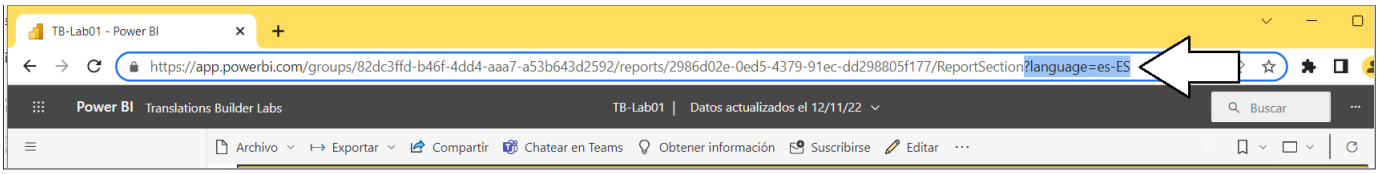
Testing Translations in the Power BI Service

One of the issues that makes working with translations a bit more complicated is that you cannot test your work in Power BI Desktop. Instead, you must test your work in the Power BI Service in a workspace associated with a Premium capacity. After you have added translation support with Translations Builder and you have saved your changes to the underlying PBIX file, you can then publish the PBIX project from Power BI Desktop to the Power BI Service for testing.

After publishing your PBIX project to the Power BI Service, you can test loading the report using secondary language by modifying the report URL with a query string parameter named **language**. After the report loads with its default language, you can click the browser address bar and add the following **language** parameter to the end of the report URL.

```
/?language=es-ES
```

Once you add the **language** parameter to the end of the URL and press **ENTER**, you should be able to verify that the **language** parameter has been accepted by the browser as it reloads the report. If you forget to add the **?** or if you do not format the **language** parameter correctly, the browser will reject the parameter and remove it from the URL as it loads the report. Once you correctly load a report using a **language** parameter value of **es-ES**, you should see the user experience for the entire Power BI Service UI switch from English to Spanish.



You will also see that the report displays the Spanish translations for the names of columns and measures.

Now that you've seen how to test your work when working with translations, it possible to make a high-level observation about working with Translations Builder. As you begin to work with secondary languages and translations to localize a PBIX project, you will follow the same set of steps again and again:

1. Make changes in Power BI Desktop
2. Publish the PBIX project to the Power BI Service
3. Test your work with a browser in the Power BI Service using **language** parameter
4. Repeat steps 1-3 until all the translations work has been completed

Are you starting to get excited about working with Translations Builder? If you want to jump right in and get started, you can try out **Hands-on Lab: Building Multi-language Reports for Power BI**.

Embedding Power BI Reports Using a Specific Language and Locale

If you are developing with Power BI embedding, you can use the Power BI JavaScript API to load reports with a specific language and locale. This is accomplished by extending the **config** object passed to **powerbi.embed** with a **localeSettings** object containing a **language** property as shown in the following code.

```
let config = {
  type: "report",
  id: reportId,
  embedUrl: embedUrl,
  accessToken: embedToken,
  tokenType: models.TokenType.Embed,
  localeSettings: { language: "de-DE" }
};

let report = powerbi.embed(reportContainer, config);
```

Generating Machine Translations using Azure Translator Service

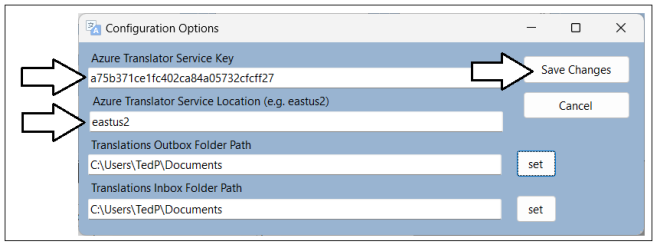
One of the biggest challenges in building multi-language reports is managing the language translation process. You must ensure that the quality of translations is high and that the translated names of tables, columns, measures and labels do not lose their meaning when translated to another language. In most cases, acquiring quality translations will require human translators to create or at least review translations as part of the multi-language report development process.

While human translators are typically an essential part of the end-to-end process, it can take a long time to send out translation files to a translation team and then to wait for them to come back. With all the recent industry advances in Artificial Intelligence (AI), you also have the option to generate machine translations using a Web API that can be called directly from an external tool such as Translations Builder. If you initially generate machine translations for each secondary language you need to support, that will give you something to work with while waiting for a translation team to return their high-quality human translations.

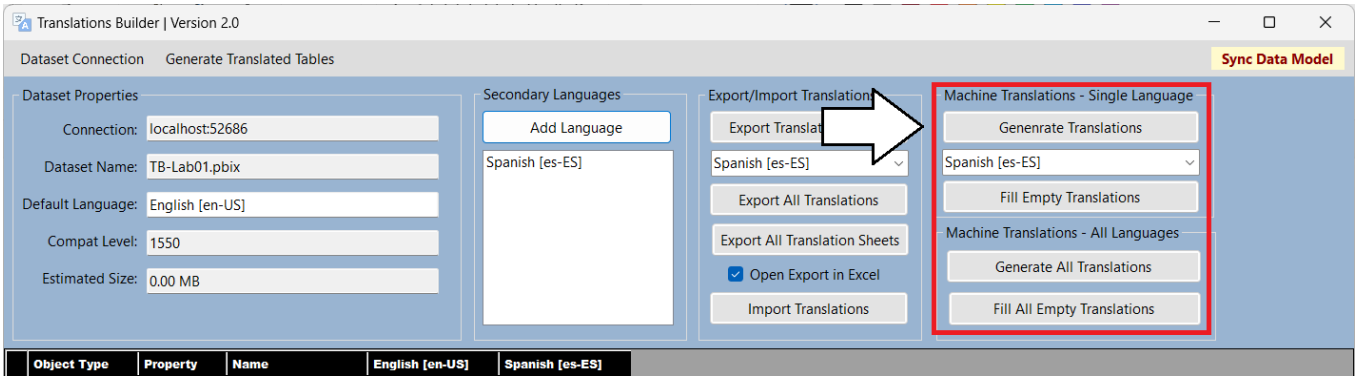
While machine translation are not always guaranteed to be high quality, they do provide value in the multi-language report development process. First, they can act as translation placeholders so you can begin your testing by loading reports using secondary languages to see if there are layout issues or unexpected line breaks. Machine translations can also provide human translators with a better starting point as they just need to review and correct translations instead of creating every translation from scratch. Finally, machine translations can be used to quickly add support for languages in scenarios where there are legal compliance issues and organizations are facing fines or litigation for non-compliance.

Translations Builder generates machine translations by executing API calls against the **Azure Translator service** which is an API endpoint offered through Azure Cognitive Services. This Web API makes it possible to automate enumerating through dataset objects to translate dataset object names from the default language to translations for secondary languages.

If you'd like to test out the support in Translations Builder for generating machine translations, you will require a Key for an instance of the Azure Translator Service. If you have an Azure subscription, you can learn how to obtain this key and its location by reading Obtaining a Key for the Azure Translator Service. Translations Builder provides a **Configuration Options** dialog which makes it possible to configure the key and location to access the Azure Translator Service.



Once a user has configured an Azure Translator Service key, Translations Builder will begin to display additional command buttons which make it possible to generate translations for a single language at a time or for all languages at once. There are also commands to generate machine translations only for the translations that are currently empty.



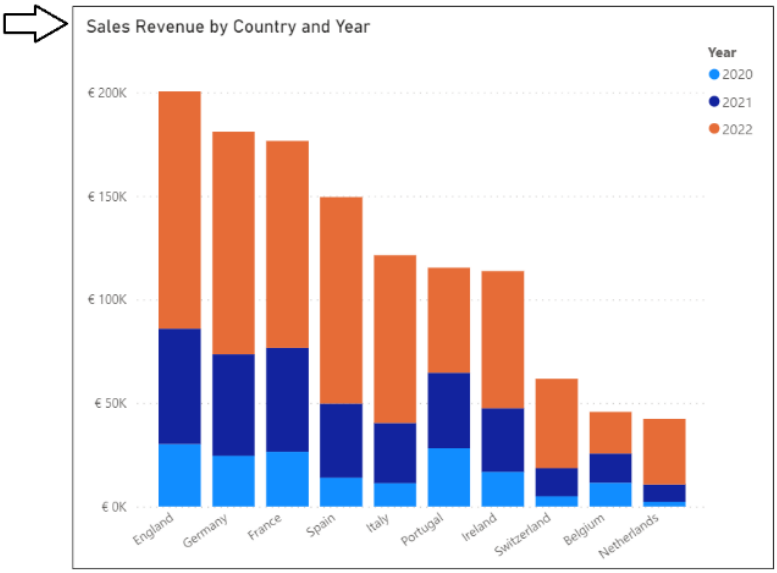
Understanding the Localized Labels Table

Earlier you learned that report label translations provide localized values for text elements on a report that are not directly associated with a dataset object. Examples of report labels are the text values for report titles, section headings and button captions. Given that Power BI provides no built-in features to track or integrate report labels, Translations Builder solves this problem using the **Localized Labels** table strategy. Before introducing this strategy, let's take a moment to discuss the problems this strategy has been designed to solve.

If you already have experience building datasets and reports with Power BI Desktop, it's critical that you learn which report design techniques to avoid when 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 hard-coded text values
- Adding a hard-coded text value for the title of a visual
- Displaying page tabs to the user

The key point here is that any hard-coded text value 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 value to its **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 supply metadata translations for the names of columns and measures in the underlying dataset definition (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 for each of the these five languages.

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 dislike the dynamically-generated visual **Title**, you must resist the temptation to replace it with a hard-coded text value. Any hard-coded 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 you should use the **Localized Labels** table strategy to create report labels that support localization.

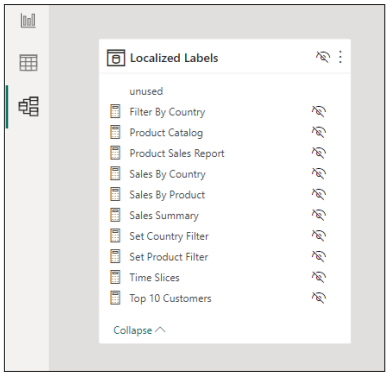
Introducing the Localized Labels Table Strategy

As discussed earlier in this article, the Power BI localization features are supported at the dataset level but not at the report layout level. At first you might ask the question "*how can I localize text-based values in a Power BI report that are not stored inside the dataset?"* The answer to this question is that there is no simple way to accomplish this. A better question to ask is "*how can I add text-based values for report labels into the dataset as dataset objects to enable localization support?"*

The idea behind the **Localized Labels** table isn't all that complicated. It builds on the idea that Power BI supports metadata translations for specific types of dataset objects including measures. When you add a report label with Translations Builder, the tool automatically adds a new measure to the **Localized Labels** table behind the scenes. Once a measure has been created for each report label, Power BI can store and manage its translations in the exact same fashion that it does for metadata translations. In fact, the **Localized Labels** table strategy uses metadata translations to implement report label translations.

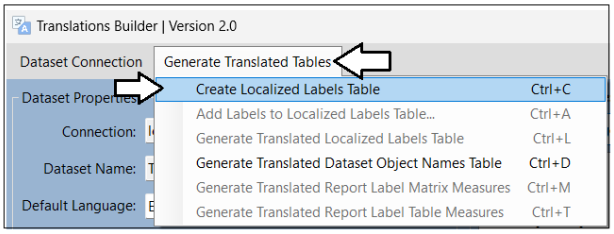
Translations Builder provides commands to create the **Localized Labels** table and to add a measure each time you need a report label. The **Localized Labels** table is created as a hidden table behind the scenes. The idea is that you can do all the work to create and manage report labels inside the Translation Builder user experience. There is no need to inspect or modify the **Localized Labels** table using the Power BI Desktop dataset design experience.

Here's an example of the **Localized Labels** table from the live demo project. As you can see it provides localized report labels for the report title, visual titles and captions for navigation buttons used throughout the report.

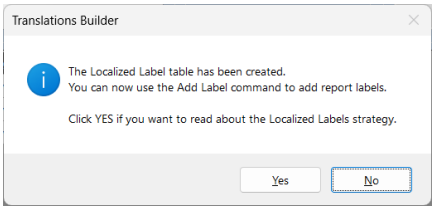


Translations Builder 1.0 introduced the **Localized Labels** table, but it did not take the strategy far enough. Consequently, the user experience was complicated and limited to surface report labels from the **Localized Labels** table directly on a report page. Translations Builder 2.0 introduces an evolved strategy to perform more work behind the scenes in order to make it easier and more natural for report designers to surface localized report labels on a report page.

You can create the **Localized Labels** table to a PBIX project by executing the **Create Localized Labels Table** command from the **Generate Translated Tables** menu.



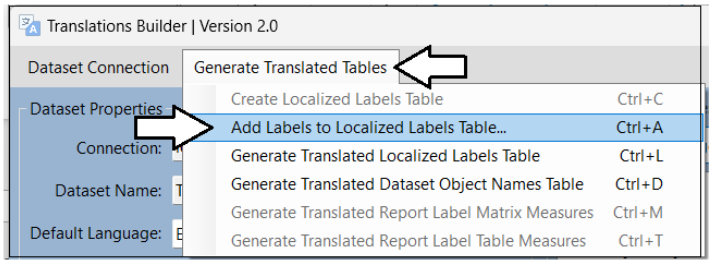
When you execute this command to create the **Localized Labels** table, you will be prompted by the following dialog asking if you want more information about the **Localized Labels** table strategy. If you click **Yes,** interestingly enough, you'll be redirected back to this very section of this very article.



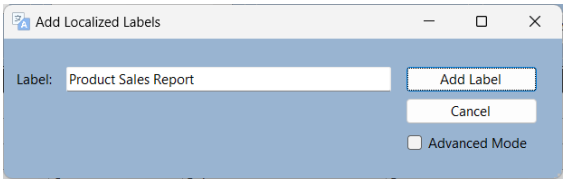
After the **Localized Labels** table has been created, you will see three sample report labels as shown in the following screenshot. In most cases you will want to delete these sample report labels and replace them with the actual report labels required on the current project.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	Dutch [nl-NL]	French [fr-FR]	German [de-DE]	Italian [it-IT]
Table	Caption	Products	Products	Productos	Producten	Produits	Produkte	Prodotti
Column	Caption	Products[Product]	Product	Producto	Product	Produit	Produkt	Prodotto
Column	Caption	Products[Image]	Image	Imagen	Beeld	Image	Bild	Immagine
Table	Caption	Sales	Sales	Ventas	Verkoop	Ventes	Umsatz	Vendite
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Omzet	Chiffre D'Affaires	Umsatz	Ricavi Delle Vendite
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Verkochte Eenheden	Unités Vendues	Verkaufte Einheiten	Unità Vendute
Measure	Caption	Localized Labels[My Report Title]	My Report Title					
Measure	Caption	Localized Labels[My Button Caption]	My Button Caption					
Measure	Caption	Localized Labels[My Visual Title]	My Visual Title					

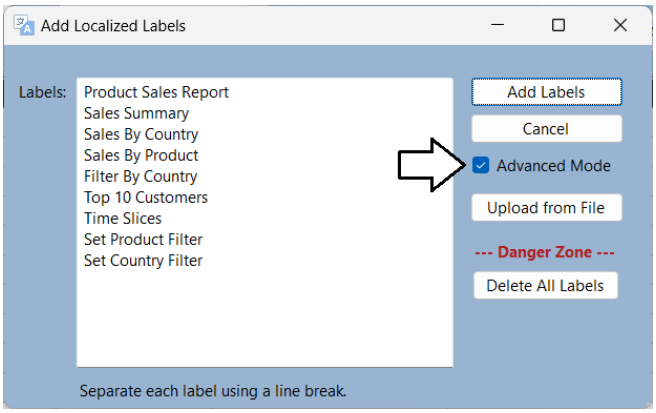
Remember, there is no need to interact with the **Localized Labels** table in Power BI Desktop. You can add and manage all the report labels you need using **Translations Builder**. To create your first report label, you can drop down the **Generate Translated Tables** menu and select **Add Labels to the Localized Labels Table**. Note you can also execute the **Add Labels to the Localized Labels Table** command using the shortcut key of **Ctrl+A**.



You can add report labels one at a time to your project by typing in the text for the label and then clicking **Add Label**.



You can alternatively switch the **Add Localized Labels** dialog into **Advanced Mode** which makes it possible to delete all existing report labels at once and to enter a large batch of report labels in a single operation.



