# Building Multi-language Reports in Power BI

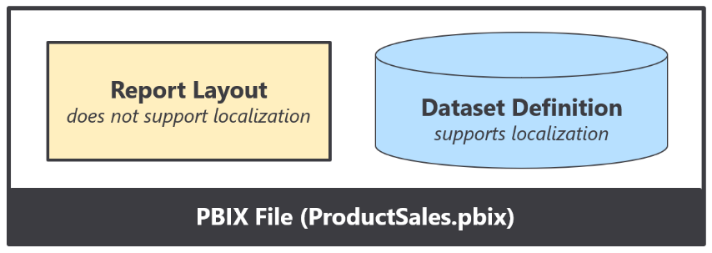
Power BI provides Internationalization and localization features which make it possible to build multi-language reports. For example, you can design a Power BI report that renders in English for some users while rendering in Spanish, French, German or Dutch for other users. If a company or organization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

The purpose of this article is to explain the key concepts involved with the Power BI features for Internationalization and localization and to provide guidance for building reports that support multiple languages.

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

A metadata translation represents the property for 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 **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**.

Keep in mind that the Power BI support for metadata translations only applies to dataset objects. Power BI does not support adding translations for text values stored as part of the report layout.



Think about a common scenario where you add a textbox or a button to a Power BI report and then you type in literal text for a string value displayed to the user. That text value is stored in the report layout and cannot be localized. Therefore, you must avoid using textboxes and buttons that include literal text when designing multi-language reports. As a second example, page tabs in a Power BI report are also problematic because their display names cannot be localized. Therefore, you must design multi-language reports so that page tabs are never displayed to the user.

You should understand that Power BI reports and datasets that support multiple languages must run in a dedicated capacity. That means you must be working with either Power BI Premium or the Power BI Embedded service in Microsoft Azure. Multi-language reports will not work correctly when loaded into a Power BI workspace in the shared capacity. If you are working in a Power BI workspace that does not display the diamond indicating a Premium workspace, you will find that multi-language reports do not work as expected because they cannot load secondary translations.

### Understand 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.

* Metadata Translations
* Report Label Translations
* Data Translations

**Metadata translations** provides localized values for dataset objects. The object types which support metadata translation include tables, columns, measures, hierarchies and hierarchy levels. Metadata translations are the easiest to create, managed and integrate into a Power BI report experience. By using the machine translation features of Translations Builder, you can add all the metadata translations you need to test a Power BI report in a matter of minutes. While adding metadata translations to your dataset is an essential first step, it doesn't provide a complete solution by itself. A complete solution will require going further to localize report labels.

**Report label translations** provide localized values for text-based elements on a report that are not directly associated with a dataset object. Examples of report labels include the report title, section heading and button captions. Report label translations are harder to create and manage than metadata translations because Power BI provides no direct feature to integrate them. Translations Builder solves this problem by creating a hidden table in the data model named Localized Labels and it adds a hidden measure to track the translations for each report label that requires localization.

**Data translations** provide translated values for text-based columns from the rows of data shown in a report. Data translations are harder to design and implement as the underlying datasource must be designed with additional columns for each language that needs to be supported. Once the underlying datasource has been extended with extra translation columns, you can use the new feature in Power BI Desktop named Field Parameters to design a scheme that allows a report to provide a mechanism where filtering can be used to load the data translations for a specific language.

While every multi-language report will require metadata translations, it's is not as clear whether they will also require data translations. Some projects will require data translations while other projects will not.

### The Multi-language Report Development Process

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

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



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

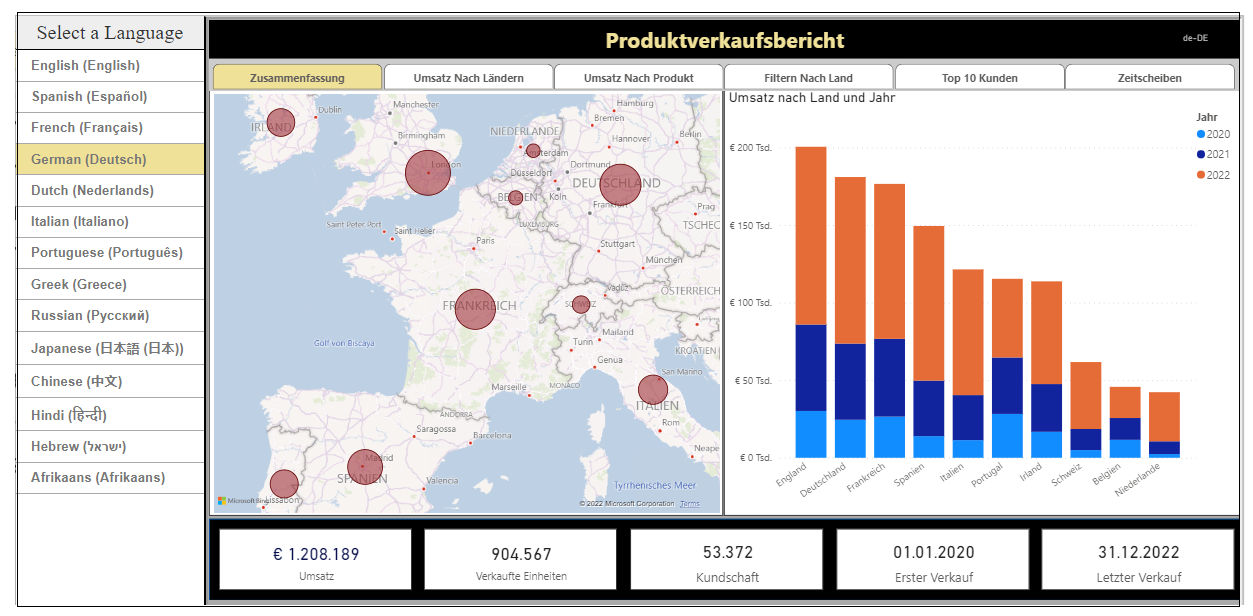
To use the shared dataset approach, you create one PBIX project file with a dataset definition and an empty report which remains unused. Once this dataset has been deployed to the Power BI Service, report builders can connect to it using Power BI Desktop to create report-only PBIX files. This makes it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



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

### Translations Builder Live Demo

This article is accompanied a [**live demo**](https://multilanguagereportdemo.azurewebsites.net/) based on a single PBIX file solution named[**TranslationsBuilderLiveDemo.pbix**](https://github.com/PowerBiDevCamp/TranslationsBuilder/raw/main/LiveDemo/TranslationsBuilderLiveDemo.pbix). This live demo shows the potential of building multi-language reports for Power BI. The report in the lice demo supports English, Spanish, French, German, Dutch.



xx

<https://multilanguagereportdemo.azurewebsites.net>

While the solution provided by [**TranslationsBuilderLiveDemo.pbix**](https://github.com/PowerBiDevCamp/TranslationsBuilder/raw/main/LiveDemo/TranslationsBuilderLiveDemo.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.

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

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

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

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

The **live demo** demonstrates how to implement localized labels in the report title and the top navigation menu buttons. The following screenshot shows the how button captions are translated when loaded with five different languages.

The live demo also demonstrates how to implement data translations. With metadata translations, you can see the names of columns and measures change as you switch between languages. Data translations go further to localize the product names in rows of the **Products** table. The following screenshot shows how the **ProductSales.pbix** developer sample provides data translations for product names as well.

## Prepare Datasets and Reports for Localization

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

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

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

Adding a healthy degree of padding to localized labels is the norm when developing 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.

### Avoid Report Design Techniques that Do Not Support Localization

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

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

The key point here is that any literal text that gets added to the report layout cannot be localized. Consider the case where you add a column chart to your report. By default, a Cartesian visual such as a column chart is assigned a dynamic 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 language supported by the **ProductSales.pbix** developer sample.

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

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

### Understanding the Localized Labels Table

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

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

SHOW MENU to create localized labels table.

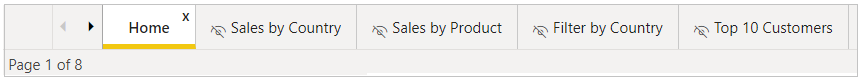
The **TranslationsBuilderLiveDemo.pbix** sample demonstrates a hidden **Localized Labels** table to provide a set of localized labels for all titles, headings and button captions used throughout the report.

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

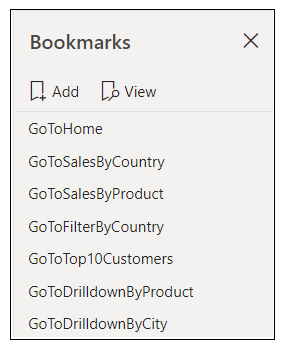
### Add Support for Page Navigation

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

The first thing you need to do is to hide every page in the report except for the first page which acts as the landing page.

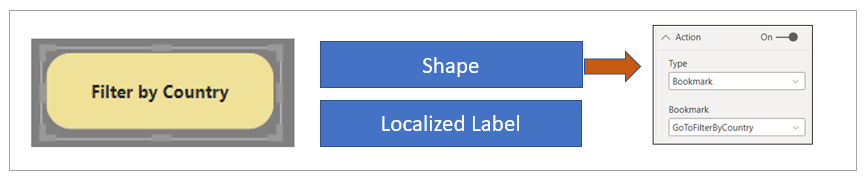


Next, create a set of bookmarks. Each bookmark should be created to navigate to a specific page. The **ProductSales.pbix** developer sample demonstrates this technique by adding a bookmark for each page supported by the navigation menu.



Remember, that you cannot add a button with literal text to a multi-language report. Instead, you configure the button to pull its text value from a measure in the Translated Localized Labels table.

From a design perspective, the user sees the localized label with the button caption but does not see the shape. When the user clicks on the localized label visual, the invisible shape on top acts as a button and responds to the user’s click action by applying a bookmark to navigate to the target page.



At this point, you've learned how to create the **Localized Labels** table and add labels. 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 must support multiple languages

## Understanding How Translations Builder Works

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

ProductSales-Translations-Spanish.csv

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

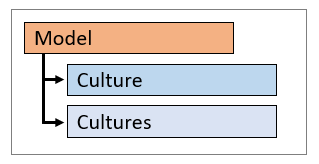
e4287640-d84c-4cb9-abce-443568c5be64-Translations-Spanish.csv.