Once you've added the required report labels to your PBIX project, they will appear in the translation grid. At that point, you can add and edit localized label translations just like any other type of translation in the translation grid.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	French [fr-FR]	German [de-DE]
Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Rapport Sur Les Ventes De Produits	Produktverkaufsbericht
Measure	Caption	Localized Labels[Sales Summary]	Sales Summary	Resumen De Ventas	Récapitulatif Des Ventes	Zusammenfassung
Measure	Caption	Localized Labels[Sales By Country]	Sales By Country	Ventas Por País	Ventes Par Pays	Umsatz Nach Ländern
Measure	Caption	Localized Labels[Sales By Product]	Sales By Product	Ventas Por Producto	Ventes Par Produit	Umsatz Nach Produkt
Measure	Caption	Localized Labels[Filter By Country]	Filter By Country	Filtrar Por País	Filtrer Par Pays	Filtern Nach Land
Measure	Caption	Localized Labels[Top 10 Customers]	Top 10 Customers	Top 10 Clientes	Top 10 Des Clients	Top 10 Kunden
Measure	Caption	Localized Labels[Product Catalog]	Product Catalog	Catálogo De Productos	Catalogue De Produits	Produktkatalog
Measure	Caption	Localized Labels[Set Product Filter]	Set Product Filter	Establecer Filtro De Producto	Définir Le Filtre De Produit	Produktfilter Einstellen
Measure	Caption	Localized Labels[Set Country Filter]	Set Country Filter	Establecer Filtro De País	Définir Le Filtre Par Pays	Länderfilter Einstellen
Measure	Caption	Localized Labels[Time Slices]	Time Slices	Intervalos De Tiempo	Tranches De Temps	Zeitscheiben

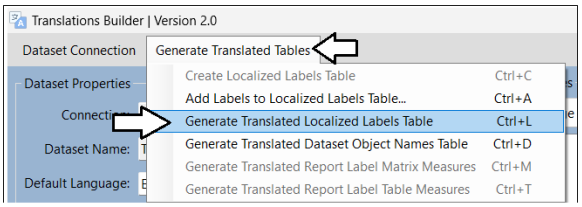
As you learned earlier, Translations Builder only populates the translation grid with dataset objects that are not hidden from **Report View**. The measures in the **Localized Labels** table are hidden from **Report View** and they provide the one exception to the rule that excludes hidden objects from being displayed in the translation grid.

One valuable aspect of the **Localized Labels** table strategy is that report labels can be created, managed and stored in the same PBIX project file that holds the metadata translations for the names of tables, columns and measures. The **Localized Labels** table strategy is able to merge metadata translations and report label translations together in a unified experience in the translation gird. There is no need to distinguish between metadata translations and report label translations when it comes to editing translations or when using Translations Builder features to generate machine translations.

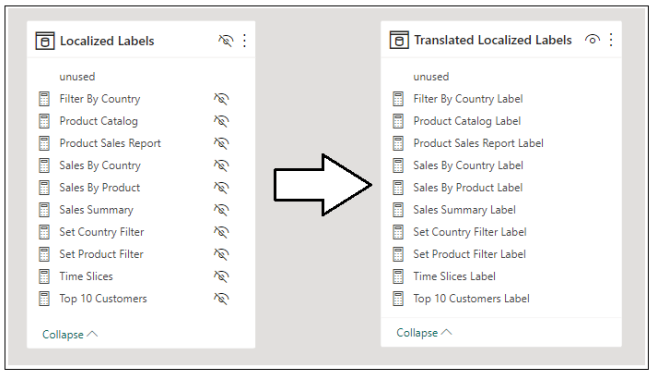
In the Power BI community, there are other popular localization techniques that track report label translations in a separate CSV file. While these techniques work just fine, they are not as streamlined as the **Localized Labels** table strategy because report label translations must be stored in a separate CSV file. In other words, report label translations must be created separately and managed differently from the metadata translations in a PBIX project. The **Localized Labels** table strategy allows for report label translations and metadata translations to be stored together and managed in the exact same way.

Generating the Translated Localized Labels Table

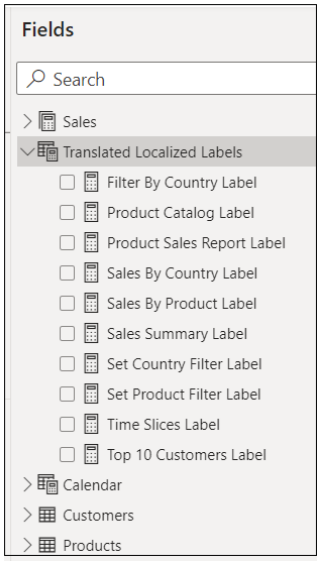
The **Localized Labels** table contains a measure with translations for each report label in a PBIX project. However, the measures inside the **Localized Labels** table are hidden and are not intended to be used directly by report authors. Instead, the **Localized Labels** table strategy is based on running code to generate a second table named **Translated Localized Labels** with measures that are meant to be used directly on a report page. You can create this table by executing the **Generate Translated Localized Labels Table** command.



The first time you execute the **Generate Translated Localized Labels Table** command, Translations Builder executes code to create the **Translated Localized Labels** table and populate it with measures. After that, executing the **Generate Translated Localized Labels Table** command will delete all the measures in the **Translated Localized Labels** table and then recreate them to synchronize all the report label translations between the **Localized Labels** table and the **Translated Localized Labels** table.



Unlike the **Localized Labels** table, the **Translated Localized Labels** table is not hidden from **Report View**. In fact, it's quite the opposite. The **Translated Localized Labels** table provides measures that are intended to be used to surface report labels in a report. Here is how the **Translated Localized Labels** table appears to a report author in the **Fields** pane when the report is in **Report View** in Power BI Desktop.



You can see that every measure in the **Translated Localized Labels** table has a name that ends with the world **Label**. The reason for this is that two measures inside the same dataset cannot have the same name. Measure names must be unique on a project-wide basis so it's not possible to create measures in the **Translated Localized Labels** table that have the same name as the measures in the **Localized Labels** table. the **Localized Labels** table strategy appends the word **Label** to all measure names in the **Translated Localized Labels** table to ensure their names are unique.

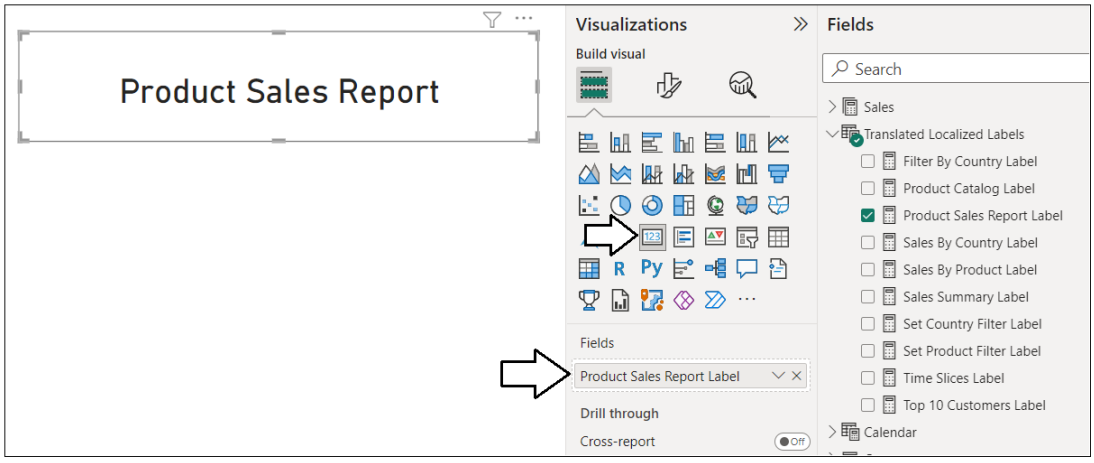
If you examine the machine-generated DAX expressions for measures inside the **Translated Localized Labels** table, you will see they are based on the same DAX pattern shown earlier. First, this pattern uses the DAX functions **USERCULTURE** and **LEFT** to determine the 2-character language ID of the current user. The pattern then uses the **SWITCH** function to return the best translation for the current user. Note that this DAX pattern falls back on the translations of the default language for the project's underlying dataset if no match is found with a secondary language.

```
Product Sales Report Label = SWITCH(LEFT(USERCULTURE(), 2),
   "es", "Informe De Ventas De Productos",
   "fr", "Rapport Sur Les Ventes De Produits",
   "de", "Produktverkaufsbericht",
   "nl", "Productverkooprapport",
   "it", "Report Sulle Vendite Dei Prodotti",
   "pt", "Relatório De Vendas De Produtos",
   "el", "Αναφορά Πωλήσεων Προϊόντων",
   "ru", "Отчет О Продажах Продукции",
   "ja", "製品販売レポート",
   "zh", "产品销售报告",
   "hi", "उत्पाद बिक्री रिपोर्ट",
   "he", "דוח מכירות מוצרים",
   "af", "Produkverkope Verslag",
   "Product Sales Report"
)
```

You must remember to execute **Generate Translated Localized Labels Table** anytime you make changes to the **Localized Labels** table. Keep this in mind because it is easy to forget. You should also resist any temptation to edit the DAX expressions for measures in the **Translated Localized Labels** table. Any edits you make will be lost as all the measures in this table are deleted and recreated each time you execute **Generate Translated Localized Labels Table**.

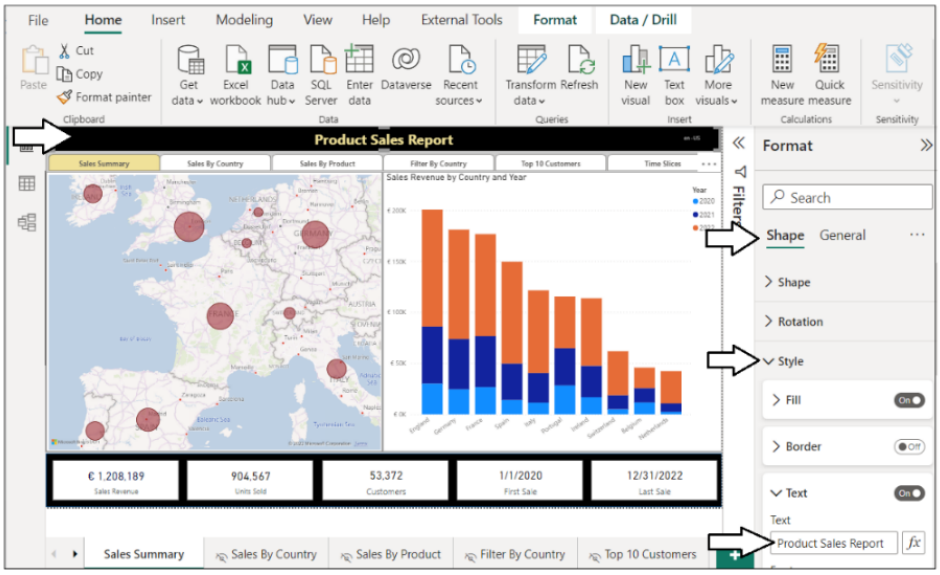
Surfacing Localized Labels on a Report Page

As you have learned, report labels are implemented as dynamic measures in the **Translated Localized Labels** table. That makes them very easy to surface in a Power BI report. For example, you can add a **Card** visual to a report and then configure its **Fields** data role in the **Visualizations** pane with a measure from the **Translated Localized Labels** table.

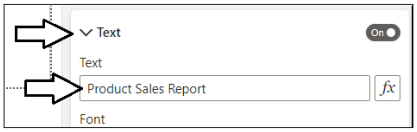


As Microsoft continues to evolve the report design experience in Power BI Desktop, there have been several new enhancements which make it easier for content creators to build multi-language reports. One essential aspect of these enhancements is a greater ability to use measures in a report layout to configure dynamic property values for report elements such as visuals and shapes.

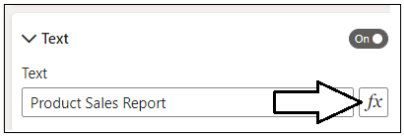
The live demo project uses a **Rectangle** shape to display the localized report label for the report title. The following screenshot shows how to select a **Rectangle** shape and then navigate to configure its **Text** property value in **Shape** > **Style** > **Text** section in the **Format** pane.



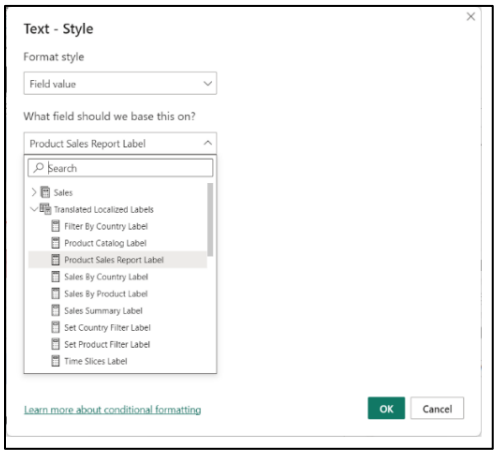
The **Text** property of a **Rectangle** shape can be configured with a hard-coded string as shown in this screenshot.



However, you already know you must avoid hard-coding text values into the report layout when creating multi-language reports. If you click on the ***fx*** button to the right, Power BI Desktop will display a dialog which allows you to configure the **Text** property of the **Rectangle** shape using a measure from the **Translated Localized Labels** table .



Once the **Text - Style** dialog appears, you can navigate to the **Translated Localized Labels** table and select any measure.

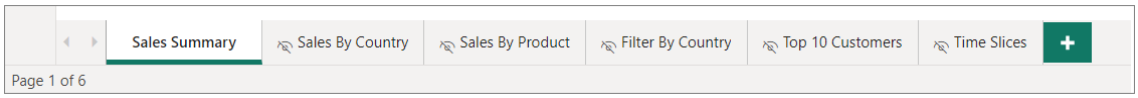


You can use the same technique to localize a visual **Title** using a measure from the **Translated Localized Labels** table.

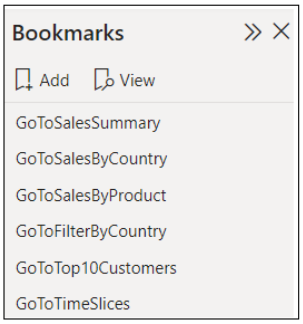
Adding Support for Page Navigation

As you recall, you cannot display page tabs to the user in a multi-language report because page tabs in a Power BI report 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 technique where you add a navigation menu using buttons. When the user clicks on a button, the button is configured to apply a bookmark to navigate to another page. Let's step through the process of building a navigation menu that supports localization using measures from the **Localized Labels** table.

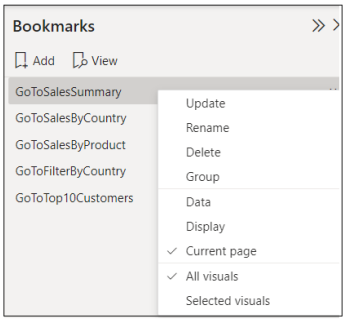
The first thing you need to do when building a custom navigation menu is to hide every page in the report except for the first page which acts as the report landing page.



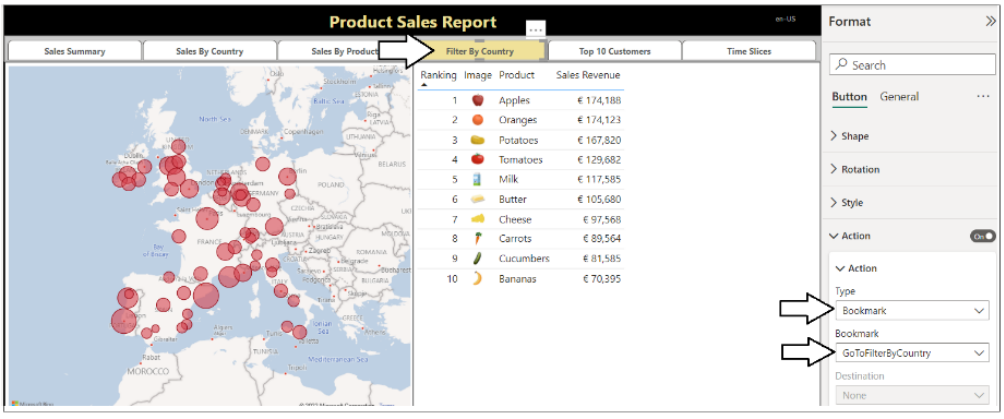
Next, create a set of bookmarks. Each bookmark should be created to navigate to a specific page. The **live demo** sample demonstrates this technique by adding a bookmark for each page supported by the navigation menu.



When creating bookmarks for navigation, you should disable **Data** and **Display** and only enable **Current Page** behavior.

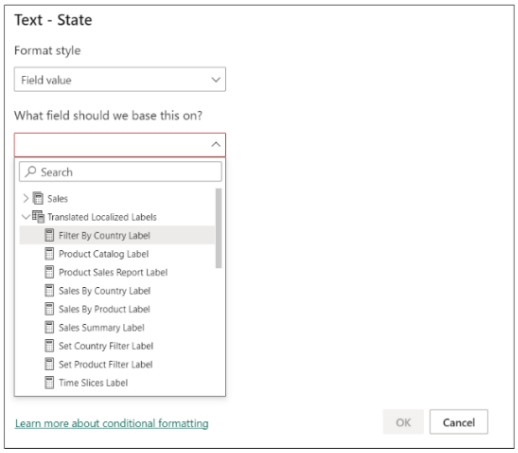


The next step is to configure each button in the navigation menu to apply a bookmark to navigate to a specific page.



After you've configured a button with a bookmark, the final step is to configure the **Text** property with a localized label.

The **Text** property of each button can be configured with a measure from the **Translated Localized Labels** table.



At this point, you've learned how to create the **Localized Labels** table and how to add localized report labels to a PBIX project. You also learned how to generate the **Translated Localized Labels** table and to bind the measures in that table to report elements such as Card visuals, shapes and buttons. These are the localization techniques you will continue to use as you create and maintain reports that are required to support multiple languages. Now this section will conclude with some general advice for building Power BI reports that support multiple languages.

Using Best Practices When Localizing Power BI Reports

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 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.

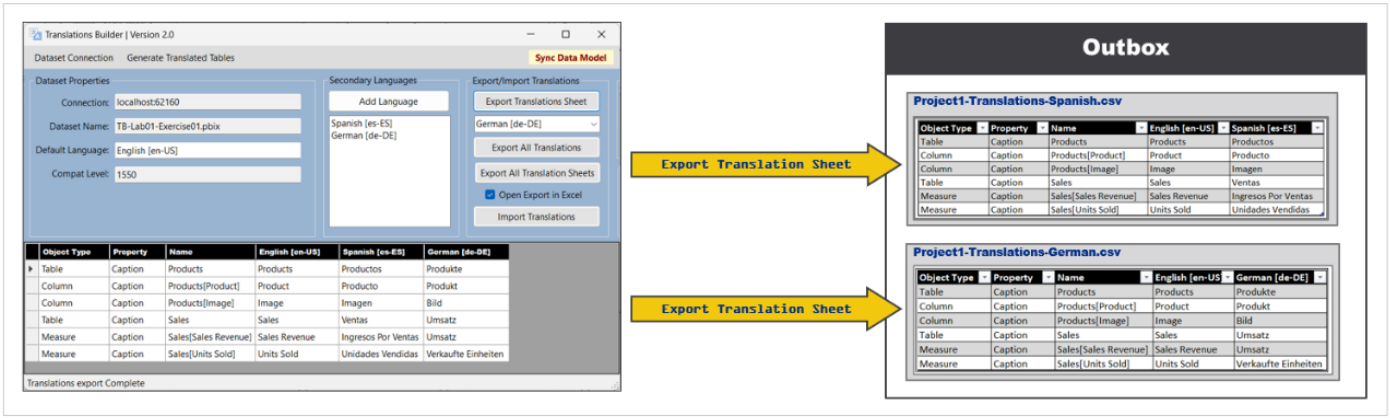
Adding a healthy degree of padding to localized labels is the norm when developing Internationalized software and it's essential that you test your reports with each language you plan to support. In essence, you need to ensure your report layouts looks the way you expect with any language you have chosen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

Enabling Workflows for Human Translation using Export and Import

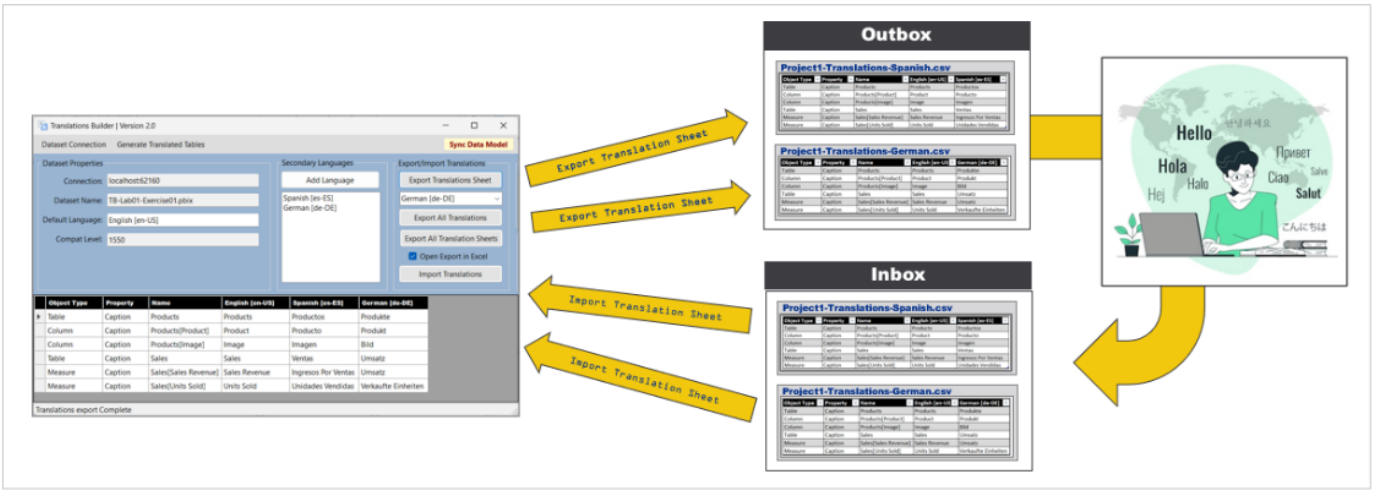
Up to this point, you have learned to structure a Power BI report and its underlying dataset to support translations. You also learned how to complete this work in a quick and efficient manner by using Translations Builder and by generating machine translations. However. It's important to acknowledge that machine-generated translations alone will not be adequate for many production scenarios. You need to find a way to integrate other people acting as translators into a human workflow process.

> The Translations Builder introduces the concept of a ***translation sheet***. A translation sheet is a CSV file that you generate with an export operation to send out to a translator. The human acting as a translator performs the work to update the translation sheet and then returns it back to you. You can then execute an import command to integrate the changes made by a translator back into the current PBIX project's dataset.

When you click the **Export Translation Sheet** button, Translations Builder generates a CSV file for the selected language using a special naming format (e.g. **PbixProjectName-Translations-Spanish.csv**) which includes the dataset name and the language for translation. The generated translation sheet file is saved to a special folder known as the **Outbox** folder.



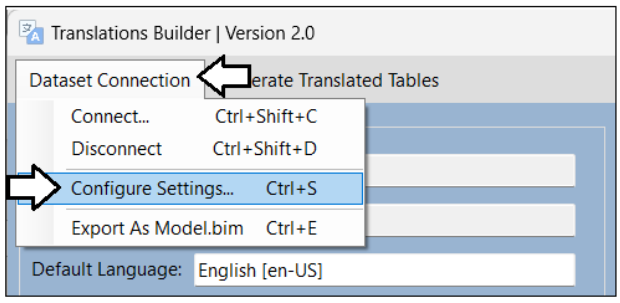
As you will see, human translators can makes edits to a translation sheet using Microsoft Excel. Once you've received an updated translation sheet back from a translator you can copy it to the **Inbox** folder. Translations Builder provides an **Import Translations** command to integrate those updated translations back into the dataset for the current project.



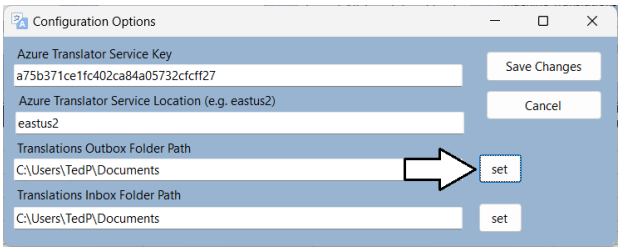
Configuring Target Folders for Import and Export Operations

If you're required to work with an external team of translators, you will need to manage the translation sheet files that are generated and sent to translators as well as those translations sheet files that are returned and ready for import. Translations Builder allows you to configure the location of the **Output** folder and the **Inbox** folder to assist with the file management of translations sheets.

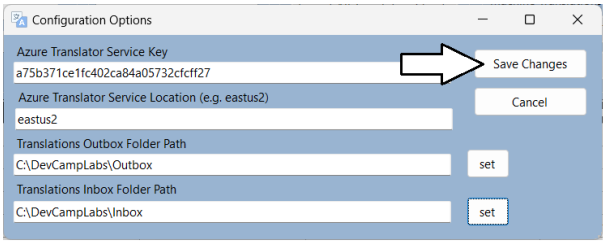
Let's sat you'd like to configure settings in Translations Builder so that you can decide which folders on your local hard drive are used as targets for export and import operations. You can drop down the **Dataset Connection** menu and click **Configure Settings** to display the **Configuration Options** dialog.



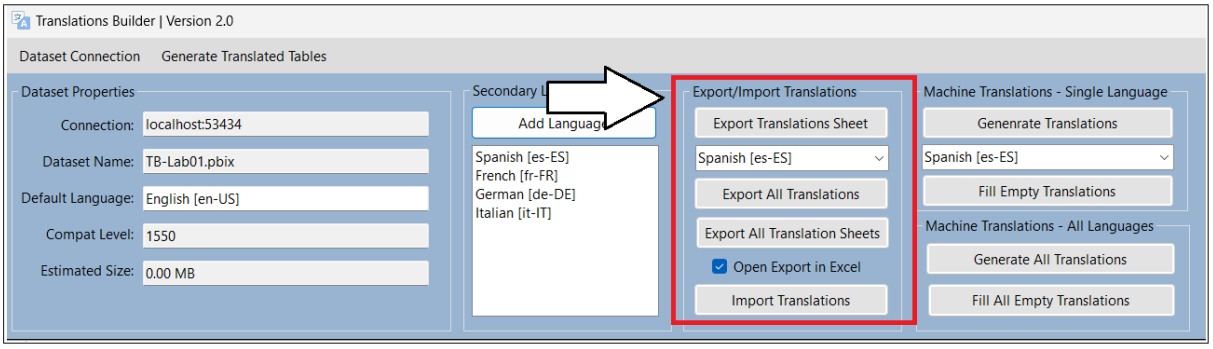
By default, folder paths for the **Outbox** folder and **Inbox** folder are configured to target the current user's **Documents** folder. Click the **set** button to the right to update the setting for **Translations Outbox Folder Path**.



Once you have configured the **Outbox** folder path and the **Inbox** folder path the way you like, click **Save Changes**.

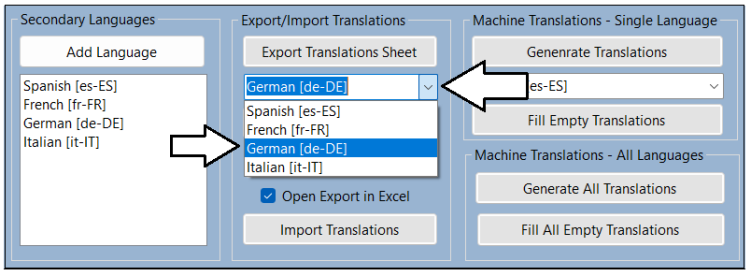


After you have configured the folder paths for **Outbox** and **Inbox**, you can begin to export and import translation sheets. As you can see, the **Export/Import Translations** section provides the commands for export and import operations.

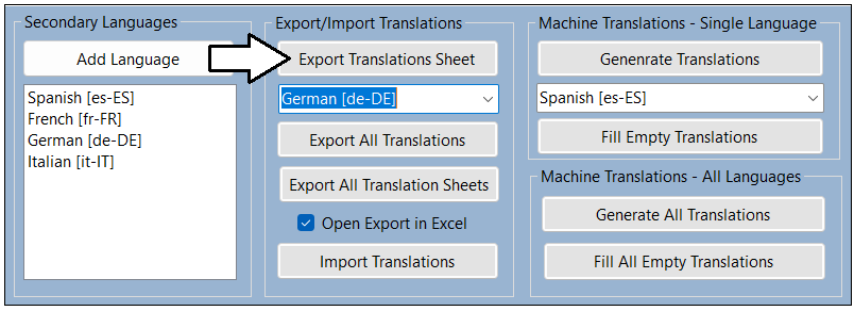


Exporting a Translation Sheet for a Secondary Language

Let's start by generating a translation sheet for a single language. First, you should drop down the selection menu under the **Export Translations Sheet** button and select a language such as **German [de-DE]**.



After selecting a language, you can click the **Export Translations Sheet** button to generate a translation sheet for that language.

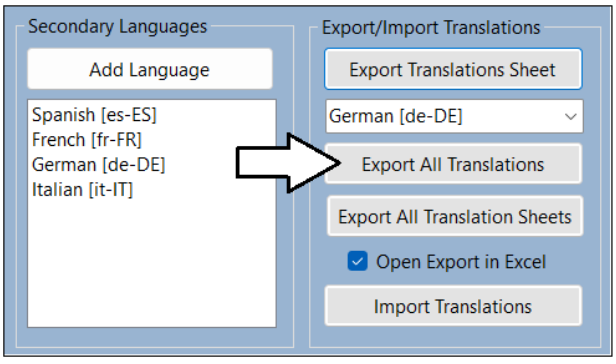


When generating translation sheets in this manner, you can enable or disable the **Open Export in Excel** option. When this option is enabled, Translations Builder will open the exported CSV file in Excel each time you generate a translation sheet. The **Open Export in Excel** option makes it possible to quickly view and edit the contents of a translation sheet.

	A	B	C	D	E
1	Object Type	Property	Name	English [en-US]	German [de-DE]
2	Table	Caption	Products	Products	Produkte
3	Column	Caption	Products[Product]	Product	Produkt
4	Column	Caption	Products[Image]	Image	Bild
5	Table	Caption	Sales	Sales	Umsatz
6	Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Umsatz
7	Measure	Caption	Sales[Units Sold]	Units Sold	Verkaufte Einheiten
8	Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Produktverkaufsbericht
9	Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Produktliste Nach Umsatz

Exporting the Master Translation Sheet

The **Export Translation Sheet** command that you've just seen will export a translation sheet with translations for just one secondary language at a time. You can alternatively use the **Export All Translations** command which generates a master translation sheet with all the secondary languages and translations that have been added to the current project.

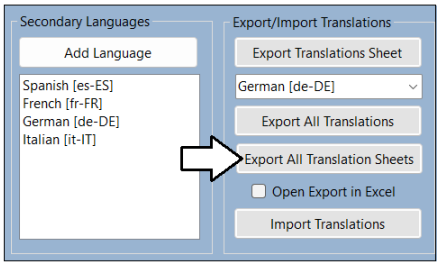


When you click the **Export All Translations** button, Translations Builder generates a CSV file for the master translation sheet named **PbixProjectName-Translations-Master.csv**. When the master translations sheet opens in Microsoft Excel, you can see all secondary language columns and all translations. You can think of the master translation sheet as a backup of all the translation work you have done on a project-wide basis.

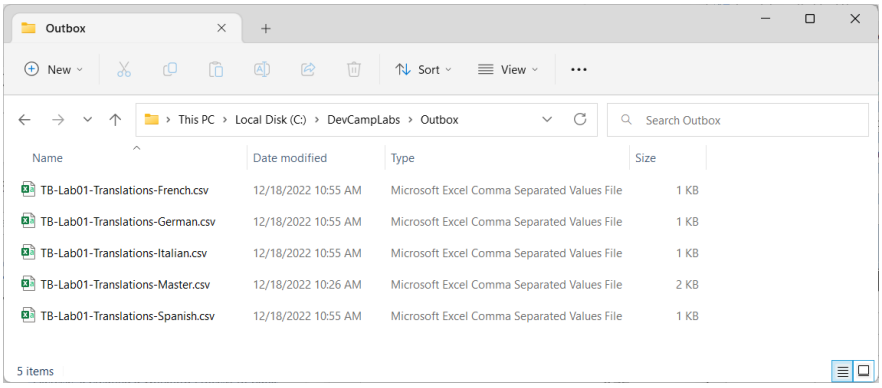
	A	B	C	D	E	F	G	H
1	Object Type	Property	Name	English [en-US]	Spanish [es-ES]	French [fr-FR]	German [de-DE]	Italian [it-IT]
2	Table	Caption	Products	Products	Productos	Produits	Produkte	Prodotti
3	Column	Caption	Products[Product]	Product	Producto	Produit	Produkt	Prodotto
4	Column	Caption	Products[Image]	Image	Imagen	Image	Bild	Immagine
5	Table	Caption	Sales	Sales	Ventas	Ventes	Umsatz	Vendite
6	Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Chiffre D'Affaires	Umsatz	Ricavi Delle Vendite
7	Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Unités Vendues	Verkaufte Einheiten	Unità Vendute
8	Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Rapport Sur Les Ventes De Produits	Produktverkaufsbericht	Report Sulle Vendite Dei Prodotti
9	Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas	Liste De Produits Par Chiffre D'Affaires	Produktliste Nach Umsatz	Elenco Dei Prodotti Per Fatturato Delle Vendite

Exporting Translation Sheets for All Secondary Languages

The final export command you should understand is the **Export All Translation Sheets** command which is provided to assist with the quick generation and the management of outbound translation sheet files.