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

### Add Secondary Cultures to a Dataset

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

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



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

string defaultCultureName = model.Culture;

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

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

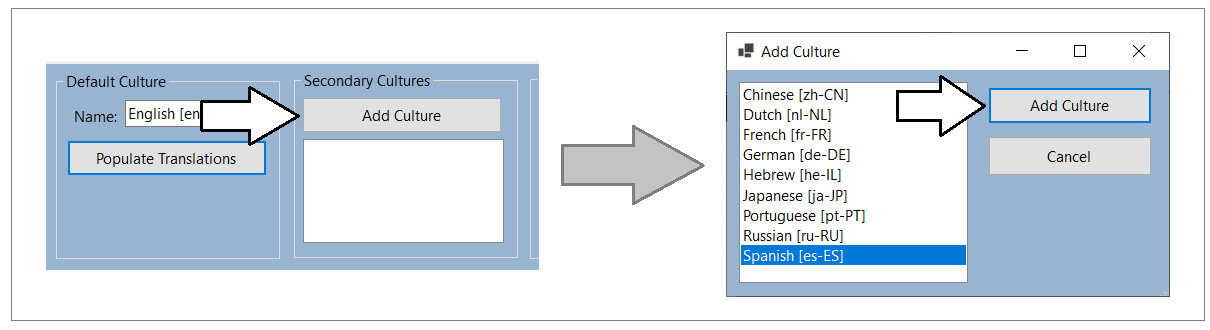
In a new dataset definition, the **Cultures** collection contain a single **Culture** object based on the default culture. You must explicitly add new **Culture** objects to the **Cultures** collection to support secondary languages. Here is an example of adding a secondary culture to support adding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

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

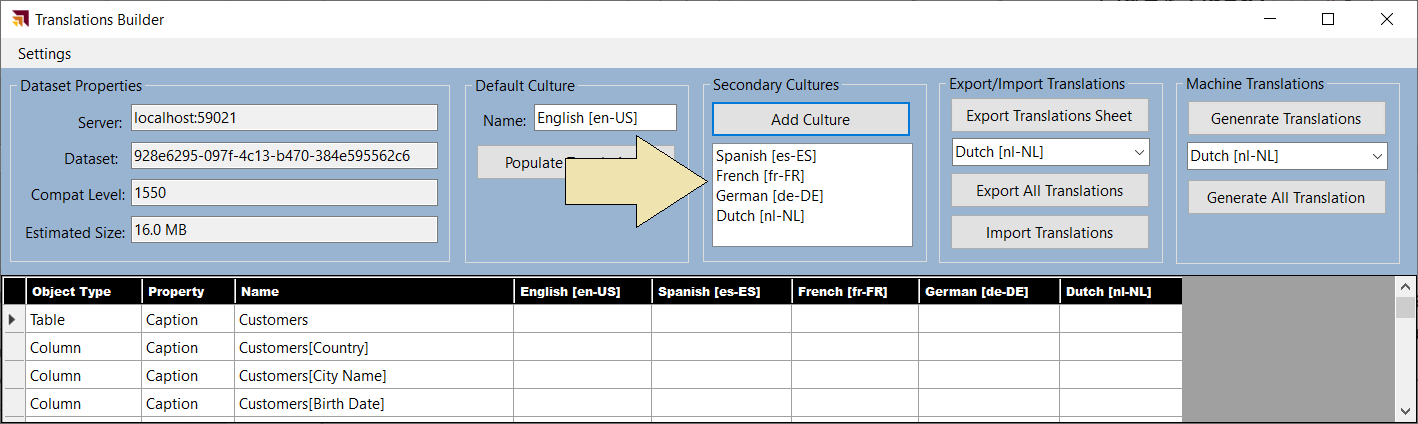
model.SaveChanges();

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

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



After adding new cultures to a dataset definition, you should be able to see them in the list box in the **Secondary Cultures** section. You should also notice that a new column appears for each culture in the table grid shown below.



**TranslationsBuilder** uses the following code to discover which secondary cultures exist within the current dataset.

public static List<string> GetSecondaryCulturesInDataModel() {

List<string> secondaryCultures = new List<string>();

// enumerate through Cultures collection

foreach (var culture in model.Cultures) {

// include every culture except for the default culture

if (!culture.Name.Equals(model.Culture)) {

secondaryCultures.Add(culture.Name);

}

}

return secondaryCultures;

}

### Add Metadata Translations to a Dataset Object

A key point to understand is that every metadata translation must be added within the scope of a **Culture** object that already exists in the current dataset definition. Each **Culture** object provides a collection property named **ObjectTranslations** which exposes a **SetTranslation** method which is used to add new metadata translations.

When calling the **SetTranslation** method, you must pass three parameters including the target dataset object, the target property and the value of the metadata translation. Let's start with the easiest case when adding a metadata translation to the default culture for a dataset object such as a table.

// acquire reference to datase object

Table tableCustomers = model.Tables["Customers"];

// get default Culture object

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

// call SetTranslation to add translation for Caption property of the dataset object

culture.ObjectTranslations.SetTranslation(tableCustomers, TranslatedProperty.Caption, tableCustomers.Name);  
  
// save changes back to model

model.SaveChanges();

In the call to **SetTranslation** shown above, you can see that the first parameter is passed as a **Table** object. It is also possible to pass other types of dataset objects in the first parameter including **Column**, **Measure**, **Hierarchy** or **Level**. The second parameter indicates the target property type which can be passed as an enumeration value of **Caption**, **Description** or **DisplayFolder**. The third parameter passes the value of the translation being created.

The **TranslationsManager** class provides the [**PopulateDefaultCultureTranslations**](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs#L291) method which automates the process of adding metadatatranslations for all non-hidden tables, columns, measures, hierarchies and levels in the current dataset definition. The following listing shows a simplified version of the **PopulateDefaultCultureTranslations** method.

public static void PopulateDefaultCultureTranslations() {

// (1) get default Culture object

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

foreach (Table table in model.Tables) {

if (!table.IsHidden) {

// (2) set translation for table

culture.ObjectTranslations.SetTranslation(table, TranslatedProperty.Caption, table.Name);

// (3) set translations for all non-hidden columns

foreach (Column column in table.Columns) {

if (!column.IsHidden) {

culture.ObjectTranslations.SetTranslation(column, TranslatedProperty.Caption, column.Name);

}

};

// (4) set translations for all non-hidden measures

foreach (Measure measure in table.Measures) {

if (!measure.IsHidden) {

culture.ObjectTranslations.SetTranslation(measure, TranslatedProperty.Caption, measure.Name);

}

};

// (5) set translations for all non-hidden hierarchies

foreach (Hierarchy hierarchy in table.Hierarchies) {

if (!hierarchy.IsHidden) {

culture.ObjectTranslations.SetTranslation(hierarchy, TranslatedProperty.Caption, hierarchy.Name);

// (6) set translations for all hierarchy levels

foreach (var level in hierarchy.Levels) {

culture.ObjectTranslations.SetTranslation(level, TranslatedProperty.Caption, level.Name);

}

}

};

}

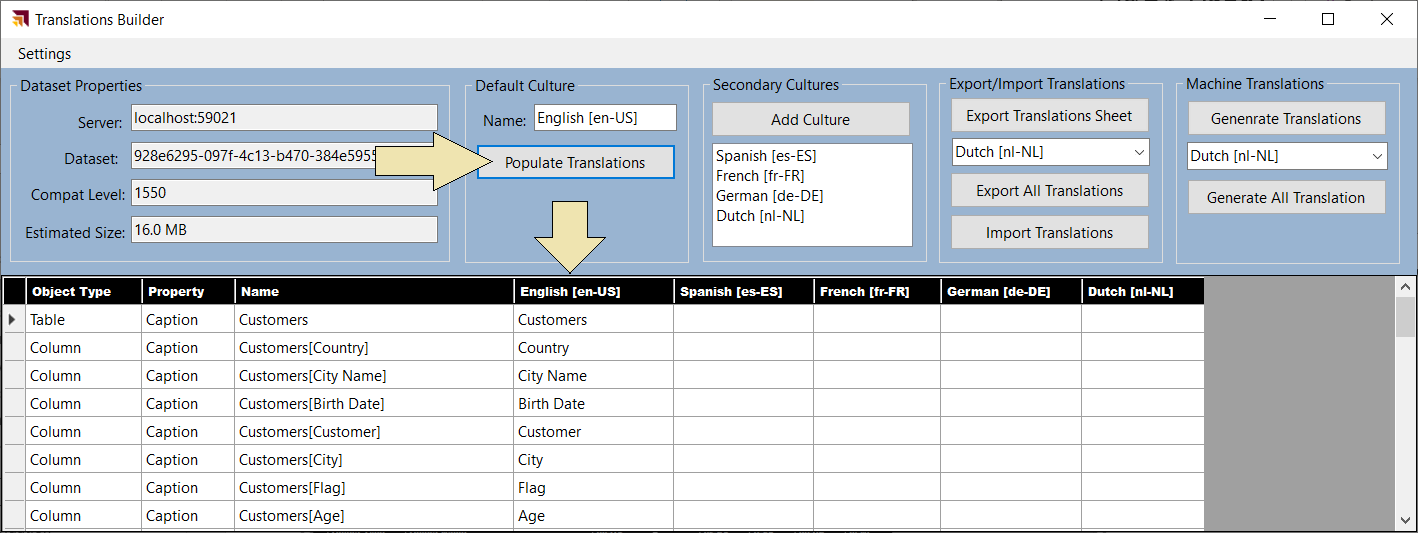
model.SaveChanges();

}

}

The actual implementation of **PopulateDefaultCultureTranslations** includes extra code to add metadata translations to dataset object using the **Description** property and the **DisplayFolder** property in addition to the **Caption** property.

The **TranslationsBuilder** application provides a **Default Culture** section with a **Populate Translations** button. When you click this button, it executes **PopulateDefaultCultureTranslations**. After clicking **Populate Translations**, you should be able to view the metadata translations that have been added into the default culture as shown in the table grid below.



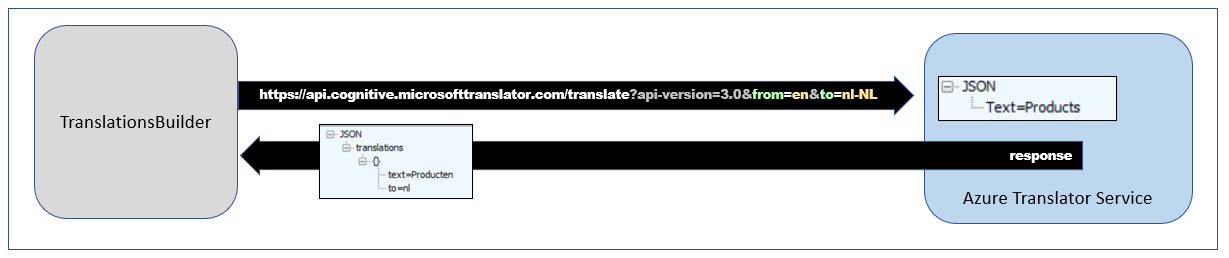
### Generate Machine Translations using Azure Translator Service

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 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 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 **TranslationsBuilder**. If you initially generate machine translations for each secondary culture 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.

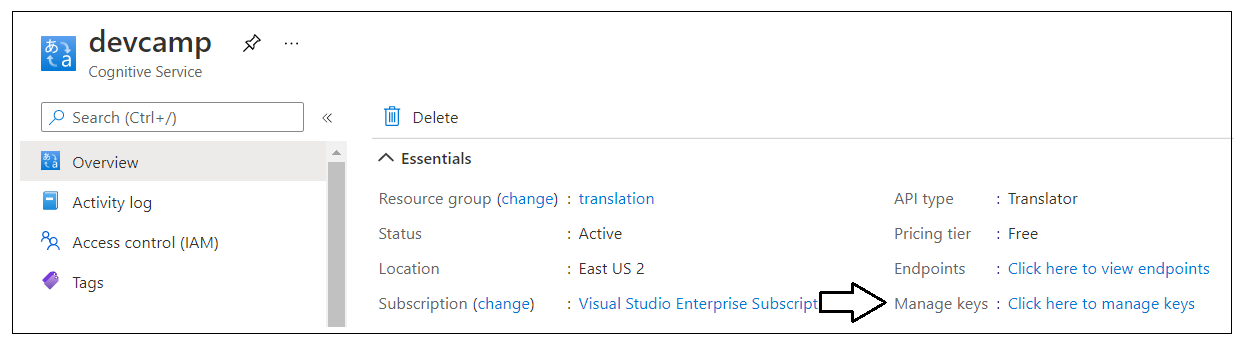
While machine translation are not always guaranteed to be high quality, they do provide value in the multi-language report development process. First, they can act as translation placeholders so you can begin your testing by loading reports using secondary languages to see if there are layout issues 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.

The **TranslationsBuilder** application includes support to generate machine translations with the [Azure Translator service](https://docs.microsoft.com/en-us/azure/cognitive-services/translator/translator-info-overview) that is part of 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 secondary languages. The diagram below shows the details of executing a Translator service API request to convert a table name from English (en) to Dutch (nl).

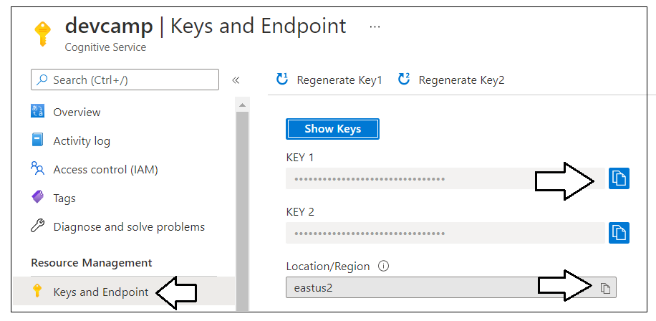


If you'd like to test out the support in **TranslationsBuilder** for generating machine translations, you will require an Azure subscription. An Azure subscription is required because you must create your own instance of the Translator service in Microsoft Azure. Once you have created an instance of the Translator service, you must determine its location and acquire an application key before you can successfully call to it from the **TranslationsBuilder** application.

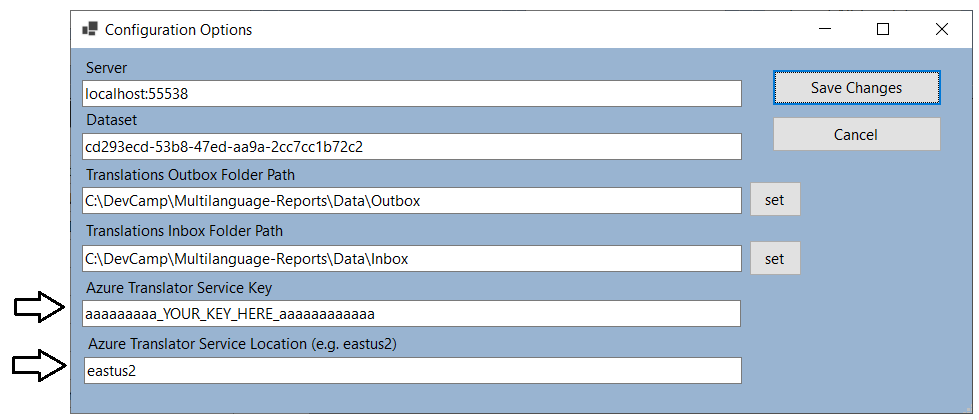
The Azure Translator service does not support anonymous API calls. Instead, you must acquire an application keys which acts as a security credential because it allows you to execute secure Web API calls to the Translator service. When working with an instance of the Translator service in the Azure portal, you can navigate to the **Keys and Endpoint** page where you can view and acquire an application key.



On the **Keys and Endpoint** page, you can copy the application key and then paste it into the **Configuration Options** dialog in **TranslationsBuilder**. You will also need to copy the service **Location** and paste that value into the **Configuration Options** dialog as well.



Once you have copy-and-pasted the application key and the location for your Translator service instance, you can save your changes in the **Configuration Options** dialog and begin generating machine translations.

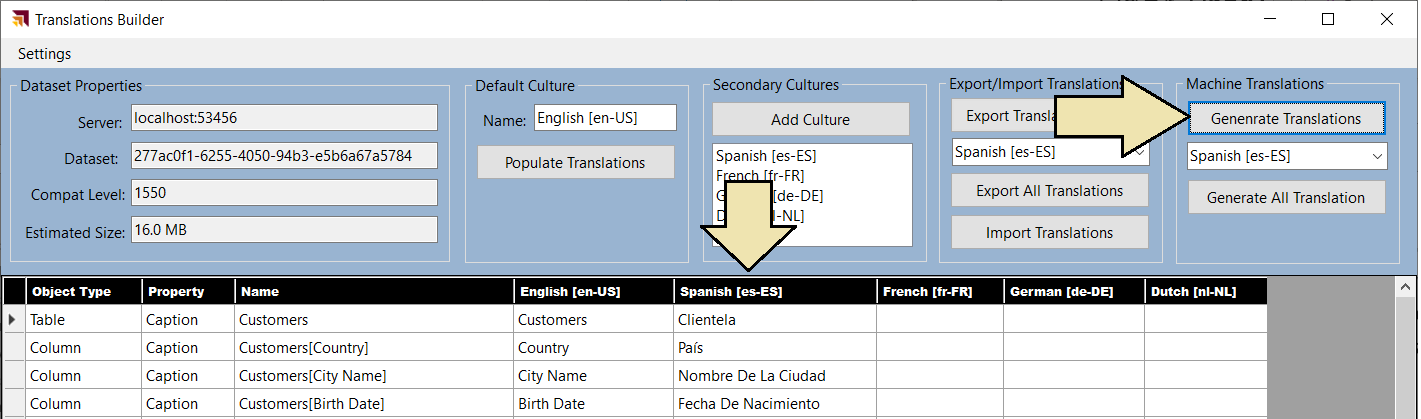


The **TranslationsBuilder** project contains a class named [**TranslatorService**](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslatorService.cs) which contains code to manage the low-level details of calling the Translator service API to translate text values from one language to another. The **TranslatorService** class provides the public static method named [**TranslateContent**](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslatorService.cs#L64) which can be called to translate text values.

The **TranslateContent** method accepts two parameters. The first parameter is a string value containing the text to be translated. The second parameter is used to pass the target language for translation. The implementation of **TranslateContent** abstracts calling the Translator server Web API and does the work to translate text value from the default culture language to the target language. Here is an example of calling **TranslateContent** to translate a text value from the default culture language (e.g. English) to the language of Spanish in the locale of Spain.

String translatedSpanishName = TranslatorService.TranslateContent("Customers", "es-ES");

The **TranslationsBuilder** application contains a **Machine Translations** section with a **Generate Translations** button. Note that the **Machine Translations** section will not be visible until you have added an application key and a location for the Azure Translation Service in the **Configuration Options** dialog. Once you have configured **TranslationsBuilder** with an application key and a location for your Translator service instance, you should be able to select a target culture with the language you want and click the **Generate Translations** button to generate machine translation for that language.