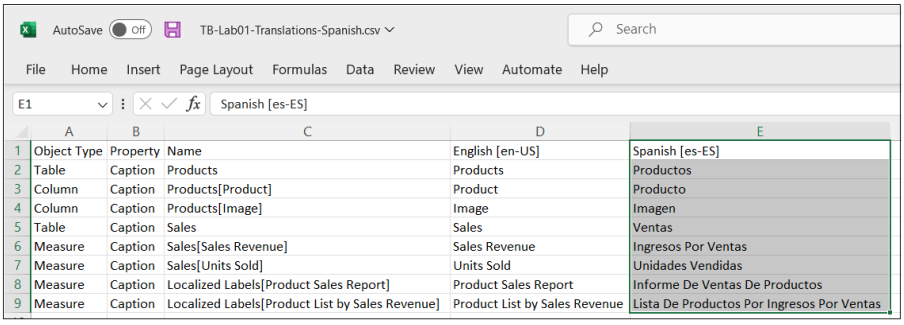


When you execute the **Export All Translation Sheets** command, it generates the complete set of translation sheets to be sent to translators. If you examine the **Outbox** folder, you should see that a sperate translation sheet has been generated for each secondary language that has been included in the current project.



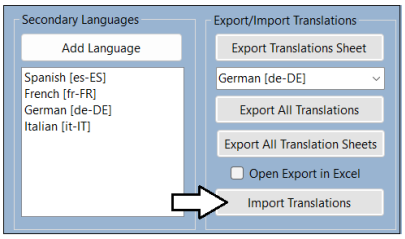
Importing Translation Sheets

Imagine a scenario where you have generated a translation sheet to send to a Spanish translator. When opened in Excel, this translation sheet appears as the one shown in the following screenshot.

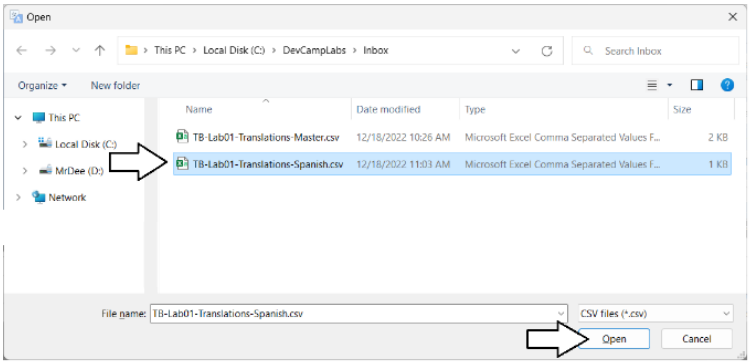


	A	B	C	D	E
1	Object Type	Property	Name	English [en-US]	Spanish [es-ES]
2	Table	Caption	Products	Products	Productos
3	Column	Caption	Products[Product]	Product	Producto
4	Column	Caption	Products[Image]	Image	Imagen
5	Table	Caption	Sales	Sales	Ventas
6	Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas
7	Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas
8	Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos
9	Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas

The job of the translator is to review all translations in the fifth column and to make updates where appropriate. From the perspective of the translator, the top row with column headers and the first four columns should be treated as read-only values. Once you receive the translation sheet back from the translator with updates to the translations in the fifth column, you can return to Translations Builder and click the **Import Translations** button.



Remember to close translation sheet files in Microsoft Excel before attempting to import them with Translations Builder to prevent errors. In the **Open** file dialog, select the translation sheet file and click **Open**.



You should see that your updates to the Spanish translation sheet now appear in the translation grid.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	French [fr-FR]
Table	Caption	Products	Products	Productos	Produits
Column	Caption	Products[Product]	Product	Producto	Produit
Column	Caption	Products[Image]	Image	Imagen	Image
Table	Caption	Sales	Sales	Ventas	Ventes
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Chiffre D'Affaires
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Unités Vendues
Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Rapport Sur Les Ventes De Produits
Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas	Liste De Produits Par Chiffre D'Affaires

Importing a Master Translation Sheet

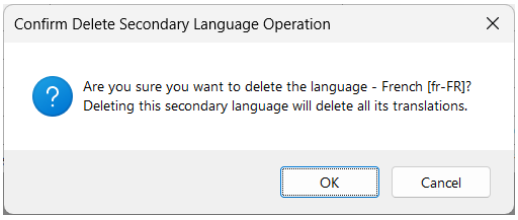
In many scenarios, it makes sense to import updated translation sheets that only contain translations for a single secondary language. However, you can also import a master translation sheet with which has multiple columns for secondary languages. Therefore, the master translation sheet can provide an effective way to backup and restore the work you have done with translations on a project-wide basis.

To make this point, let's move through a simple scenario in which you have already generated the master translation sheet for a project that includes several secondary languages. Now imagine you delete French as a language from the project by right-clicking on the **French [fr-FR]** column header and selecting **Delete Secondary Language**.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	French [fr-FR]	German [de-DE]
Table	Caption	Products	Products	Productos	Produits	
Column	Caption	Products[Product]	Product	Producto	Produit	Produkt
Column	Caption	Products[Image]	Image	Imagen	Image	Bild
Table	Caption	Sales	Sales	Ventas	Ventes	Umsatz
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Chiffre D'Affaires	Umsatz
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Unités Vendues	Verkaufte Einheiten
Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Rapport Sur Les Ventes De Produits	Produktverkaufsbericht
Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas	Liste De Produits Par Chiffre D'Affaires	Produktliste Nach Umsatz

Delete Secondary Language

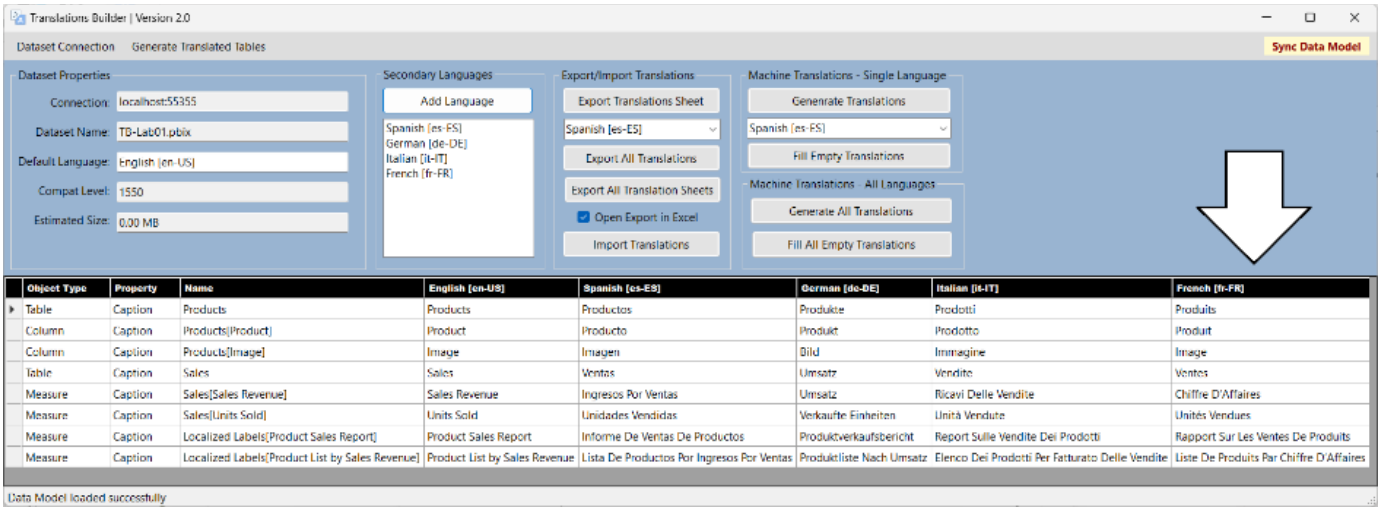
When you attempt to delete the column for a secondary language, Translations Builder will prompt you with the **Confirm Delete Secondary Language Operation** dialog.



You can click **OK** to continue and complete the delete operation. After you confirm the delete operation, you will see that the column for French has been removed from the translations grid. Behind the scenes, Translations Builder has also deleted all the French translations from the project.

Object Type	Property	Name	English [en-US]	Spanish [es-ES]	German [de-DE]	Italian [it-IT]
Table	Caption	Products	Products	Productos	Produkte	Prodotti
Column	Caption	Products[Product]	Product	Producto	Produkt	Prodotto
Column	Caption	Products[Image]	Image	Imagen	Bild	Immagine
Table	Caption	Sales	Sales	Ventas	Umsatz	Vendite
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Umsatz	Ricavi Delle Vendite
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Verkaufte Einheiten	Unità Vendute
Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Produktverkaufsbericht	Report Sulle Vendite Dei Prodotti
Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas	Produktliste Nach Umsatz	Elenco Dei Prodotti Per Fatturato Delle Vendite

Continuing with our scenario, you sense that something has gone wrong and you exclaim “Oh Mon Dieu!”. That’s because you just realized that you have deleted all the French translations accidently. Fortunately, you previously generated a master translation sheet that contains the French translations. This means you have not lost all your work. If you import the master translation sheet, the **French [fr-FR]** column should reappear as the last column on the right.



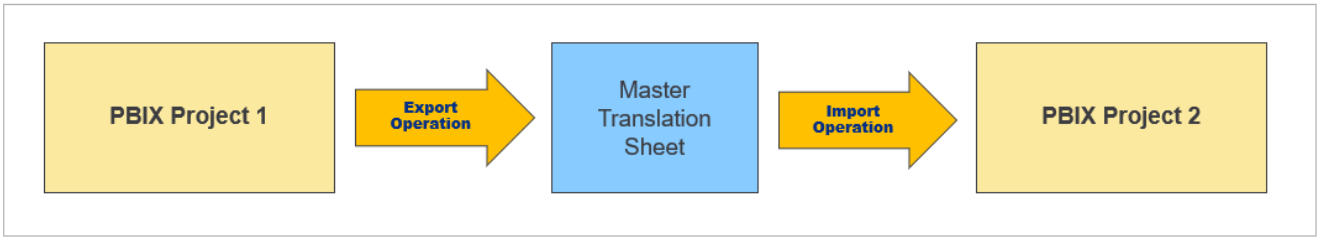
Object Type	Property	Name	English [en-US]	Spanish [es-ES]	German [de-DE]	Italian [it-IT]	French [fr-FR]
Table	Caption	Products	Products	Productos	Produkte	Prodotti	Produits
Column	Caption	Products[Product]	Product	Producto	Produkt	Prodotto	Produit
Column	Caption	Products[Image]	Image	Imagen	Bild	Immagine	Image
Table	Caption	Sales	Sales	Ventas	Umsatz	Vendite	Ventes
Measure	Caption	Sales[Sales Revenue]	Sales Revenue	Ingresos Por Ventas	Umsatz	Ricavi Delle Vendite	Chiffre D'Affaires
Measure	Caption	Sales[Units Sold]	Units Sold	Unidades Vendidas	Verkaufte Einheiten	Unità Vendute	Unités Vendues
Measure	Caption	Localized Labels[Product Sales Report]	Product Sales Report	Informe De Ventas De Productos	Produktverkaufsbericht	Report Sulle Vendite Dei Prodotti	Rapport Sur Les Ventes De Produits
Measure	Caption	Localized Labels[Product List by Sales Revenue]	Product List by Sales Revenue	Lista De Productos Por Ingresos Por Ventas	Produktliste Nach Umsatz	Elenco Dei Prodotti Per Fatturato Delle Vendite	Liste De Produits Par Chiffre D'Affaires

Managing Dataset Translations at Enterprise Level

In the previous section, you learned how to import translations from a master translation sheet. You have seen that the behavior of the **Import Translations** command is programmed to automatically add a secondary language along with its translations to a PBIX project if it is found in the translation sheet but not in the target project. The logic that has been programmed into the **Import Translations** command goes even further and this logic makes it possible to create an enterprise-level master translation sheet which can be imported at the start when you create a new PBIX project.

Imagine you have two PBIX projects that have a similar data model in terms of the tables, columns and measures. In the first project, you have already done all the work to add metadata translations for all the non-hidden dataset objects. In the second project, you have not yet started to add any support for any secondary languages or translations. What would happen if you exported the master translation sheet from the first project and then imported this master

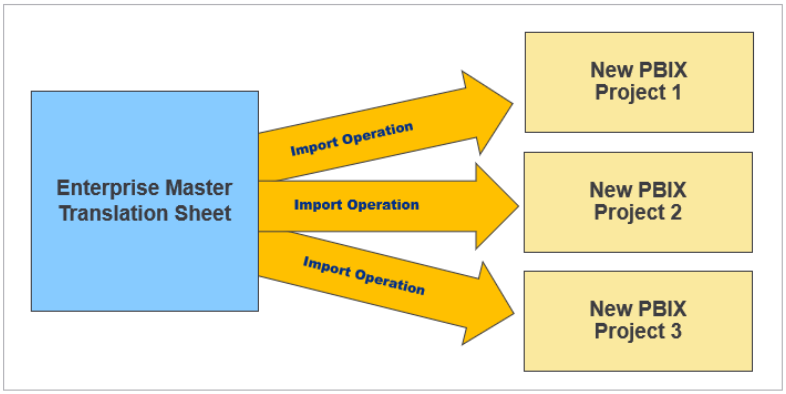
translation sheet into the second project? It would provide a quick way to copy the localization and translations work you have done from one PBIX project to another.



The logic behind the **Import Translations** command starts by determining whether there are any secondary languages in the translation sheet that are not in the target PBIX project. As you have seen, it automatically add any secondary languages not already present in the target project. After that, the code in the **Import Translations** command moves down the translation sheet row by row. For each row, it determines whether dataset object in the CSV file matches a dataset objects of the same name in the PBIX project. When a match is found, the **Import Translations** command then copies all the translations for that dataset object into the PBIX project. If no match is found, then the **Import Translations** command ignores that row and continues down to the next row.

The logic behind the **Import Translations** command provides special treatment for report labels that have been added to the **Localized Labels** table. If you import a translation sheet with one or more localized report labels into a new PBIX project, the **Import Translations** command will automatically create the **Localized Labels** table behind the scenes.

The ability of the **Import Translations** command to create the **Localized Labels** table and copy report labels into a target PBIX project provides the foundation for maintaining an enterprise-level master translation sheet with a reusable set of localized report labels you can use across all your PBIX projects. Each time you create a new PBIX project, you can simply import the enterprise-level translation sheet to instantly add the generalized set of localized report labels.



Implementing a Data Translations Strategy

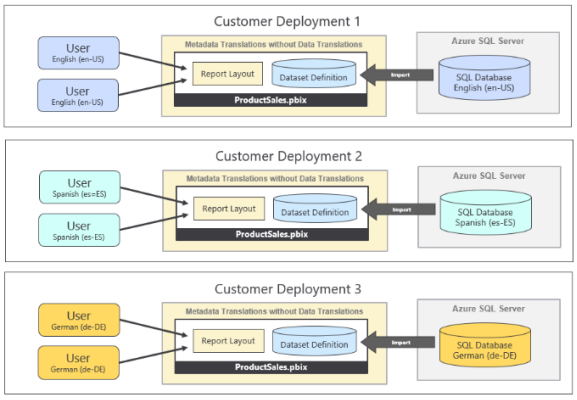
While all multi-language reports will require metadata translations and report label translations, you cannot assume the same for data translations. Some projects will require data translations and others will not. In order to determine whether your project will require data translations, you'll need to think through the use cases you plan to support with your reporting solution. You will find that adding support for data translations can involve a good deal of planning and effort. You might decide to only support data translations if they are a hard requirement for your project.

Implementing data translations is quite different from implementing metadata translations or report label translations. They are different because Power BI doesn't offer any localization features to assist you with data translations. Instead, you must implement a data translation strategy which typically involves extending the underlying datasource with extra columns to track translations for text in rows of data such as the names of products, categories and countries.

Determining Whether Your Solution Really Requires Data Translations

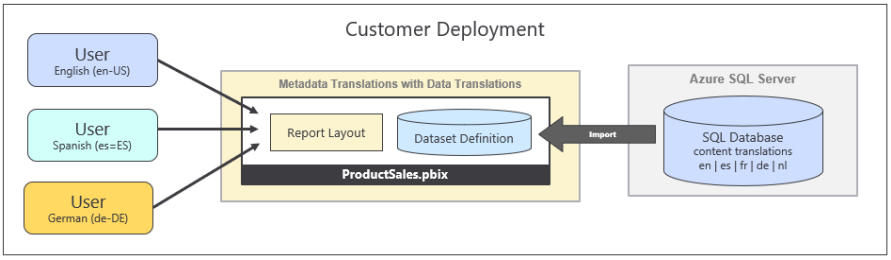
To determine whether you need to implement data translations, start by thinking about how your reporting solution will be deployed and think about the use case for its intended audience. That leads to a key question. **Will you have people who speak different languages looking at the same database instance?**

Imagine a scenario where you are developing a report template for a SaaS application with a well-known database schema. Now let's say some customer maintain their database instance in English while others maintain their database instances in other languages such as Spanish or German. There is no need to implement data translations in this use case as the data from any database instance only needs to be viewed by users in a single language.