The **TranslationsManager** class contains a static method named [**PopulateCultureWithMachineTranslations**](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/TranslationsBuilder/Services/TranslationsManager.cs#L354) which takes a single string parameter for the target language. The implementation of this method adds new machine-generated translations to every non-hidden table, column, measure, hierarchy and level in the dataset. You can walk through the following listing to see a simplified version of this method which adds metadata translations to a secondary culture.

public static void PopulateCultureWithMachineTranslations(string CultureName) {

// (1) add secondary culture to model if it doesn't already exist

if (!model.Cultures.ContainsName(CultureName)) {

model.Cultures.Add(new Culture { Name = CultureName });

}

// (2) get target Culture object where metadata translations will be added

Culture culture = model.Cultures[CultureName];

// (3) enumerate through all non-hidden tables, columns, measures and hierachies

foreach (Table table in model.Tables) {

if (!table.IsHidden) {

// (4) get machine translation for table name and use it set translation for Caption property

String translatedName = TranslatorService.TranslateContent(table.Name, CultureName);

culture.ObjectTranslations.SetTranslation(table, TranslatedProperty.Caption, translatedName);

// (5) get machine translations for visible column names and use them to set translations

foreach (Column column in table.Columns) {

if (column.IsHidden) {

translatedName = TranslatorService.TranslateContent(column.Name, CultureName);

culture.ObjectTranslations.SetTranslation(column, TranslatedProperty.Caption, translatedName);

}

};

// (6) get machine translations for visible measure names and use them to set translations

foreach (Measure measure in table.Measures) {

if (!measure.IsHidden) {

translatedName = TranslatorService.TranslateContent(measure.Name, CultureName);

culture.ObjectTranslations.SetTranslation(measure, TranslatedProperty.Caption, translatedName);

}

};

// (7) get machine translations for visible hierachy names and use them to set translations

foreach (Hierarchy hierarchy in table.Hierarchies) {

if (!hierarchy.IsHidden) {

translatedName = TranslatorService.TranslateContent(hierarchy.Name, CultureName);

culture.ObjectTranslations.SetTranslation(hierarchy, TranslatedProperty.Caption, translatedName);

// (8) get machine translations for hierachy level names and use them to set translations

foreach(var level in hierarchy.Levels) {

var translatedLevelName = TranslateContent(level.Name, CultureName);

culture.ObjectTranslations.SetTranslation(level, TranslatedProperty.Caption, translatedLevelName);

}

}

};

}

}

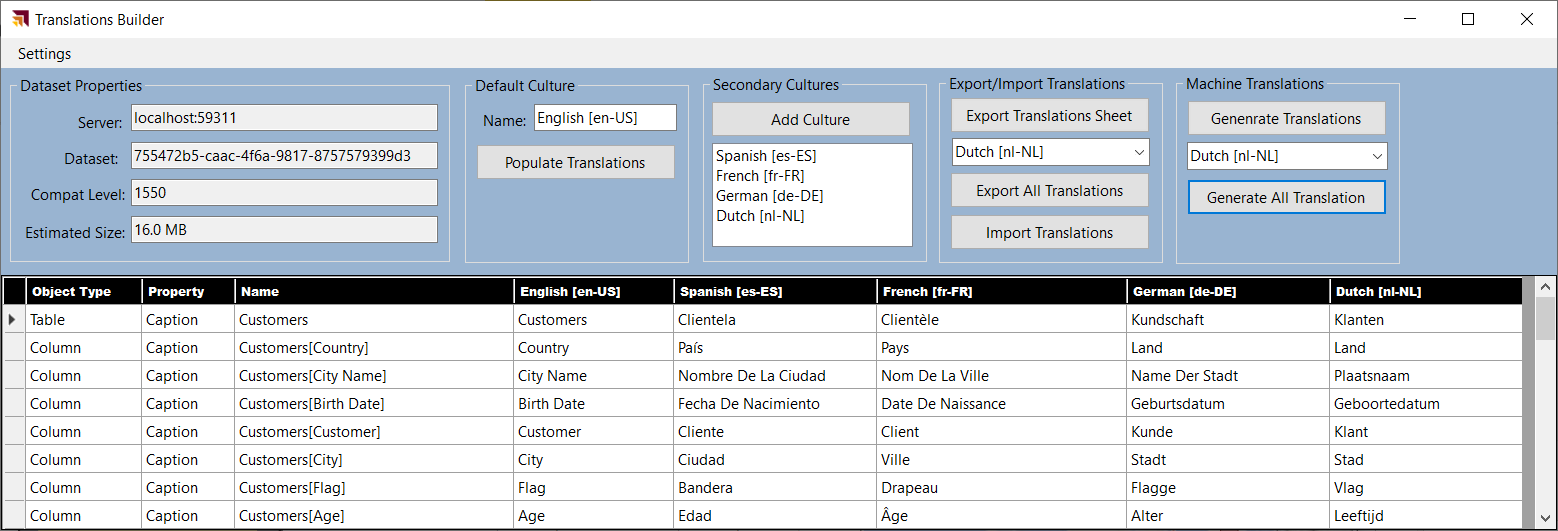
// (6) Save changes to the underlying dataset

model.SaveChanges();

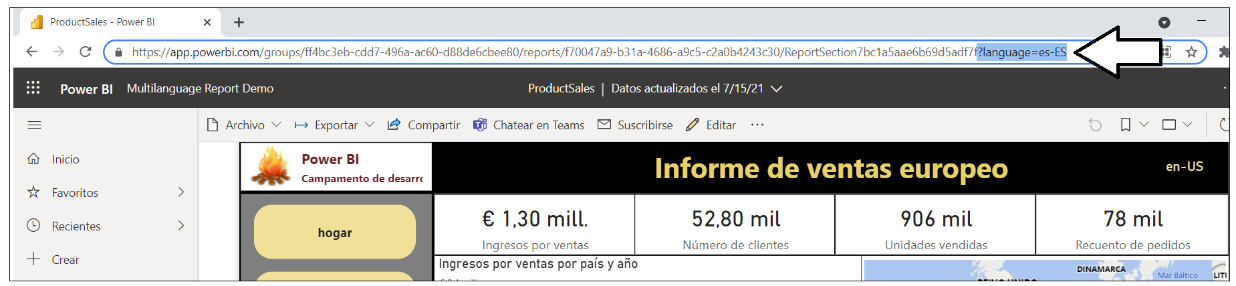
}

The actual implementation of **PopulateCultureWithMachineTranslations** contains more code than is shown in this code listing. More specifically, the actual implementation of **PopulateCultureWithMachineTranslations** includes extra code to ensure that metadata translation are added for any dataset object that has non-blank property values for either its **DisplayFolder** property or its **Description** property. Adding this extra support to localize the property values for the **DisplayFolder** property and **Description** property is important. This is especially true for scenarios will be using different languages to open report in edit mode.

You now seen how the **TranslationsBuilder** application can automate common localization tasks such as adding secondary cultures and generating machines translations. This will allow you to reach you first milestone of adding the first round of metadata translations so you can begin testing them by loading a localized report with different languages.



Once you have added metadata translations for a secondary culture, you can test your work by publishing the dataset and report to a workspace in the Power BI Service. Remember that the workspace must be associated with a Premium capacity or a Power BI Embedded capacity or metadata translations will not load as expected. To load the metadata translations for a specific language, you can add the **language** query string parameter at the end of the URL for a report. For example, you can load a report with Spanish translations by adding a **language** query string parameter of **es** or **es-ES**.



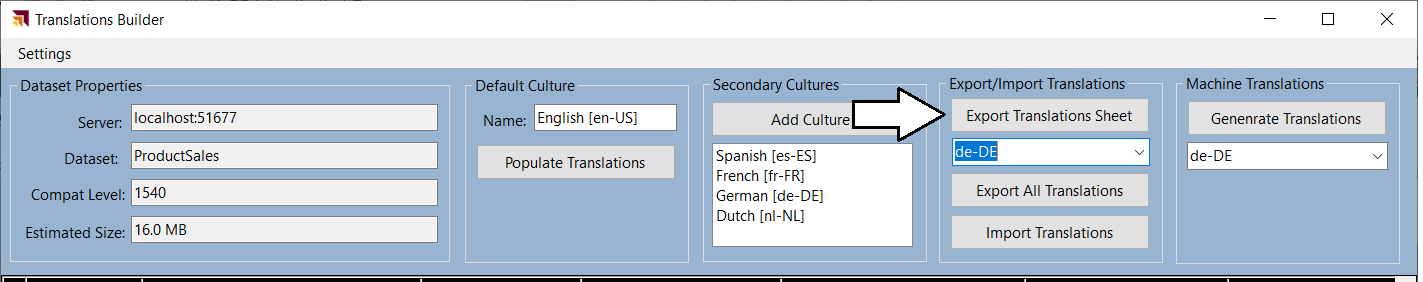
There is one important thing to note about loading reports with the **language** query string parameter. At the time of this writing in July of 2021, loading a report with the **language** query string parameter does not change the behavior of the **UserCulture** function in DAX. If you have written measures which use the **UserCulture** function to conditionally return different values depending on the user's language or locale, they will not work correctly. Instead you must configure the browser setting for user language and user locale to effectively test measures that call **UserCulture**. Alternatively, you can test reports using Power BI embedding where you have completed control over which language and locale is used to load a report.

### Enable Human Workflows for Translation using Export and Import

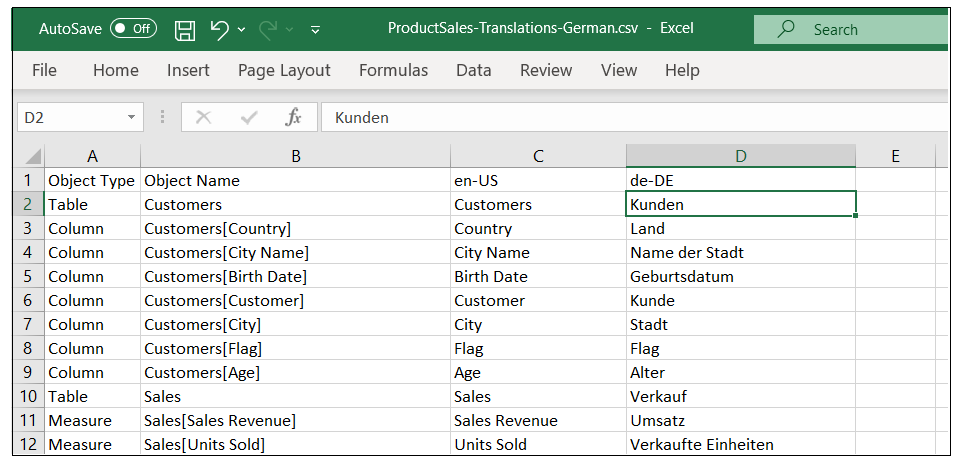
Another important consideration when building multi-language reports involves the human aspect of translating text values from one language to another. While it's possible to generate the first round of metadata translations using machine translations, it's likely you will eventually need to integrate those friendly carbon-based life forms (i.e. people) who play the role of translators to generate high quality translations. Furthermore, you cannot expect that people who work as professional translators will be able to use an advanced Power BI data modeling tool like the Tabular Editor.

While it's theoretically possible to have human translators work on files generated by the **Export Translations** command of the Tabular Editor, the JSON-based format will likely be rejected by professional translator teams due to it being a non-standard file format that is hard to work with. Once you begin writing custom code with TOM, however, you can generate the translation files that are sent out to a translation team using whatever file format they require.