The important observation is that each customer deployment uses a single language for its database and all its users. Both metadata translations and report label translations must be implemented in this use case so you can deploy a single version of the PBIX file across all customer deployments. However, there is no need to implement data translations when no database instance ever needs to be viewed in multiple languages.

Now let's examine a different use case which introduces the requirement of data translations. This is the use case for the **ProductSalesMultiLanguage.pbix** live demo which involves a single database instance containing sales performance data across several European countries. This reporting solution has the requirement to display its report in different languages while the data being analyzed is coming from a single database instance.



Once again, the key question to ask is whether you will have people who speak different languages looking at the same database instance. If the answer to that question is ***NO***, then you will not be required to implement data translations. If the answer to that question is ***YES***, then you should ask additional questions because there are other consideration you should think through before deciding whether it makes sense to implement data translations.

When you're considering whether to implement data translations, you should examine the text-based columns which are candidates for translation to determine how hard will it be to translate those text value to secondary languages. Columns with short text values for things like product names, product categories and country names are a good candidates for data translations because the values are short and easy to translate. What if you have a column for product descriptions where each row has two to three sentences of text. While you can provide translations for product descriptions, they will require more effort to generate high quality translations. In general, columns with longer text values are less ideal as candidates for data translations.

You should also consider the number of distinct column values that will require translation. You can easily translate product names in a database that holds 100 products. You can probably translate product names when the number gets up to 1000. However, what happens if the number of translated values reaches 10,000 or 100,000. If you cannot rely on machine-generate translations. your translation team might have trouble scaling up to handle that volume of human translations.

You also have to consider that your commitment to implement data translations might require on-going maintenance. Every time someone adds a new record to the underlying database, there is the potential to introduce new text values that require translation. This is very different from implementing metadata translations or report label translations where you create a finite number of translations and, after that point, your work is done. Metadata translations and report label translations don't require on-going maintenance as long as the underlying dataset schema and the report layout remain the same.

In summary, there are many factors that go into deciding whether you should implement data translations. You must decide whether it's worth the time and effort required to implement data translations properly. In certain scenarios, you might decide that implementing metadata translations and report label translations goes far enough. If your primary goal is to make your reporting solution compliant with laws or regulations, you might also find that implementing data translations is not a requirement.

Extending the Datasource Schema to Support Data Translations

There are multiple ways to implement data translations in Power BI. The strategy shown here in this article represents just one possible approach. There is no single right answer when designing a data translation strategy for Power BI. However, some data translation strategies are better than others. Whatever approach you choose, make sure it scales in terms of performance. You should also ensure your strategy scales in terms of the overhead required to add support for new secondary languages as part of the on-going maintenance.

An earlier version of this article demonstrated a solution for implementing data translations based on adding extra rows to tables containing text-based columns such as product names. This solution relied on filtering rows to select a language. However, this strategy is limited in terms of scalability because it requires many-to-many relationships between tables. Fortunately, the strategy demonstrated in this version of this article presents new and more scalable solution. This new strategy for implementing data translations is made possible by the new feature recently added to Power BI Desktop known as ***Field Parameters***.

Let's start by making modifications to the underlying datasource. For example, the **Products** table can be extended with extra columns with translated product names to support data translations. In this case the **Products** table has been extended with a separate column with product name translations in English, Spanish, French and German.

	ProductId	ProductTranslationEnglish	ProductTranslationSpanish	ProductTranslationFrench	ProductTranslationGerman
1	1	Apples	Manzanas	Pommes	Äpfel
2	2	Bananas	Plátanos	Bananes	Bananen
3	3	Oranges	Naranjas	Oranges	Orangen
4	4	Carrots	Zanahoria	Carottes	Möhren
5	5	Cucumbers	Pepinos	Concombres	Gurken
6	6	Potatoes	Papas	Pomme De Terre	Kartoffeln
7	7	Tomatoes	Tomates	Tomates	Tomaten
8	8	Milk	Leche	Lait	Milch
9	9	Butter	Mantequilla	Beurre	Butter
10	10	Cheese	Queso	Fromage	Käse

Note that the design approach shown here is using a three-part naming convention for table column names used to hold data translations. The naming convention parses together the entity name (e.g. **Product**) together with the word **Translation** and the language name (e.g. **Spanish**). For example, the column which contains product names translated into Spanish is **ProductTranslationSpanish**. While using this three-part naming convention is not a hard requirement for implementing data translations, Translations Builder is able to give these columns special treatment.

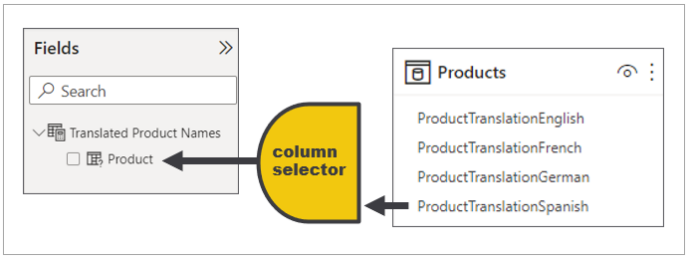
Implementing Data Translation using Field Parameters

Let's start with a simple question and a somewhat complicated answer. What is a Field Parameter?

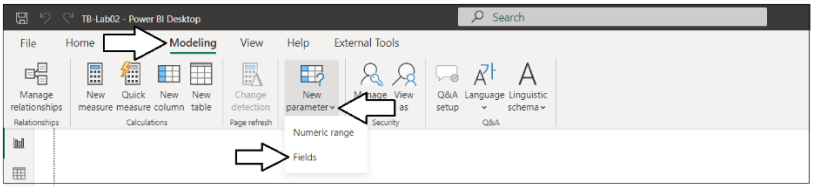
A Field Parameter is a table in which each row represents a field and where each these fields must be defined as either a column or a measure. In one sense, a Field Parameter is just a pre-defined set of fields. Given that these fields are represented by rows in a table, the set of fields of a Field Parameter supports filtering. Therefore, you can think of a Field Parameter as a filterable set of fields.

When you create a Field Parameter, you have the option of populating the fields collection using either measures or columns. Most of the good examples out there on the Internet from popular Power BI bloggers involve creating Field Parameters using measures. However, when using Field Parameters to implement data translations, you will be using columns instead of measures. The primary role that Field Parameters play in implementing data translations is providing a single, unified field to be used in report authoring that can be dynamically switched between source columns behind the scenes.

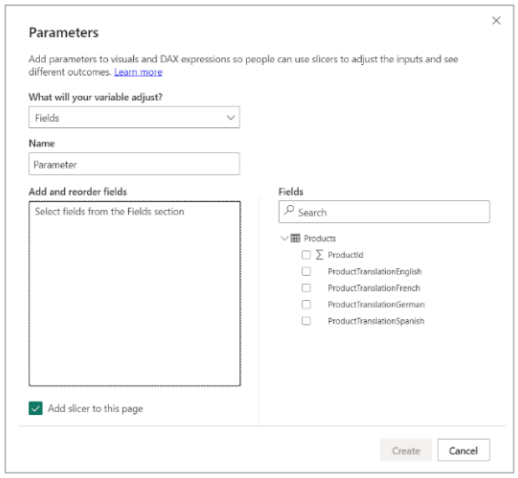
Before the introduction of Field Parameters, it was challenging to implement data translations efficiently in Power BI. That's because Power BI did not provide any way to evaluate a calculated column dynamically after the dataset loading process has completed. The advantage of using Field Parameters is that they provide a column selector mechanism that can be used to dynamically switch back and forth between multiple source columns in the underlying datasource.



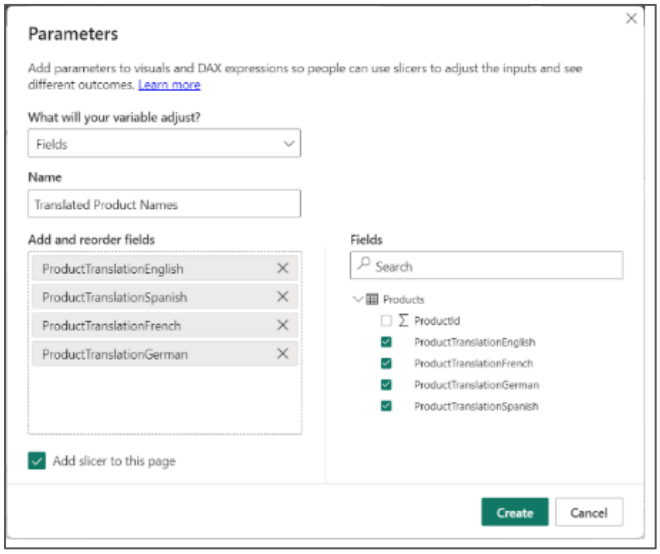
To create a Field Parameter in Power BI Desktop, navigate to the **Modeling** tab and select **New parameter > Fields**.



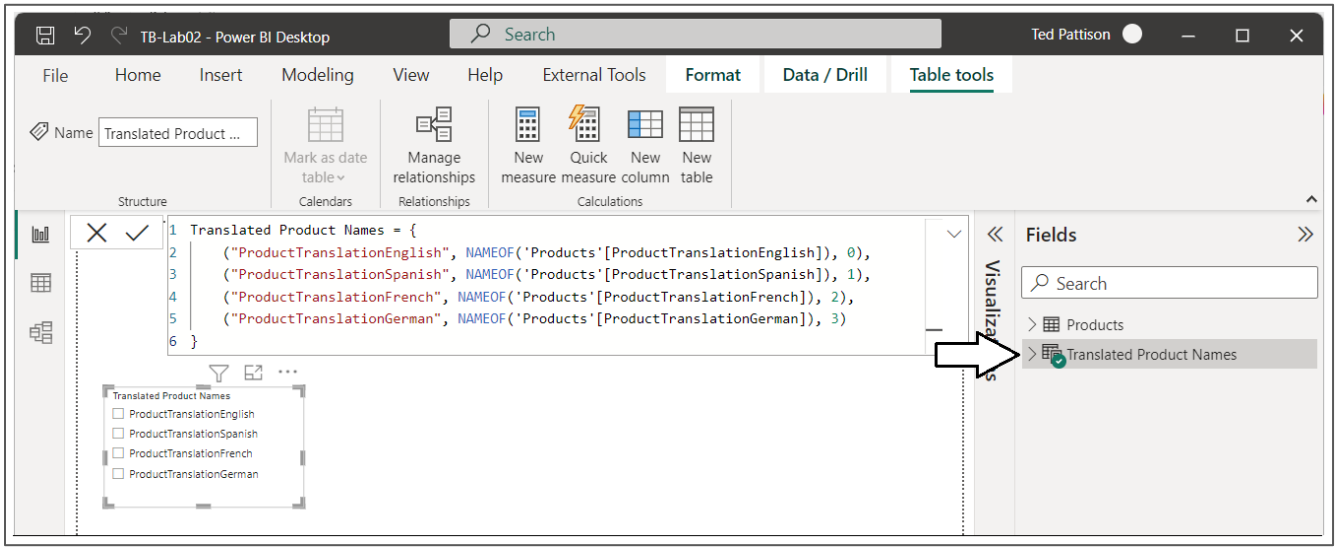
When you are prompted by the **Parameters** dialog, you can supply a **Name** for the new Field Parameter. You can also add the set of translated name columns from the **Products** table using the **Fields** pane on the right.



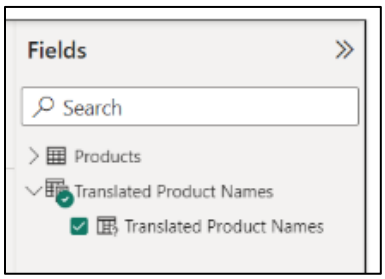
For our scenario, let's create a new Field Parameter named **Translated Product Names**. Let's also populate the fields connection of this Field Parameter with the four columns from the **Products** table with the translated product names. When you are just starting to experiment with Field Parameters, you should leave the **Add slicer to page** option enabled as it helps in running a few tests to build your understanding.



After you have created a new Field Parameter, it appears in the **Fields** list on the right as a new table. If you select a Field Parameter such **Translated Product Names** while in **Report** view, you should see the Field Parameter definition is based on a DAX expression as shown in the following screenshot.

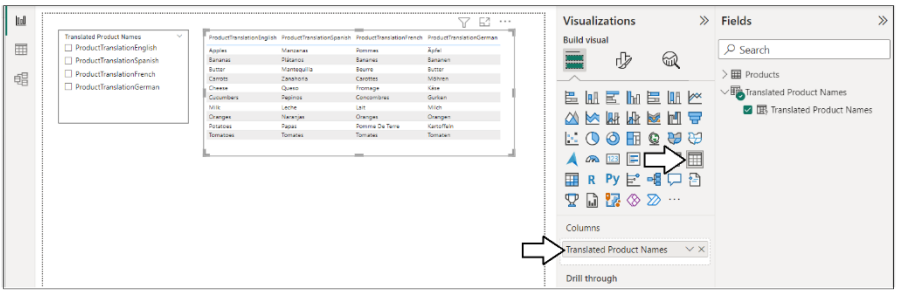


If you expand the **Fields** list while in **Report** view, you will see a single field with the same name as the parent table.

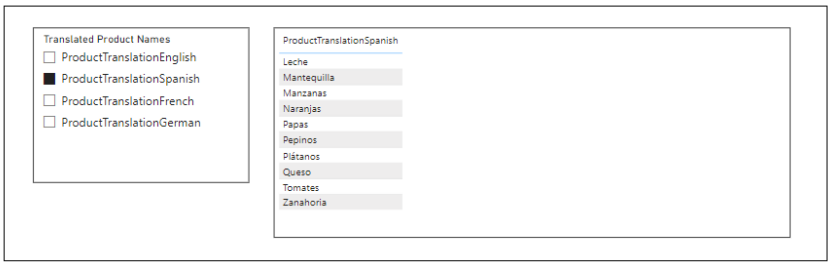


From a data modeling perspective, you can see that a Field Parameter is created as a table with a set of fields.

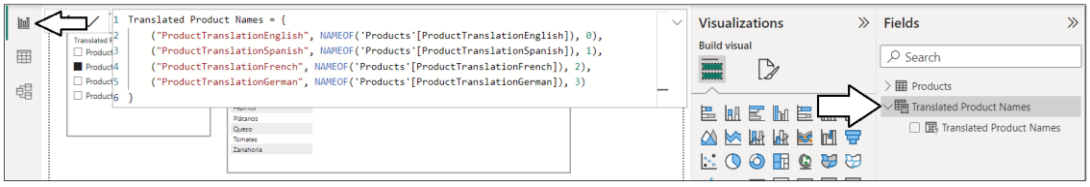
Let's conduct a quick experiment so you can better understand how Field Parameters work. Let's add a **Table** visual to the report page to the right of the slicer. Next, add the field inside the Field Parameter into the **Columns** data role of the **Table** visual. As long as nothing is selected in the slicer, the table visual displays all four source columns.



Now, let's select a specific column in the slicer. When you do, the slicer applies filtering that reduces the number of columns displayed in the table visual from four columns to a single column.



In the previous screenshot, you can see that the column values for product names have been translated into Spanish. However, there is still an issue with the column header. The column header still displays the column name from the underlying datasource which is **ProductTranslationSpanish**. The reason for this is that those column header values were hard-coded into the DAX expression when Power BI Desktop created the new Field Parameter.