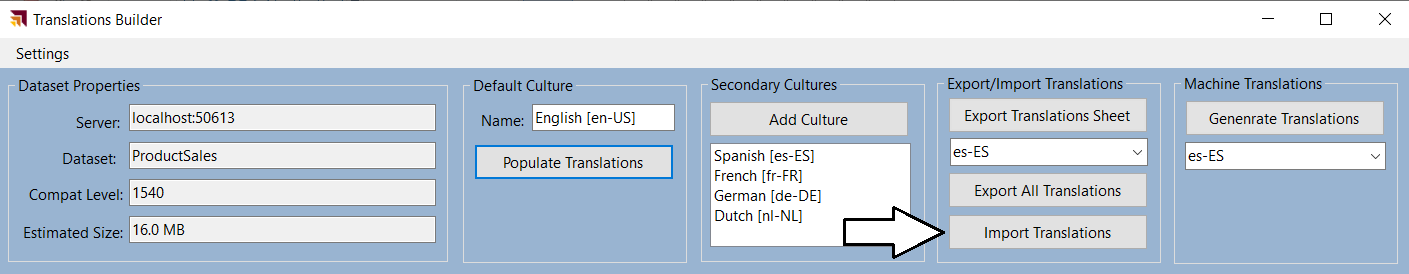
If you are working with a professional translation team, you might be requires to generate translation files in a standard translation format such as RESX files or XLIFF files. Depending on the requirements of the translator team, you might be able to generate translations files in easy-to-use formats such as CSV files or XLSX files. The **TranslationsBuilder** application provides an **Export Translation Sheet** command which demonstrates how to generate a translation sheet for a human translator using a simple CSV format.



When you click the **Export Translation Sheet** button, the **TranslationsBuilder** application generates a CSV file for the selected language using a naming format (e.g. **ProductSales-Translations-German.csv**) which includes the dataset name and the language for translation. After generating the file and saving it to the location of **TranslationsOutboxFolderPath** the **TranslationsBuilder** application will then open the translation sheet in Microsoft Excel.



Once these translation files have been generated, they can be sent out to the translation team. Once these translations files have been edited and returned with the high-quality human translations, you can then use the **Import Translations** command to important these human translations which will then effectively overwrite the machine translations.



### Embed Power BI Reports with Specific Languages and Locales

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 using the **localeSettings** parameter. The **localeSettings** parameter is an object with a **language** property and a **formatLocale** property that can be included as part of the **config** object that is passed in the call to **powerbi.embed** as shown in the following code.

let config = {

type: "report",

id: reportId,

embedUrl: embedUrl,

accessToken: embedToken,

tokenType: models.TokenType.Embed,

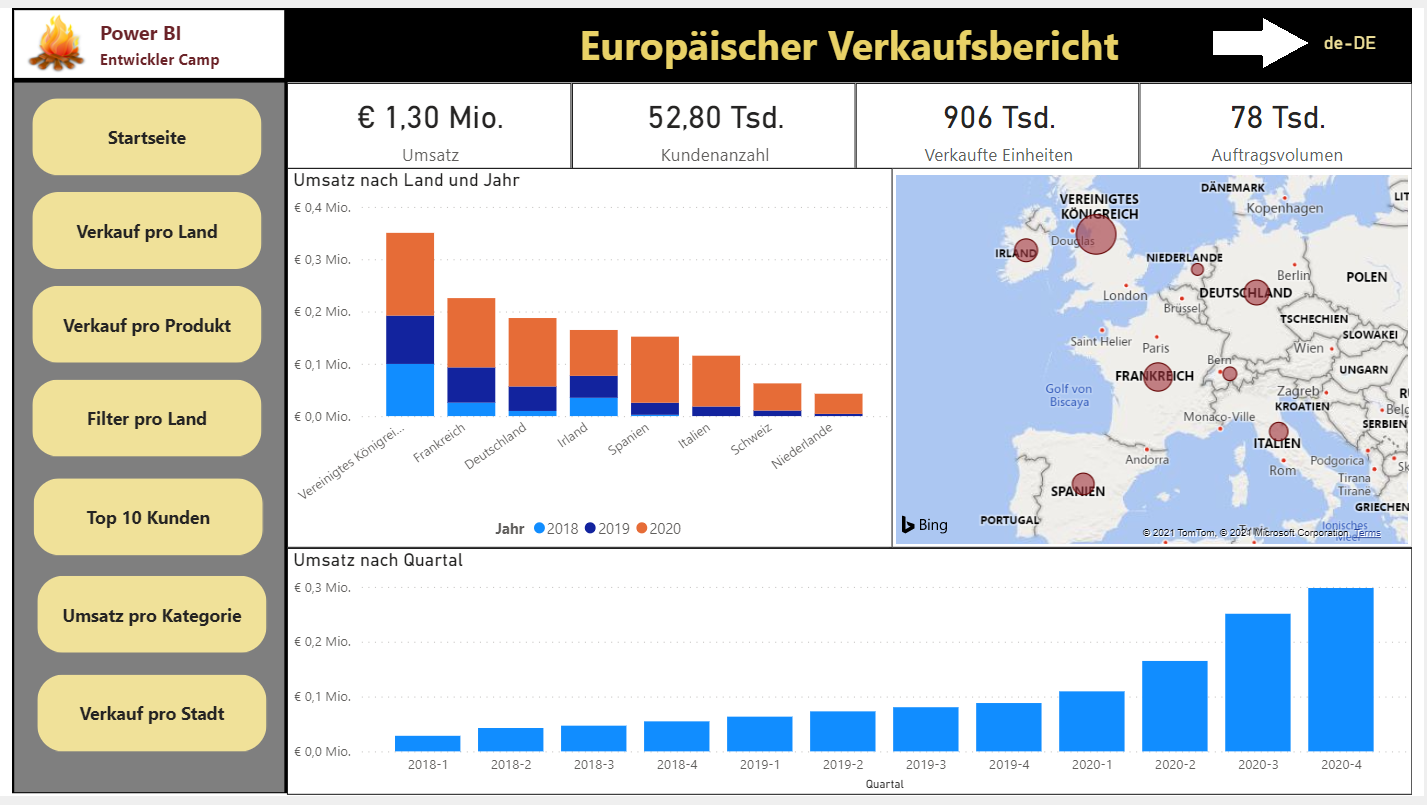
localeSettings: { language: "de-DE", formatLocale: "de-DE" }

};

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

When you embed reports using an explicit value for the **formatLocale** parameter, the **UserCulture** function will work correctly. That means you can write and test measures that conditionally return values based on the user's locale. This is different from the scenario in which you’re loading reports in the Power BI Service using the **language** query string parameter where the return value of the **UserCulture** function does not return the expected value.

In the screenshot below, you can see a visual in the top right displaying a text value of **de-DE**. This visual displays the value returned by the **UserCulture** function. You can inspect this value when loading a report into the Power BI Service or with Power BI embedding to determine whether the **UserCulture** is returning the language and locale you expect.



This concludes the coverage of developing an external tool with custom code and TOM to automate the development tasks associated with creating and maintaining metadata translations in a multi-language reporting solution. Now it’s time to move ahead to the final section which addresses the ***why***, ***when*** and ***how*** of implementing data translations.

## Design and Implement a Data Translations Strategy

While all multi-language reports will require metadata translations, you cannot assume the same for data translations. Some projects will require data translations and others will not. In order the 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. 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 on your own by extending a dataset definition with fundamental building blocks such as tables, columns, measures and filters. This article walks through one possible solution for implementing data translations by examining how this requirement is met in the **ProductSales.pbix** developer solution.

### Determine Whether Your Solution 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. Metadata translations must be implemented is 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 **ProductSales.pbix** developers sample 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 make 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. Your translation team might have trouble scaling to handle that volume of language translation by humans.

You also have to consider that your commitment to implement data translations often requires on-going maintenance. Every time someone adds a new record to the underlying database, you might be required to add new data translations for it. This is very different from implementing metadata translations where you create a finite number of translation for database objects and your work is done. Metadata translations don't require on-going maintenance as long as the underlying dataset schema remains the same.

In summary, there are many factors that go into deciding whether you should implement data translations. You must decide whether you can afford to spend the time and effort it takes to implement data translations properly. You might decide that implementing metadata translations goes far enough. If your primary goal is to make your reporting solution compliant with laws or regulations, you might that find implementing metadata translations is all you need.

### Modify the Dataset Design to Support Data Translations

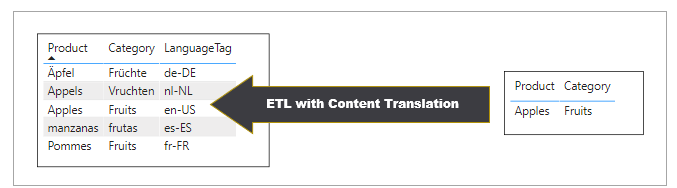
Long before Microsoft introduced Power BI, software developers around the world have been building multi-language applications that support data translations. After two decades of designing and refining various database designs, several common design patterns have emerged as industry best practices to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

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

Until dynamic column support is added to Power BI, it will be difficult to implement an efficient data translations stategy based on adding a column for each language. Currently, you would be required to use measures instead of columns but that approach is very limiting because measures do not provide row context. For example, measures cannot be used to supply values for the axes in a bar chart or line chart. Furthermore, measures cannot be used in the data roles of a legend or to filter data using slicers.

When it comes to implementing data translation strategy in Power BI, this is a new frontier for the Power BI community. Finding the best design approach for implementing data translations for any particular project will require innovative thinking. As an example, Chris Webb from the Power BI Customer Advisory Team (PBICAT) wrote a [blog post](https://blog.crossjoin.co.uk/2021/02/21/implementing-data-as-well-as-metadata-translations-in-power-bi/) in February of 2021 which discusses an advanced design approach which uses DirectQuery datasets and new functionality of the [composite model](https://docs.microsoft.com/en-us/power-bi/transform-model/desktop-composite-models) to implement a solution for data translations.

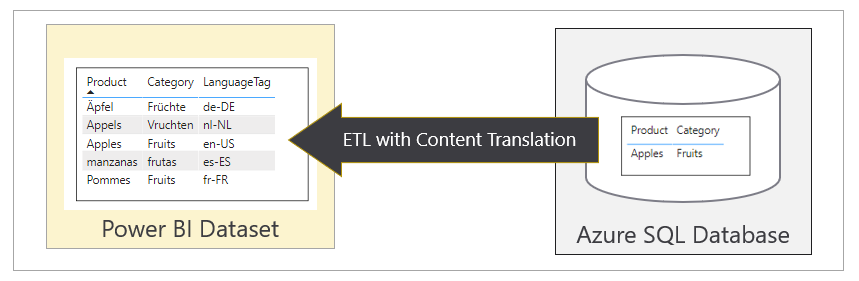
The **ProductSales.pbix** developer sample demonstrates implementing data translations using row replication with an import-mode dataset. Consider a simple example of this pattern. Let's say the **Products** table contains two text columns named **Product** and **Category** and you'd like your report to support five different languages including English, Spanish, French, German and Dutch. For each product in the **Products** table, you need to generate five records where each record contains the product name and product category translated to a specific language. Whenever the report is loaded, a row filter is applied to the **LanguageTag** column so that users only see rows for one language at a time.