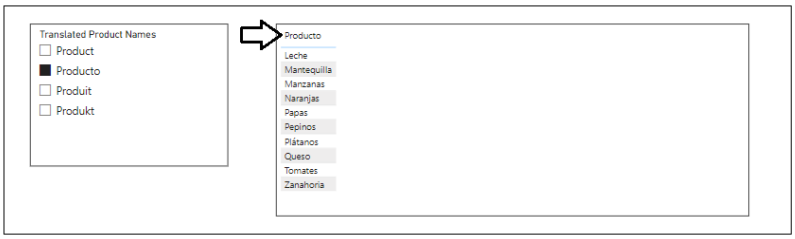
If you examine the DAX expression generated by Power BI Desktop, you will see the hard-coded column names from the underlying datasource such as **ProductTranslationEnglish** and **ProductTranslationSpanish**.

```
Translated Product Names = {
    ("ProductTranslationEnglish", NAMEOF('Products'[ProductTranslationEnglish]), 0),
    ("ProductTranslationSpanish", NAMEOF('Products'[ProductTranslationSpanish]), 1),
    ("ProductTranslationFrench", NAMEOF('Products'[ProductTranslationFrench]), 2),
    ("ProductTranslationGerman", NAMEOF('Products'[ProductTranslationGerman]), 3)
}
```

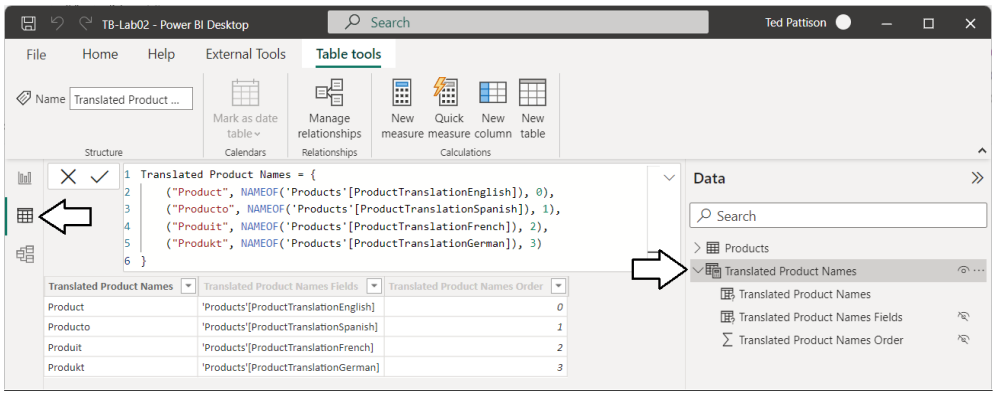
The way to resolve this issue is to update the DAX expression to replace the column names with localized translations for the word **Product** as shown in the following code listing.

```
Translated Product Names = {
    ("Product", NAMEOF('Products'[ProductTranslationEnglish]), 0),
    ("Producto", NAMEOF('Products'[ProductTranslationSpanish]), 1),
    ("Produit", NAMEOF('Products'[ProductTranslationFrench]), 2),
    ("Produkt", NAMEOF('Products'[ProductTranslationGerman]), 3)
}
```

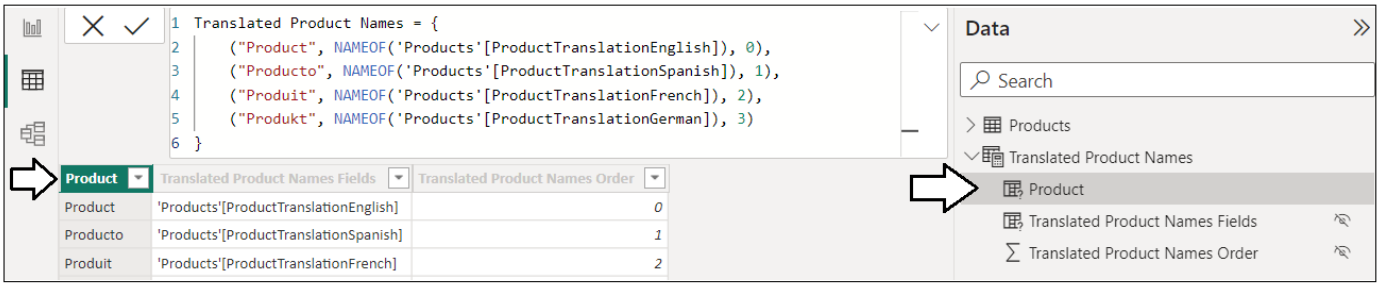
Once you make this change, you will see that the column header is now translated properly along with product names.



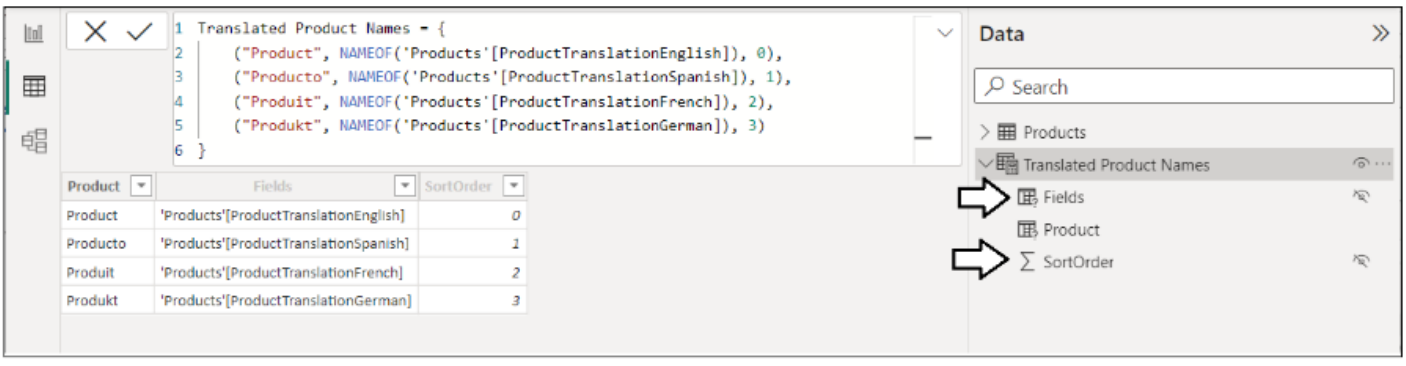
Up to this point we have only examined the Field Parameter in **Report** view. Now it's time to navigate to **Data** view where you can see two addition fields inside the Field Parameter that are hidden from **Report** view. If you expand the node for a Field Parameter such as **Translated Product Names**, you will see there are two more hidden fields in addition to the field you can see in **Report** view.



Note that the names of the columns inside a Field Parameter are automatically generated based on the name you gave to the top-level Field Parameter. The columns inside a Field Parameter can (and should) be renamed to simplify the data model and to improve readability. You can just double-click on a field inside the Field Parameter node to rename it. For example, you can rename the one field which is visible in **Report** view to **Product**.



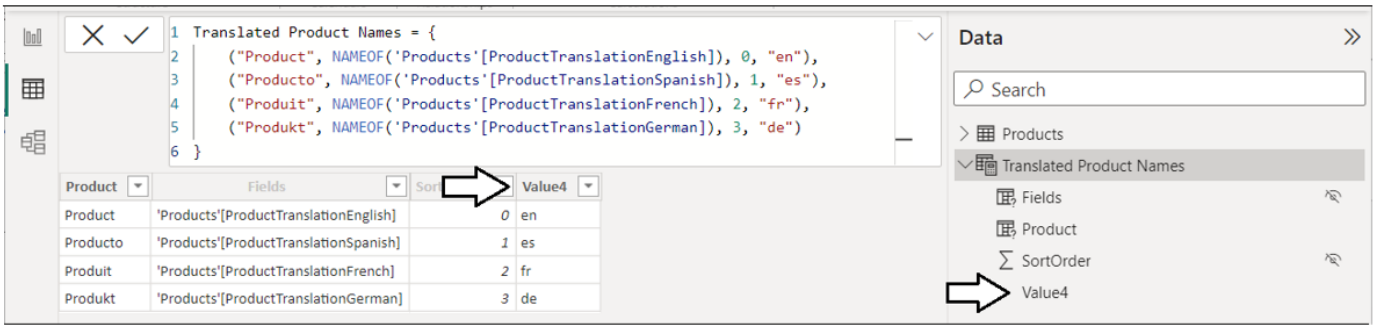
Likewise, you can rename the two other hidden fields with shorter names such as **Fields** and **SortOrder**.



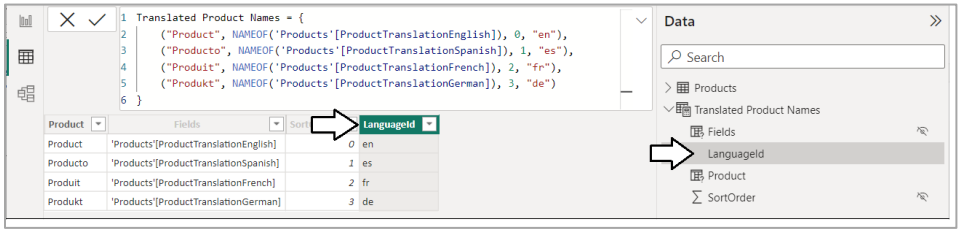
Now, here is where things get interesting. The Field Parameter that has been created is a table with three columns named **Product**, **Fields** and **SortOrder**. The next step in configuring a Field Parameter to support data translations is to add a fourth column with a language identifier to enable filtering by language. You can accomplish this by modifying the DAX expression for the Field Parameter by adding a fourth string parameter to the row for each language with the lower-case two character language identifier.

```
Translated Product Names = {
    ("Product", NAMEOF('Products'[ProductTranslationEnglish]), 0, "en" ),
    ("Producto", NAMEOF('Products'[ProductTranslationSpanish]), 1, "es" ),
    ("Produit", NAMEOF('Products'[ProductTranslationFrench]), 2, "fr" ),
    ("Produkt", NAMEOF('Products'[ProductTranslationGerman]), 3, "de" )
}
```

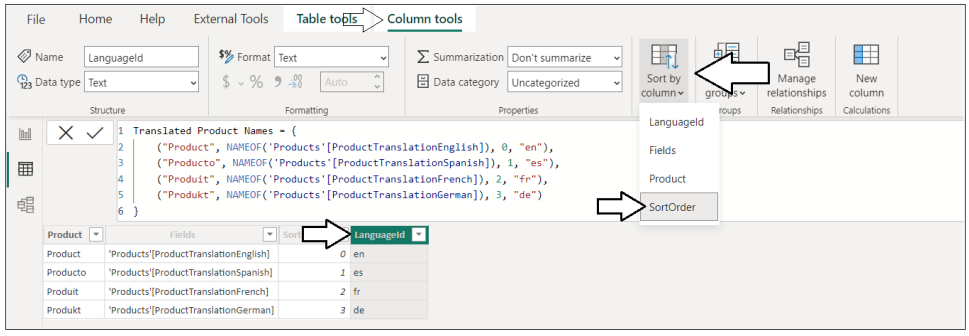
Once you have updated the DAX expression with a language identifier for each language, a new column will appear in the **Data** view of the **Products** table named **Value4**.



The name **Value4** isn't quite specific enough for our needs. Let's rename the forth column to **LanguageId**.



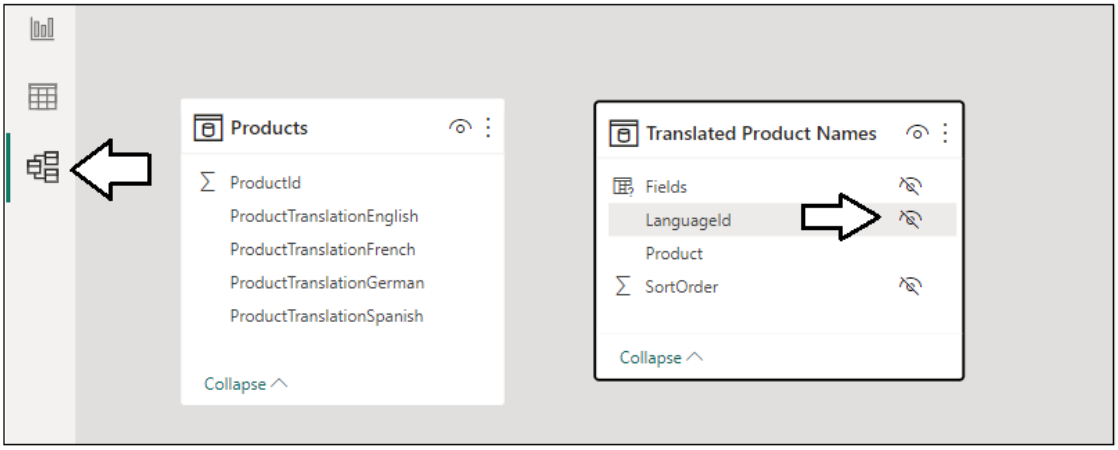
Finally, let's not forget to configure the sort column for the new column named **LanguageId**.



You do not have to worry about configuring the sort column for the two pre-existing fields named **Fields** and **Product**. That is done automatically by Power BI Desktop when you create a new Field Parameter. However, you need to explicitly configure the sort column when you add additional columns such as **LanguageId**.

The authors would like to thank Gerhard Brueckl for his great blog post where we first learned about this technique.

There is just one more thing to do with the new Field Parameter. Let's move to **Model** view to inspect the Field Parameter named **Translated Product Names**. As a final step, let's hide the **LanguageId** column from **Report** view. Report authors will never need to see this column as it will be used to select a language by filtering behind the scenes.

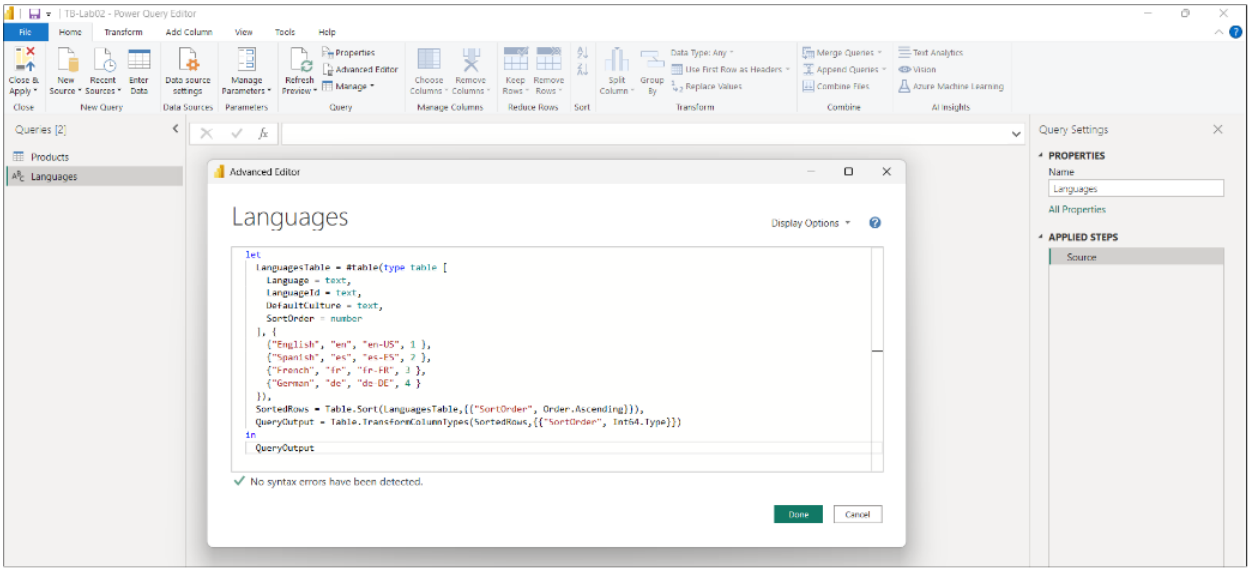


> At this point, we no longer need the slicer that can be automatically added by Power BI Desktop when creating the Field Parameter. While the slicer automatically added by Power BI Desktop is great for simple demos, it can only control a single Field Parameter at a time. You need a more scalable, report-wide strategy for switching back and forth between languages that works across multiple Field Parameters.

Let's summarize what we have done so far. We have created a Field Parameter named **Translated Product Names** and extended it with an extra column named **LanguageId**. The **LanguageId** column will be used to filter which source column is used, and therefore, which language will be displayed to report consumers. In the next section, we will continue building out the strategy for data translations by adding a new table named **Languages** which will be used to filter multiple Field Parameters at once in order to synchronize them as you switch between languages

Adding the Languages Table to Filter Field Parameters

As a content creator working with Power BI Desktop, there are many different ways to add a new table to a data model. For this scenario, let's use Power Query and the M query language to create a new table named **Languages**. In Power BI Desktop, you can create a Blank Query and name it **Languages**. After that, open the Advanced Editor window where you can type in M code or copy it from somewhere else and paste it in.



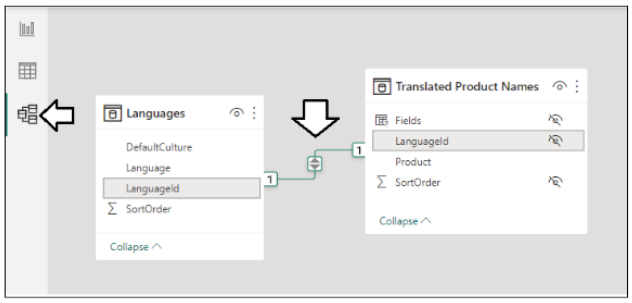
Examine the following M code for the query that is being used to generate the **Languages** table.

```
let
  LanguagesTable = #table(type table [
    Language = text,
    LanguageId = text,
    DefaultCulture = text,
    SortOrder = number
  ], {
    {"English", "en", "en-US", 1 },
    {"Spanish", "es", "es-ES", 2 },
    {"French", "fr", "fr-FR", 3 },
    {"German", "de", "de-DE", 4 }
  }),
  SortedRows = Table.Sort(LanguagesTable,{{"SortOrder", Order.Ascending}}),
  QueryOutput = Table.TransformColumnTypes(SortedRows,{{"SortOrder", Int64.Type}})
in
  QueryOutput
```

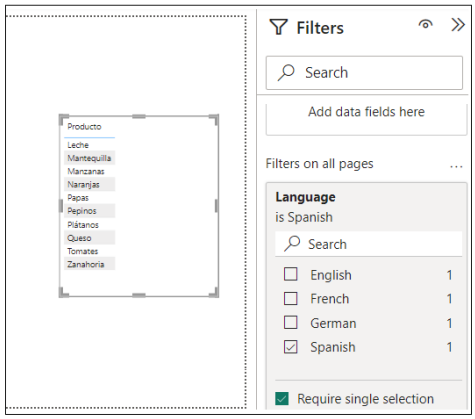
When this query executes, it generates the **Languages** table with a row for each of the four supported languages.