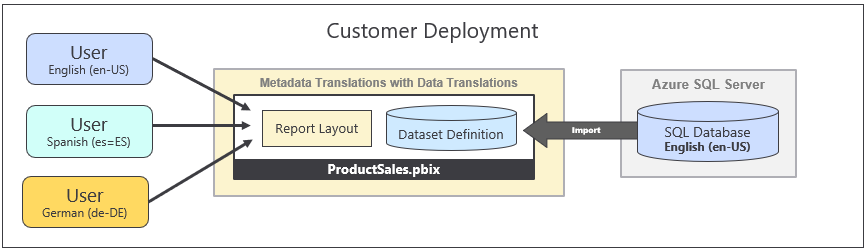
When using row replication to implement data translations in Power BI, you must implement all the ETL logic to generate the extra rows with translated content. Some multi-language applications add support for data translations at the database level and use an ETL tool such as SQL Server Integration Services (SSIS) to generate the extra rows containing the translations. It's relatively easy to build multi-language reports if the underlying database already implements data translations. However, it can be quite a bit of work if you need to extend an existing database.

### Use Power Query to Generate Data Translations Rows

The **ProductSales.pbix** developer sample demonstrates implementing data translations using row replication. The ETL logic for this solution is implemented using Power Query and M code which leverages lookup tables with translated content to generate the extra rows while importing data from the source database.

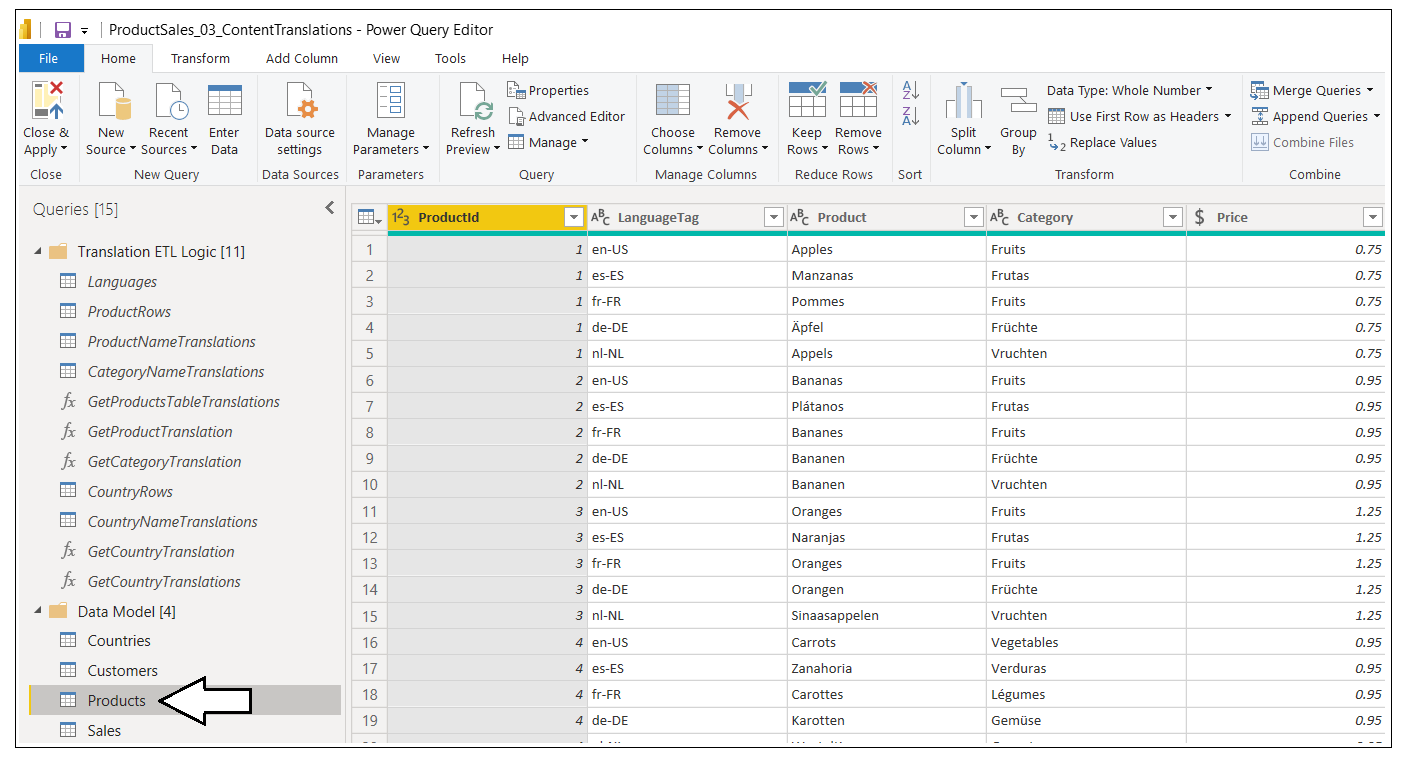


The following diagram shows the use case for the **ProductSales.pbix** developer sample. Note that this approach eliminates the need to redesign the underlying database to support data translations. Instead, all the ETL logic used to implemented data translations can be packaged and maintained inside a PBIX template file.



While implementing a data translations strategy with Power Query isn't always the right choice, it's great for scenarios where you don't have either the authority or the time it takes to implement data translations support at the database level. If you do decide to use this strategy, you'll find the writing Power Query logic in the M programming language provides a very elegant way to generate replicated rows with translated content during a dataset import operation.

If you'd like to review the M code with the ETL logic which demonstrates how to implement data translations, you can open the PBIX solution file named [**ProductSales\_03\_ContentTranslations.pbix**](https://github.com/PowerBiDevCamp/Multilanguage-Reports/raw/main/PBIX/ProductSales_03_ContentTranslations.pbix) in Power BI Desktop. Once you open this project, you can navigate to the Power Query Editor window to review the queries inside this project. The following screenshot shows all the queries in the project including the **Products** query whose output generates the **Products** table which loads a sperate **Product** table row for each supported language.



As you begin to work with this PBIX project in the Power Query Editor, you will be prompted to supply credentials for an Azure SQL Database named **EuropeanProductSales** with server path **devcamp.database.windows.net**. You should select a credential type of **Database** and then enter a user name of **CptStudent** and a password of **pass@word1**. You must also set the privacy levels of all datasource to a level of **Organization** or select to ignore privacy levels.

Now it's time to examine a few queries in the **ProductSales.pbix** developer solution so you can see how Power Query can be used to generate the extra rows required in the row replication pattern. Let's begin by examining the query named **Languages** which generates a table based on the following M code.

let

OutputSchema = type table [ LanguageTag=text, DisplayName=text, NativeName=text, SortOrder=number ],

Languages = #table(OutputSchema, {

{ "en-US", "English", "English", 1 },

{ "es-ES", "Spanish", "español", 2 },

{ "fr-FR", "French", "français", 3 },

{ "de-DE", "German", "Deutsch", 4 },

{ "nl-NL", "Dutch", "Nederlands", 5 }

})

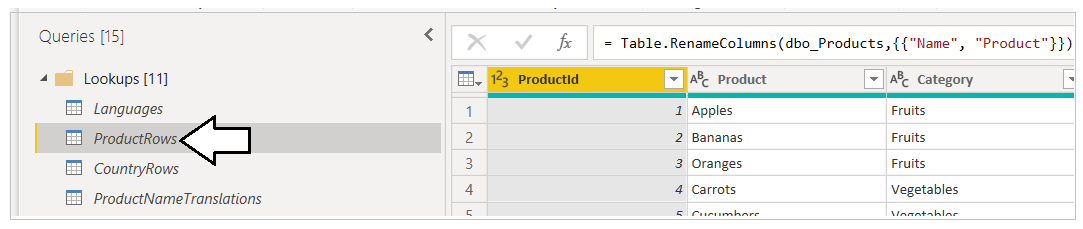
in

Languages

The **Languages** query generates a table with a row for each language which will be used in the row expansion process.



Next, let's examine the query named **ProductRows** query which creates a table with a single row for each product. The values in the **Product** column and the **Category** column are still in the default language which in this case is English.



As you have seen, the **Languages** query generates a table which contains a row for each language and the **ProductRows** query generates a table with a row for each product. The problem to solve now is how to duplicate the output of the **ProductRows** query once for each language with translated content for the **Product** column and the **Category** column.

The **ProductSales.pbix** developer sample contains a query function named **GetProductTableTranslations** which accepts a single text parameter named **language** and returns a table with the rows of the **Products** table where the column values for **Property** and **Category** have been translated to a target language. The following code listing shows the M programming logic in the **GetProductTableTranslations** query function that accomplishes this task.

(language as text) =>

Let

// (1) get rows from Products table

Source = ProductRows,

// (2) add ProductTranslation column with translated product name

AddProductTranslation = Table.AddColumn(Source, "ProductTranslation",

each GetProductTranslation([Product], language)),

// (3) add CategoryTranslation column with translated product name

AddCategoryTranslation = Table.AddColumn(AddProductTranslation, "CategoryTranslation",

each GetCategoryTranslation([Category], language)),

// (4) remove Product column and Category column with orginal values

RemoveColumns = Table.RemoveColumns(AddCategoryTranslation,{"Product", "Category"}),

// (5) rename ProductTranslation and CategoryTranslation column to Product and Category

RenamedColumns = Table.RenameColumns(RemoveColumns,{{"ProductTranslation", "Product"},

{"CategoryTranslation", "Category"}}),

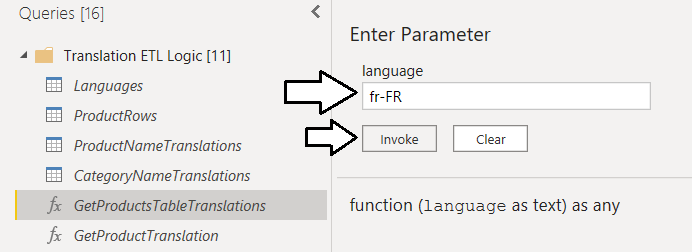
// (6) set datatype of Product column and Category column to text

Output = Table.TransformColumnTypes(RenamedColumns,{{"Product", type text}, {"Category", type text}})

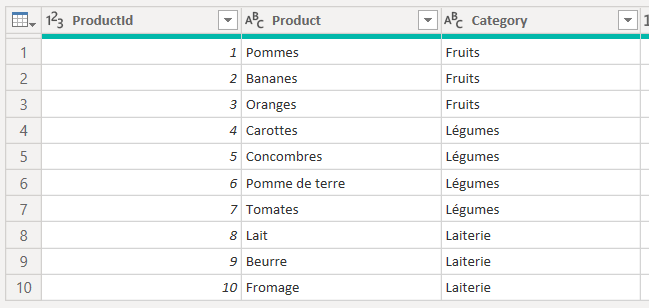
In

Output

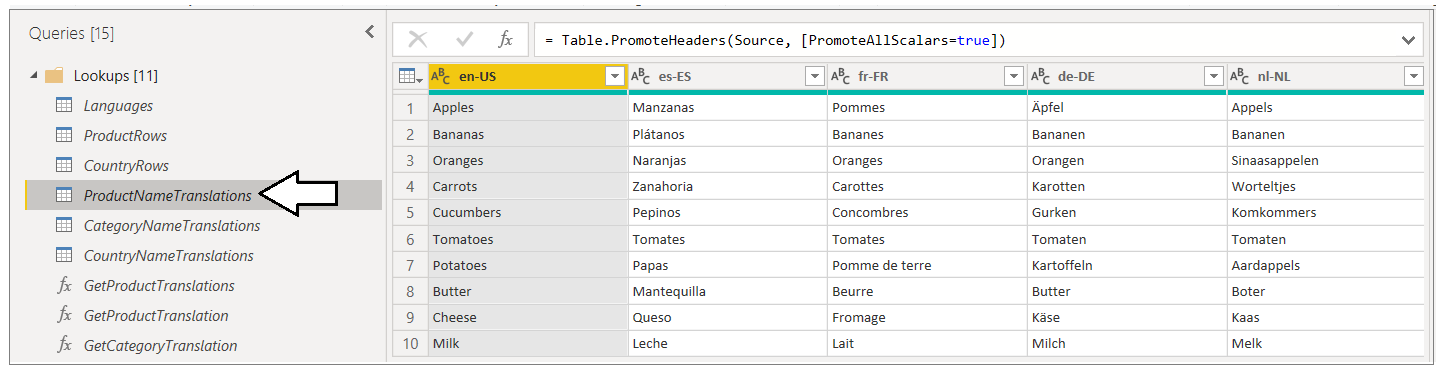
If you want to test the **GetProductTableTranslations** query function in the Power Query Editor window, you can enter a value for the **language** parameter using one of the supported languages such as **fr-FR** and then click the **Invoke** button.



You can see that **GetProductTableTranslations** generates a new set of **Product** table rows for a specific language.



Let's now examine how product names and category names are translated for each language? The solution involves loading lookup tables that have the translations for each product name and product category. For example, there is a query named **ProductNameTranslations** which returns the data from a [product translation sheet](https://github.com/PowerBiDevCamp/Multilanguage-Reports/blob/main/Data/ProductSalesTranslations/ProductSales-Translations-ProductName.csv).



When the **GetProductTableTranslations** query runs, it calls two query functions named **GetProductTranslation** and **GetCategoryTranslation** to get the translations for each product name and category. The **GetProductTranslation** query function accepts two parameters named **product** and **language** and uses these parameters to locate the correct row and column in the lookup table from which it can extract and return the correct product name translation.

(product as text, langauge as text) =>

let

LookupTable = ProductNameTranslations,

LookupTableRow = Table.SelectRows(LookupTable, each [#"en-US"] = product),

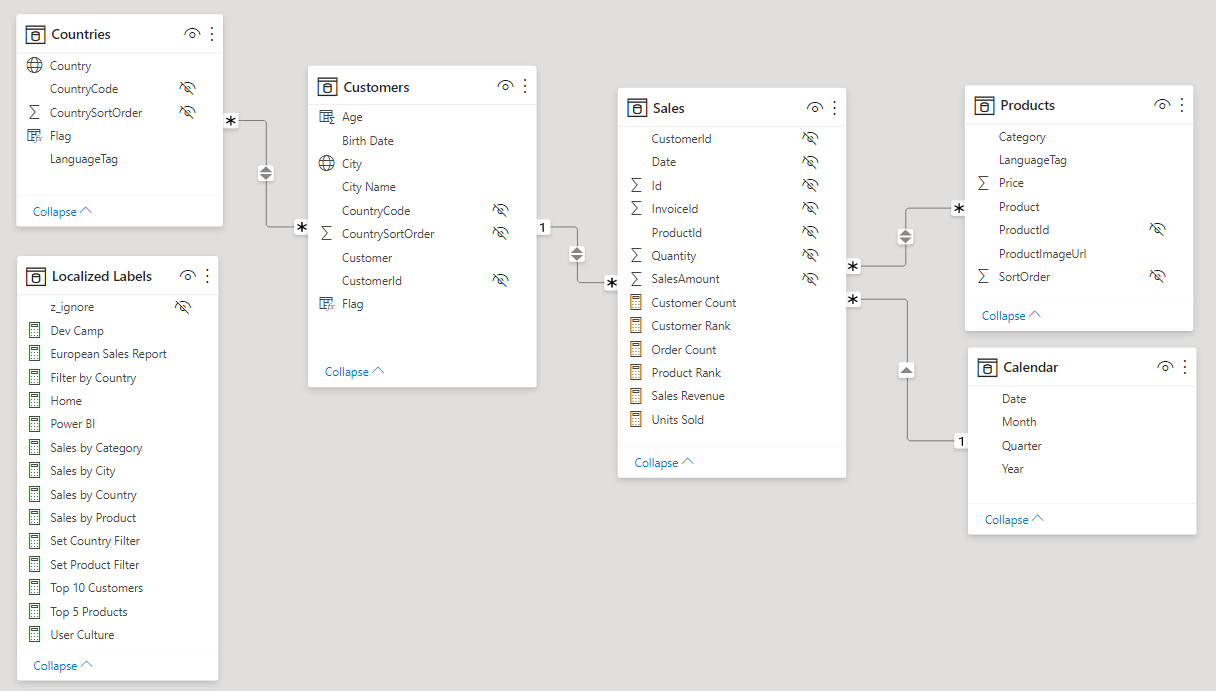
LookupTableRowColumn = Table.SelectColumns(LookupTableRow, langauge),

Translation = Table.FirstValue(LookupTableRowColumn)

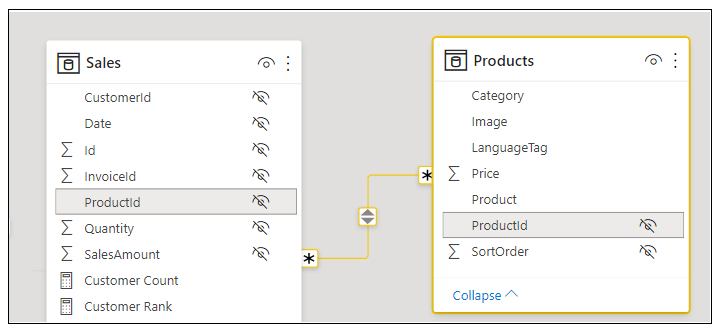
in

Translation

You have just reviewed the queries used to load the **Products** table with the row replication strategy which supplies data translations for product names and product categories in five different languages. There are other queries in the project designed to load the **Countries** table with the row replication strategy to provide data translations for country names as well. The following screenshot shows the tables and fields in the data model that are loaded into memory.



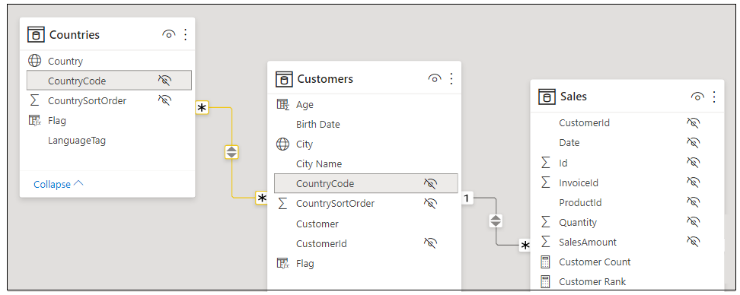
When you are implementing the row replication strategy, there is an issue concerning the relationships between fact tables and dimension tables. The issue is the **Products** table will now have more than one row with the same **ProductId** value. Since the **ProductId** column has duplicate values in both the **Sales** table and the **Products** table, you cannot use this column to create a one-to-many relationship. Instead you must create the relationship between these two tables as a many-to-many relationship.



While Power BI supports many-to-many relationships between tables, they are much less efficient when compared to one-to-many relationships. When using many-to-many relationships, you might begin to experience performance problems as the number of rows in a dimension table with replicated rows increases. Performance will be fine as the number of product rows increase from 10 to 100 and even up to 1000. However, as the number of rows in the **Products** table rises above 1000 approaching 10,000 rows or more, it's likely that performance issues will begin to appear and query times might become unacceptably long.

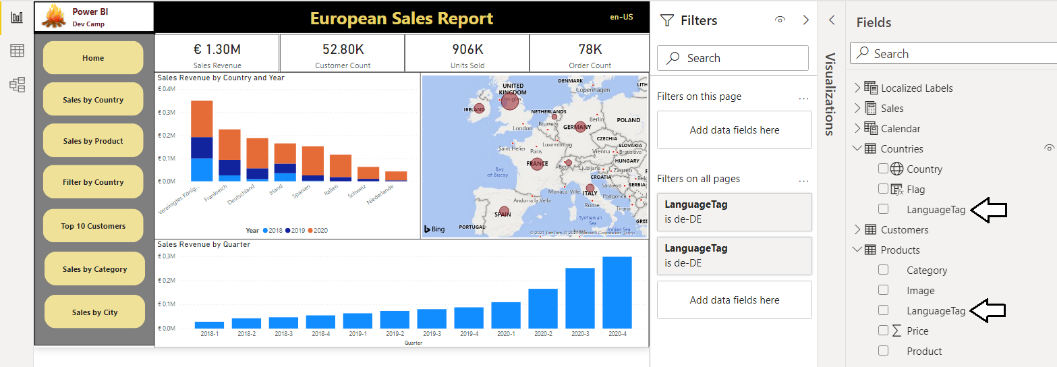
Now let's examine how this solution provides translated **Country** names. It would be possible to use the replicated row strategy on the **Customer** table. However, the **Customer** table is expected to have at least 50,000 rows. This creates a potential performance issue because you'd have to create a many-to-many relationship between two tables where each tables contain a large number of rows. Many-to-many relationships will not scale when used in this fashion.