	Language	LanguageId	DefaultCulture	SortOrder
1	English	en	en-US	1
2	Spanish	es	es-ES	2
3	French	fr	fr-FR	3
4	German	de	de-DE	4

Once you have created the **Languages** table, you can move to **Model** view to set up the filtering by creating a relationship. More specifically, you can create a one-to-one relationship between the **Languages** table and the **Translated Product Names** Field Parameter using the **LanguageId** column as shown in the following screenshot.



Once you have established the relationship between **Languages** and **Translated Product Names**, you have created the foundation for filtering the Field Parameter on a report-wide basis. For example, you can open the **Filter** pane and add the **Language** column from the **Languages** table to the **Filters on all pages** section. If you configure this filter with the **Require single selection** option, you can then test out switching between languages using the **Filter** pane.



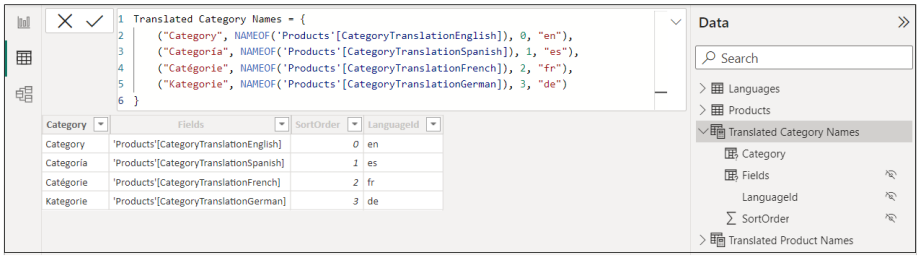
Synchronizing Multiple Field Parameters

At this point, we have added a Field Parameter to translate product names. However, most real-world reports will contain more than just one column that requires data translations. Therefore, you must ensure the mechanism you use to select a language can be synchronized across multiple field parameters. To test this out, let's create a second Field Parameter to translate product category names from the **Products** table.

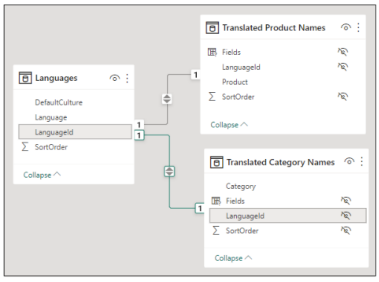
Let's assume the **Products** table contains four columns with translated category names similar to the translated product name columns. You can create a new Field Parameter named **Translated Category Names** using this DAX expression.

```
Translated Category Names = {
    ("Category", NAMEOF('Products'[CategoryTranslationEnglish]), 0, "en"),
    ("Categoría", NAMEOF('Products'[CategoryTranslationSpanish]), 1, "es"),
    ("Catégorie", NAMEOF('Products'[CategoryTranslationFrench]), 2, "fr"),
    ("Kategorie", NAMEOF('Products'[CategoryTranslationGerman]), 3, "de")
}
```

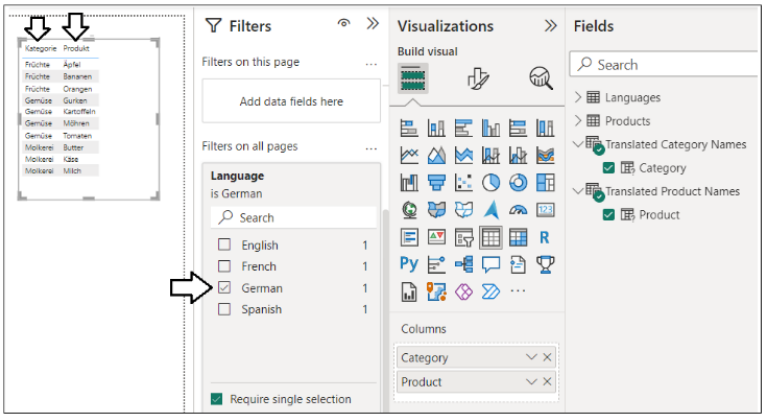
After creating the Field Parameter named **Translated Category Names**, let's update the field names and configure the sort column as we did earlier for the Field parameter named **Translated Product Names**.



The next step is to move to **Model** view where you can create a relationship based on the **LanguageId** column between **Languages** table and the **Translated Category Names** Field Parameter.



Now you should be able to add the **Category** column to the **Table** visual along with the **Product** column. As you change the **Language** selection in the **Filter** pane, the two Field Parameters are now synchronized to display the same language.

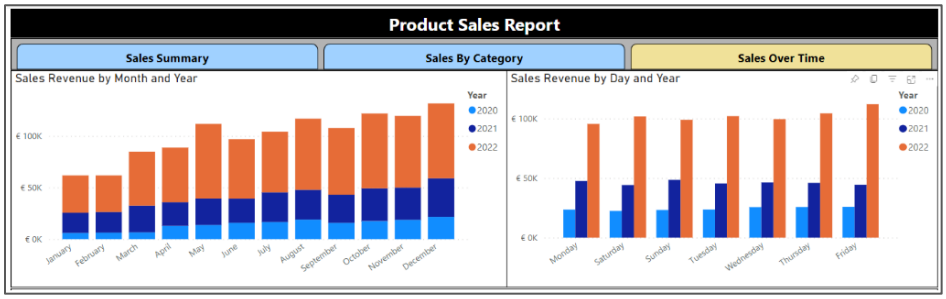


You have now learned how to synchronize the selection of language across multiple Field Parameters. The example you've just seen involves two Field Parameters. If your project involves a greater number of columns requiring data translations such as 10, 20 or even 50, you have now learned a repeatable approach that can scale as high as you need.

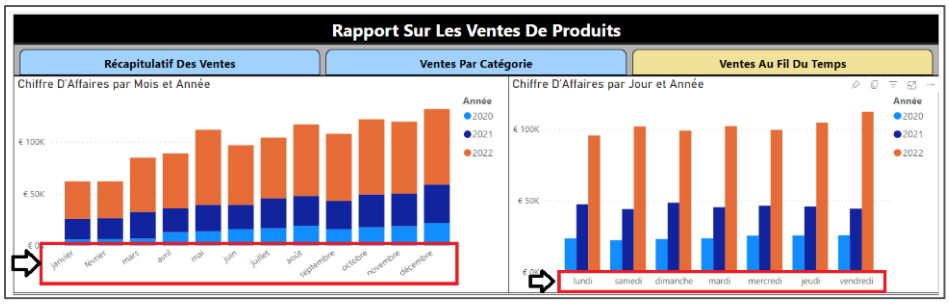
> One thing that can be confusing is trying to distinguish between the three different types of translations while testing. You can quickly test out your implementation of data translations in Power BI Desktop by changing the filter on the **Languages** table. However, the other two types of translations don't work correctly in Power BI Desktop. The metadata translations and report label translations you've added must always be tested in the Power BI Service.

Implementing Data Translations for a Calendar Table

If you are implementing data translations, you can make your users happy by adding translation support for text-based columns in a **Calendar** table such as the names of months and days of the week. For example, you might have added a custom Calendar table to your data model which makes it possible to visualize a breakdown of sales by the month or by the day.



To properly implement data translations for columns in a calendar table, you need a strategy to translate month names and day of the week names into the secondary languages you plan to support.



The strategy presented in this article for implementing calendar table column translations is based on Power Query and the power of the M query language. Power Query provides several built-in functions such as **Date.MonthName** which accept a **Date** parameter and return a text-based calendar name. If your PBIX project has **en-US** as it's default language and locale, the following Power Query function call will evaluate to a text-based value of **January**.

```
Date.MonthName( #date(2023, 1, 1) )
```

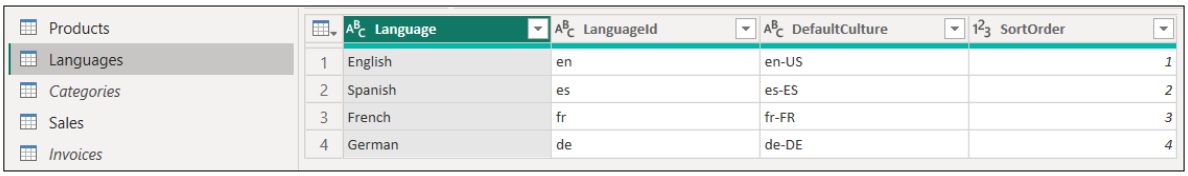
The **Date.MonthName** function accepts an second, optional string parameter to pass a specific language and locale.

```
Date.MonthName( #date(2023, 1, 1), "en-US")
```

If you want to translate the month name into French, you can pass a text value of **fr-FR**.

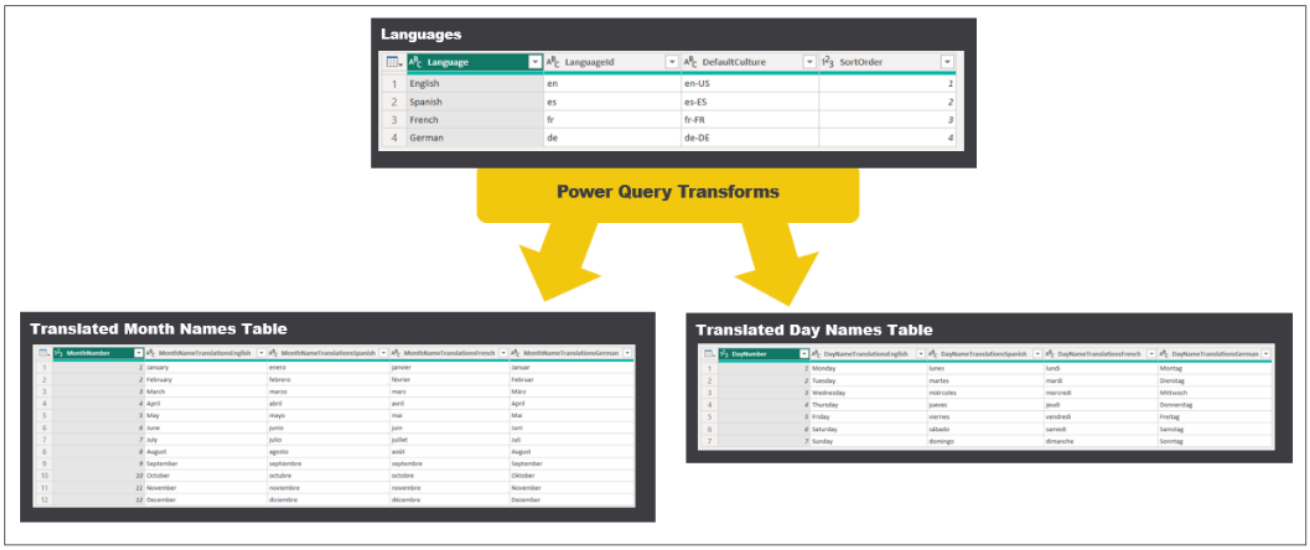
```
Date.MonthName( #date(2022, 12, 1), "fr-FR")
```

Now, let's revisit the **Languages** table you saw earlier. Now we can reveal why it includes the **DefaultCulture** column.



	Language	LanguageId	DefaultCulture	SortOrder
1	English	en	en-US	1
2	Spanish	es	es-ES	2
3	French	fr	fr-FR	3
4	German	de	de-DE	4

Power Query is built on a functional query language named M which makes it possible to enumerate through the rows of the **Languages** table to discover what languages and what default cultures are supported in the current project. This makes it possible to write a query which uses the **Languages** table as its source to generate a calendar translation table with the names of months or weekdays.

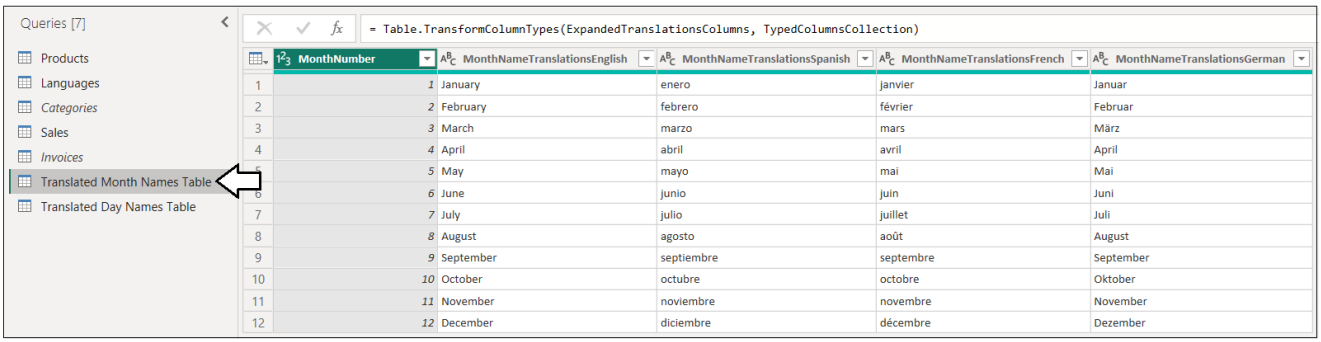


Here's an example of the M code used to generate the **Translated Month Names Table**.

```
let
    Source = #table( type table [ MonthNumber = Int64.Type ], List.Split({1..12},1)),
    Translations = Table.AddColumn( Source, "Translations",
        each
            [ MonthDate = #date( 2022, [ MonthNumber ], 1 ),
              Translations = List.Transform(Languages[DefaultCulture], each Date.MonthName( MonthDate, _ ) ),
              TranslationTable = Table.FromList( Translations, null ),
              TranslationsTranspose = Table.Transpose(TranslationTable),
              TranslationsColumns = Table.RenameColumns(
                TranslationsTranspose,
                List.Zip({ Table.ColumnNames( TranslationsTranspose ),
                           List.Transform(Languages[Language],
                           each "MonthNameTranslations" & _ ) })
              )
            ]
    ),
    ExpandedTranslations = Table.ExpandRecordColumn(Translations,
                                                    "Translations",
                                                    { "TranslationsColumns" },
                                                    { "TranslationsColumns" }),
    ColumnsCollection = List.Transform(Languages[Language], each "MonthNameTranslations" & _ ),
    ExpandedTranslationsColumns = Table.ExpandTableColumn(ExpandedTranslations,
                                                          "TranslationsColumns",
                                                          ColumnsCollection,
                                                          ColumnsCollection ),
    TypedColumnsCollection = List.Transform(ColumnsCollection, each {_, type text}),
    QueryOutput = Table.TransformColumnTypes(ExpandedTranslationsColumns, TypedColumnsCollection)
in
    QueryOutput
```

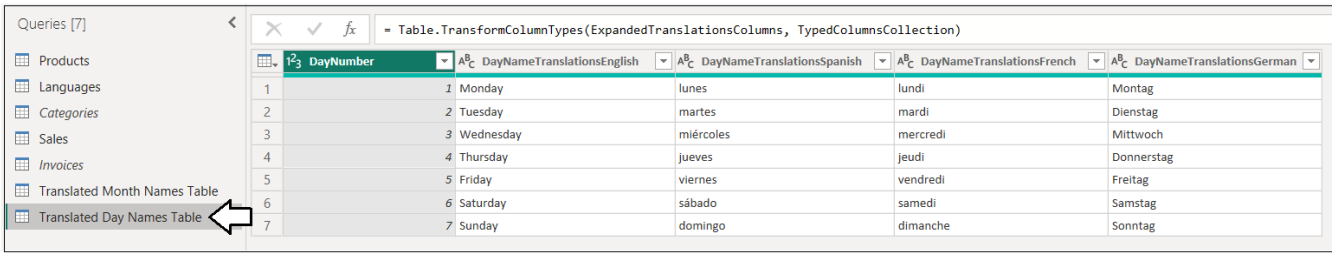
OK, so maybe this M code is a bit complicated. Don't worry. You will not be tested. You're not Chris Webb after all, so don't feel you need to be able to understand or explain this industrial-strength M code to others. You can simply copy and paste the M code from [ProductSalesMultiLanguage.pbix] whenever you need to add calendar translation tables to your project.

If the **Languages** table contains four rows for English, Spanish, French and German, the **Translated Month Names Table** query will generate a table with four translation columns as shown in the following screenshot.



	MonthNumber	MonthNameTranslationsEnglish	MonthNameTranslationsSpanish	MonthNameTranslationsFrench	MonthNameTranslationsGerman
1	1	January	enero	janvier	Januar
2	2	February	febrero	février	Februar
3	3	March	marzo	mars	März
4	4	April	abril	avril	April
5	5	May	mayo	mai	Mai
6	6	June	junio	juin	Juni
7	7	July	julio	juillet	Juli
8	8	August	agosto	août	August
9	9	September	septiembre	septembre	September
10	10	October	octubre	octobre	Oktober
11	11	November	noviembre	novembre	November
12	12	December	diciembre	décembre	Dezember

Likewise, the query named **Translated Day Names Table** will generate a table with week day name translations.

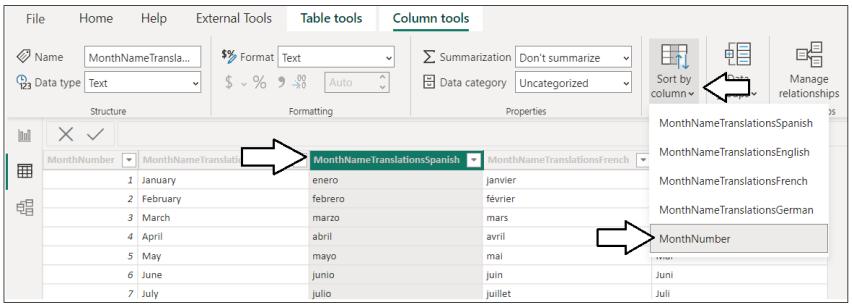


	DayNumber	DayNameTranslationsEnglish	DayNameTranslationsSpanish	DayNameTranslationsFrench	DayNameTranslationsGerman
1	1	Monday	lunes	lundi	Montag
2	2	Tuesday	martes	mardi	Dienstag
3	3	Wednesday	miércoles	mercredi	Mittwoch
4	4	Thursday	jueves	jeudi	Donnerstag
5	5	Friday	viernes	vendredi	Freitag
6	6	Saturday	sábado	samedi	Samstag
7	7	Sunday	domingo	dimanche	Sonntag

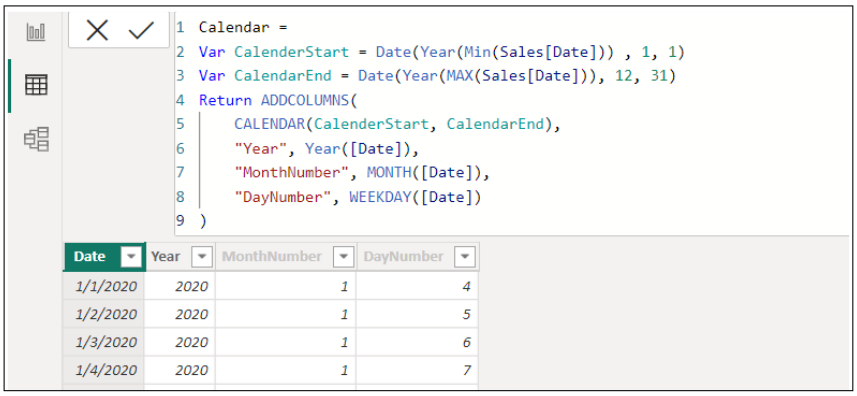
There is an important observation about the two queries named **Translated Month Names Table** and **Translated Day Names Table**. These queries have been written to be generic. In other words, they do not contains any hard-coded column names. This lowers ongoing maintenance because these queries do not require any modifications in the future when you add or remove languages from the project. All you need to do is to update the data rows in the query which generates the **Languages** table and the other two queries named **Translated Month Names Table** and **Translated Day Names Table** will automatically adapt to those changes.

Once again, always strive to use localize techniques that lower the overhead of adding new languages in the future.

When you execute these two queries for the first time, they will create two new tables in the dataset with the names **Translated Month Names Table** and **Translated Day Names Table** with a translation column for each language. One additional task you have is to configure the sort column for each of the translation columns. For example, all the translation columns in **Translated Month Names Table** should be configured to use the sort column **MonthNumber** while all the translations columns in **Translated Day Names Table** should be configured to use the sort column **DayNumber**.



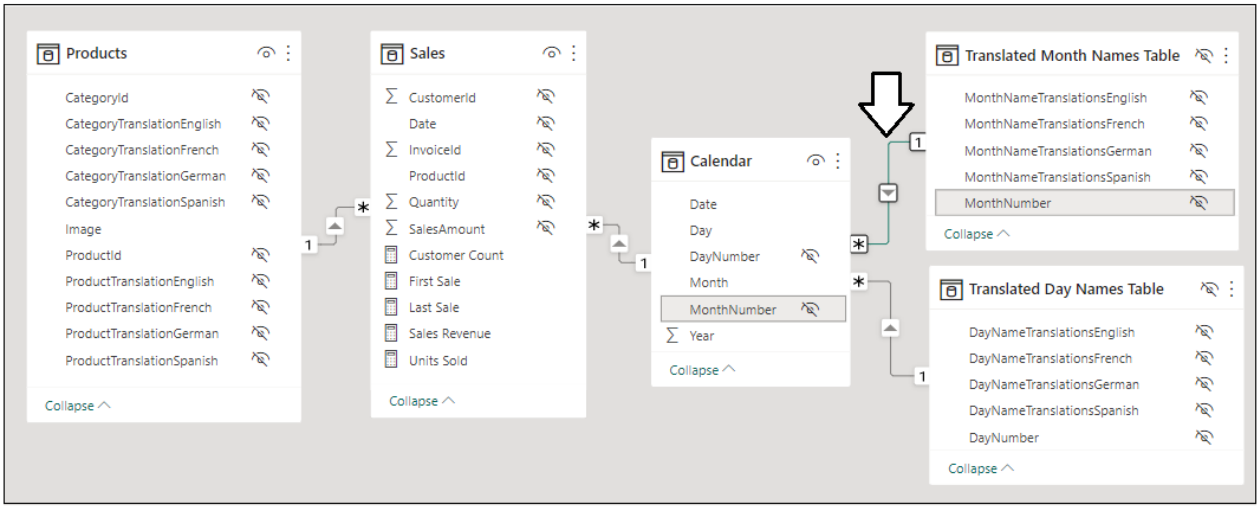
You've now seen how to generate the two translation tables named **Translated Month Names Table** and **Translated Day Names Table**. The next step is to integrate these two tables into the data model with a **Calendar** table. The **Calendar** table can be defined as a calculated table based on the following DAX expression.



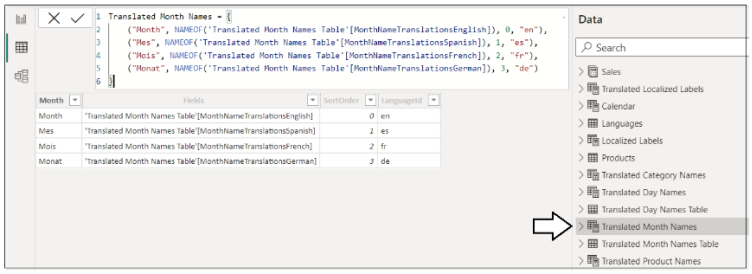
```
Calendar =
Var CalenderStart = Date(Year(Min(Sales[Date])) , 1, 1)
Var CalendarEnd = Date(Year(MAX(Sales[Date])), 12, 31)
Return ADDCOLUMNS(
    CALENDAR(CalenderStart, CalendarEnd),
    "Year", Year([Date]),
    "MonthNumber", MONTH([Date]),
    "DayNumber", WEEKDAY([Date])
)
```

Date	Year	MonthNumber	DayNumber
1/1/2020	2020	1	4
1/2/2020	2020	1	5
1/3/2020	2020	1	6
1/4/2020	2020	1	7

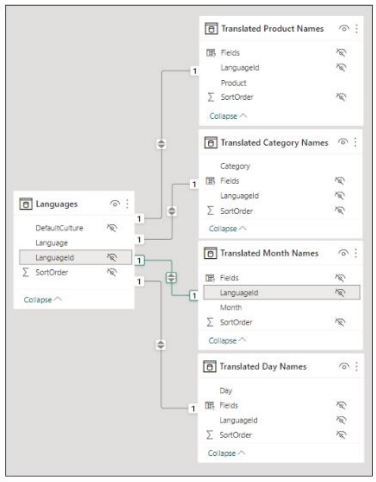
With a **Calendar** table like this, you will typically create a relationship between the **Calendar** table and other fact tables such as **Sales** using the **Date** column to create a one-to-many relationship. The relationships created between the **Calendar** table and the two translations tables are based on the **MonthNumber** column and the **DayNumber** column.



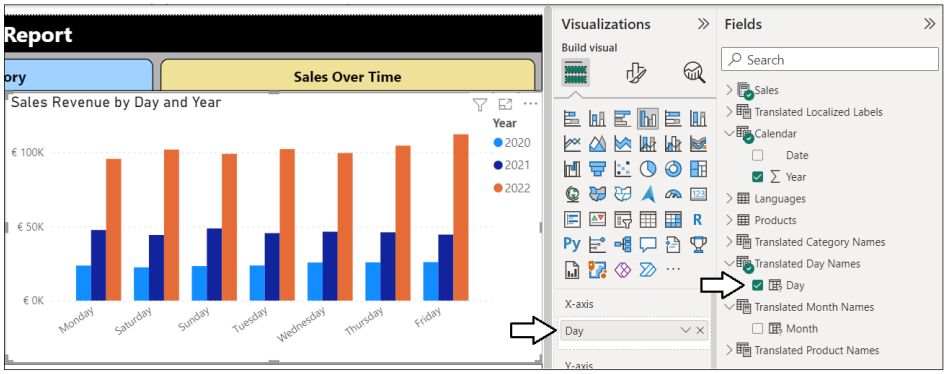
Once you have created the required relationships with the **Calendar** table, the next step is to create a new Field Parameter for each of the two calendar translations tables. Fortunately, creating a Field Parameter for a calendar translation table is just like creating the Field Parameters for product names and category names shown earlier.



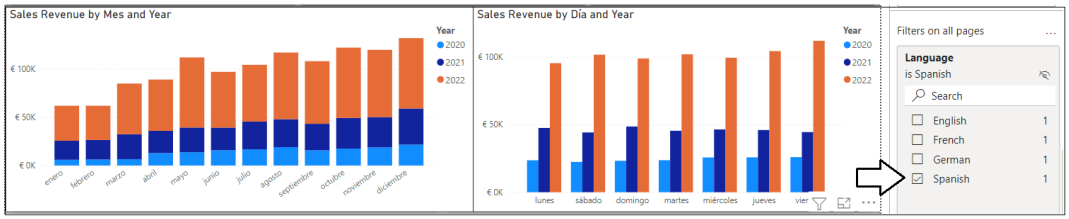
Don't forget that you need to add a relationship between these new Field Parameter tables and the **Languages** table to ensure the language filtering strategy works as expected.



Once you have created the Field Parameters for **Translated Month Names** and **Translated Day Names**, you can begin to surface them in a report using cartesian visuals, tables and matrices.

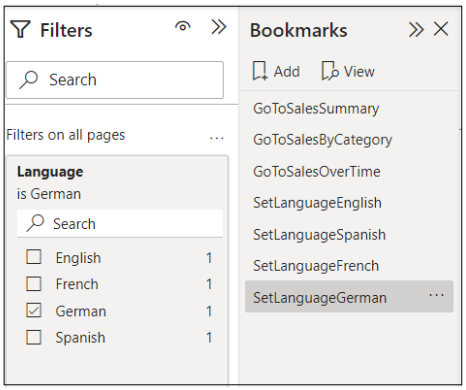


Once everything is set up correctly, you should be able test your work using a report-level filter on the **Languages** table to switch between languages and to verify translations for names of months and days of the week work as expected.

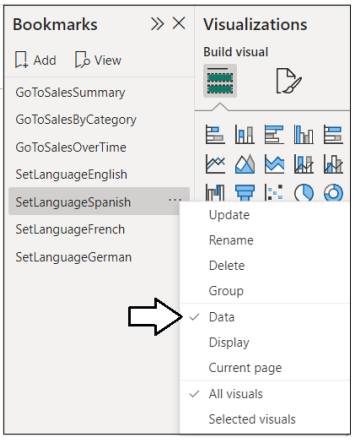


Loading Reports using Bookmarks to Select a Language

If you plan to publish a Power BI report with data translations for access by users through the Power BI Service, you must devise a way to load the report with the correct language filtering for the current user. This can be accomplished by creating a set of bookmarks which apply filters to the **Languages** table. When using this approach, you start by creating a separate bookmark for each language that supports data translations.



When creating bookmarks to filter tables, you should disable **Display** and **Current Page** and only enable **Data** behavior.



Earlier in this article, you learned that it is possible to load a report in the Power BI Service using the **language** parameter to force the Power BI Service to load the metadata translations for a specific language. Now that the report implements data translations in addition to the two other types, it is now necessary to pass a second parameter in the report URL to apply the bookmark for a specific language. This report URL parameter is named **bookmarkGuid** and this parameter makes it possible to specify the ID for a bookmark that gets applied as the report is loading.

```
/?language=es&bookmarkGuid=Bookmark856920573e02a8ab1c2a
```

The filtering on the **Languages** table is applied before any data is displayed to the user.

Embedding Reports That Implement Data Translations

Loading reports with Power BI embedding provides more flexibility than the report loading process for users accessing the report through the Power BI Service. Earlier you saw it is possible to load a report using a specific language and locale by extending the **config** object passed to **powerbi.embed** with a **localeSettings** object containing a **language** property as shown in the following code.

```
let config = {
  type: "report",
  id: reportId,
  embedUrl: embedUrl,
  accessToken: embedToken,
  tokenType: models.TokenType.Embed,
  localeSettings: { language: "de-DE" }
};

let report = powerbi.embed(reportContainer, config);
```

When you embed a report with a **config** object like this which sets the **language** property of the **localeSettings** object, the metadata translations and report label translations will work as expected. However, there is one additional step required to filter the **Languages** table to select the appropriate language for the current user.

While it's possible to apply a bookmark to an embedded a report, the use of bookmarks is not required as it is when loading reports for users in the Power BI Service. Instead, you can just apply a filter directly on the **Languages** table as the report loads using the Power BI JavaScript API. There is no need to add bookmarks for filtering the **Languages** table if you only intend to use a report using Power BI embedding.

The recommended way to apply a filtering during the loading process of an embedded report is to register an event handler for the **loaded** event. When you register an event handler for an embedded report's **loaded** event, you are able to provide a JavaScript event handler that executes before the rendering process begins. This makes the **loaded** event the ideal place to register an event handler whose purpose is to apply the correct filtering on the **Languages** table. Here is an example of JavaScript code which registers an event handler for the **loaded** event to apply a filter to the **Languages** table for Spanish.

```
let report = powerbi.embed(reportContainer, config);

report.on("loaded", async (event: any) => {

  let languageToLoad = "es";

  // create filter object
  const filters = [{
    $schema: "http://powerbi.com/product/schema#basic",
    target: {
      table: "Languages",
      column: "LanguageId"
    },
    operator: "In",
    values: [ languageToLoad ],
    filterType: models.FilterType.Basic,
    requireSingleSelection: true
  }];

  // pass filter object in a call to updateFilters
  await report.updateFilters(models.FiltersOperations.Replace, filters);

});
```

When setting filters with the Power BI JavaScript API, you should always prefer the **updateFilters** method over the **setFilters** method. That's because **updateFilters** allows you to remove existing filters while **setFilters** does not.

Summary

As you read through this article, you learned about the end-to-end process for building Power BI multilanguage reports. You should now have a solid conceptual understanding of how Power BI translations work and you should be able to distinguish between the three essential types of translations which include metadata translations, report label translations and data translations.

You've also learned that testing your work to localize Power BI reports can be challenging because metadata translations do not load correctly in Power BI Desktop. You must always publish your reports to a Premium workspace in the Power BI Service to test your work and to verify your translations are loading correctly.

This article introduced you to the external tool named Translations Builder. You've seen how Translations Builder can significantly reduce the manual effort required to add translations and localization support to PBIX project files. You have also seen how Translations Builder provides features for generating machine translation which can really accelerate the process for building and testing multilanguage reports.

This article examined how to export and import translation sheets which can enable a human workflow process and collaboration with an external team of translators. You learned how exporting a master translation sheet makes it possible to copy translations for secondary languages between PBIX projects. You also saw the potential to create an enterprise-level translation sheet which can be used to add a set of reusable report labels to each new PBIX project.

The last section of this article examined when and how to implement data translations. If you are building a PBIX project that requires data translations, you can leverage the strategy which uses Field Parameters as the foundation for a design to switch languages through filtering the **Languages** table. All in all, you now have the knowledge and technical skills required to build multi-language reports for Power BI using a strategy that is reliable, predictable and scalable.