A better solution is to change the data model from a simple star schema to a snowflake schema. You can accomplish this by factoring out country names from the **Customers** table into a separate **Countries** table. You should observe that this design makes it possible to maintain a one-to-many relationship between the **Customers** table and the **Sales** table which is essential for performance as both tables will have large row counts. While there is a many-to-many relationship between the **Countries** table and the **Customers** table, the number of rows in the **Countries** is small. Since there are six countries and five supported languages, there will only be 30 rows (6 countries \* 5 languages) in the **Countries** table.

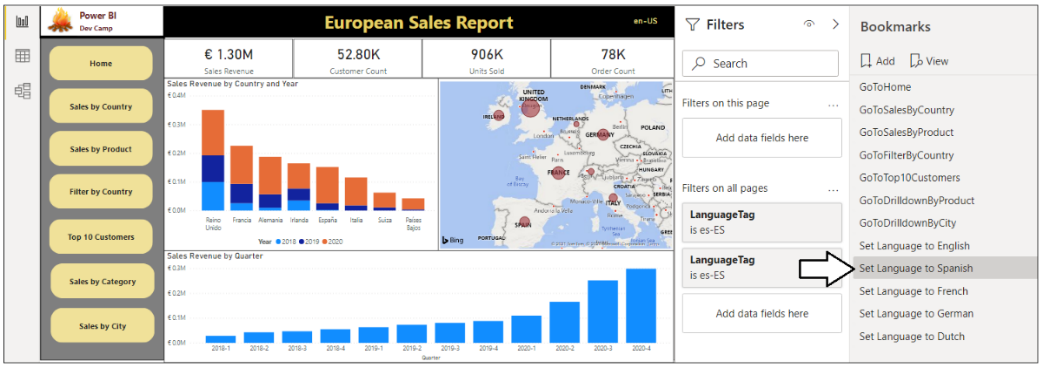


### Load Reports using Bookmarks to Filter Data Translations

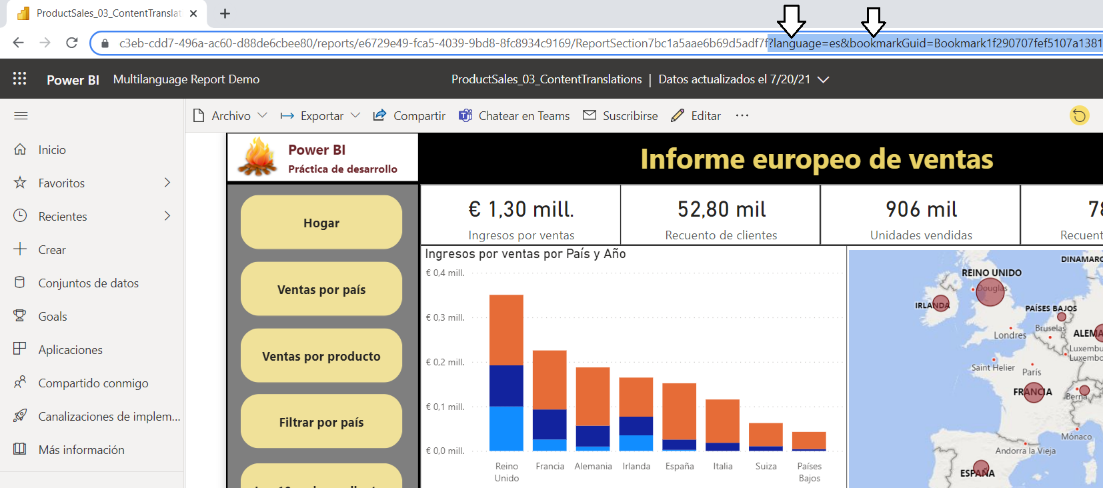
Once you have created the queries to load the dimension tables which use the row replication strategy, you must then figure out how to filter table rows so a user only sees the rows for one language at a time. In the **ProductSales.pbix** developer sample, the two tables that require filtering are **Products** and **Countries**. The following screenshot shows how you can use the **Filter** pane in Power BI Desktop to set report-level filtering on the **LanguageTag** column on both the **Products** table and the **Countries** table so only rows with **German** translations are displayed to the user.



The best way to control filtering in a Power BI report is to create a set of bookmarks. The **ProductSales.pbix** developer sample includes a set of bookmarks that can be used to apply the filtering required for each of the supported languages. Now you can simply apply a bookmark to set the data translations filtering as shown in the following screenshot.



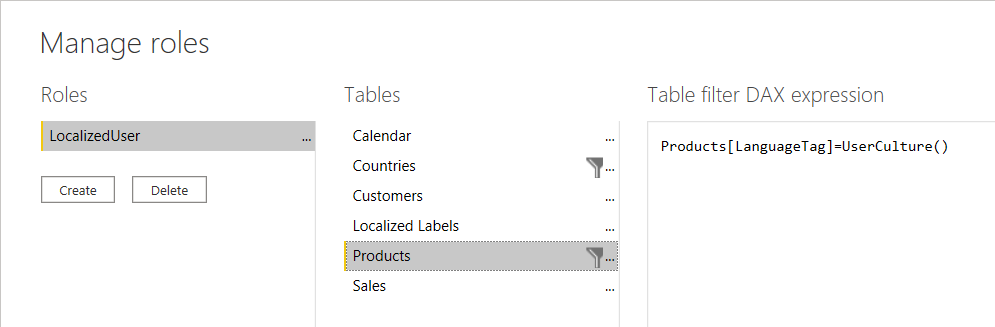
Earlier in this article, you learned that it is possible to open a report in the Power BI Service using the **language** query string parameter to force a report to load the metadata translations for a specific language. Now that the report implements data translations in addition to metadata translations, it is now necessary to pass a second query string parameter to apply a bookmark. This query string parameter is named **bookmarkGuid** and it makes it possible to apply a bookmark as the report is loading before anything is displayed to the user.



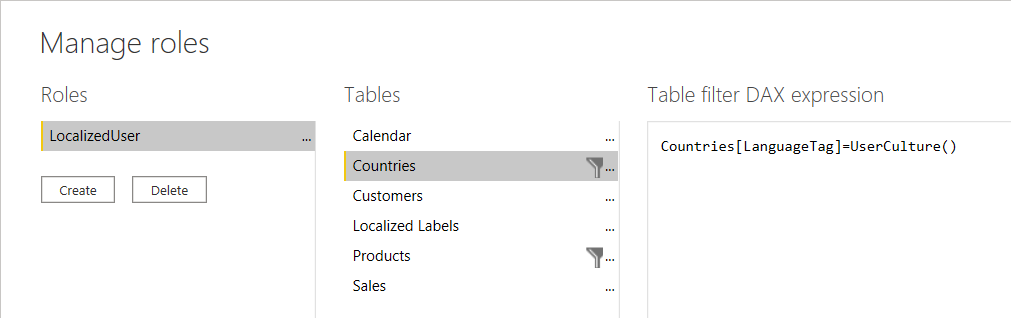
### Setting the Language for Current User using RLS and UserCulture

If you deploy a multi-language report through the Power BI Service, your deployment scenario might require you to use URLs with query string parameters for **language** and **bookmarkGuid** to ensure that reports load with the correct metadata translations and the data translations filtering for a specific language. When you develop with Power BI embedding you have more control because you can embed the report with a **localeSettings** object which allows you to the specify the **language** and the **formatLocale**.

If you are developing multi-language reports for use in a custom application that uses Power BI embedding, you can control data translations filtering using the Power BI features for Row Level Security (RLS). This approach involves creating a single RLS role named **LocalizedUser**. This RLS role applies a filter on the **Products** table where the **LanguageTag** column is equal to the return value of the **UserCulture** function.



The RLS role named **LocalizedUser** also applies a second filter on the **Countries** table where the **LanguageTag** column is equal to the return value of the **UserCulture** function.



If you are developing with App-Owns-Data embedding, you will be required to generate embed tokens with an effective identity which includes the RLS role named **LocalizedUser**. The following code listing demonstrates how to generate an embed token with an effective identity containing the **LocalizedUser** role using the Power BI .NET SDK.



## Summary

This article has examined how to use Power BI localization features to design and implement multi-language reports. Along the way you learned that Power BI datasets provide support for localization while the Power BI report designer does not. This partial support for localization in Power BI can lead to confusion as many common Power BI Desktop report design techniques do not support localization and must be avoided.

Becoming successful at building multi-language reports requires a deep understanding of Power BI architecture and a thorough knowledge of which report design techniques support localization. You 've learned how to prepare datasets and reports for localization and how to create the **Localized Labels** table so you can localize report labels such as titles, headings and button captions. You also learned several different approaches you can use to add metadata translations to a dataset definition. After reading this article, you should now possess a deeper understanding and the fundamental skills you need to build multi-language reports in a reliable and testable fashion.

This article also discussed when and how to implement data translations. You have learned that some projects will require data translations while other will not. You must decide whether to implement data translations on a project-by-project basis. Fortunately, you now know the right questions to ask in order to make that decision.

The **ProductSales.pbix** developer sample demonstrates how to implement data translations using the row replication pattern. While this design approach will work well for some scenarios, it might have trouble scaling in larger scenarios as the number of rows in the tables with translated content increases due to the use of many-to-many relationship. In the fullness of time, Power BI developers will continue to come up with creative and innovative designs to implement an efficient data translation strategy for these types of scenarios.

Also remember that one fine day in the future that Power BI will introduce support for calculated columns with dynamic evaluation. That will certainly open up opportunities for creating new designs for implementing data translation strategies that are more performant and less complex than using the row replication pattern which was demonstrated in this article with the **ProductSales.pbix** developer solution.