

Contents

Overview

[What is Power Query?](#)

Quick Starts

[Using Power Query in Power BI](#)

[Using Query Parameters in Power BI Desktop](#)

[Combining Data](#)

[Installing the PowerQuery SDK](#)

[Starting to Develop Custom Connectors](#)

[Creating your first connector - Hello World](#)

Tutorials

[Shape and combine data using Power Query Editor](#)

[Connector Development](#)

[TripPin Walkthrough](#)

[Overview](#)

[1. OData](#)

[2. REST API](#)

[3. Navigation Tables](#)

[4. Paths](#)

[5. Paging](#)

[6. Schemas](#)

[7. Advanced Schemas](#)

[8. Diagnostics](#)

[9. Test Connection](#)

[10. Folding](#)

[OAuth Tutorials](#)

[Github](#)

[MyGraph](#)

Samples

[Functional Samples](#)

[ODBC Samples](#)

[TripPin Samples](#)

[Concepts](#)

[Certification](#)

[Power Query Online Limits](#)

[Power Query query folding](#)

[Reference](#)

[Authoring Reference](#)

[Combine Files](#)

[Common Issues](#)

[Connectors](#)

[Excel](#)

[XML](#)

[PostgreSQL](#)

[SQL Server database](#)

[Text/CSV](#)

[ODBC](#)

[Connecting to VSTS via OData](#)

[Data Lake Storage](#)

[Connector Development Reference](#)

[Handling Authentication](#)

[Handling Data Access](#)

[ODBC Development](#)

[Overview](#)

[ODBC Extensibility Functions](#)

[Parameters for your Data Source Function](#)

[Parameters for Odbc.DataSource](#)

[Troubleshooting and Testing](#)

[Handling Resource Path](#)

[Handling Paging](#)

[Handling Transformations](#)

[Static](#)

[Dynamic](#)

[Handling Schemas](#)

[Handling Status Codes](#)

[Default Behavior](#)

[Wait Retry Pattern](#)

[Handling Unit Testing](#)

[Helper Functions](#)

[Handling Documentation](#)

[Handling Navigation Tables](#)

[Handling Gateway Support](#)

[Handling Connector Signing](#)

Resources

[Power BI Documentation](#)

[M Function Reference](#)

[M Language Document](#)

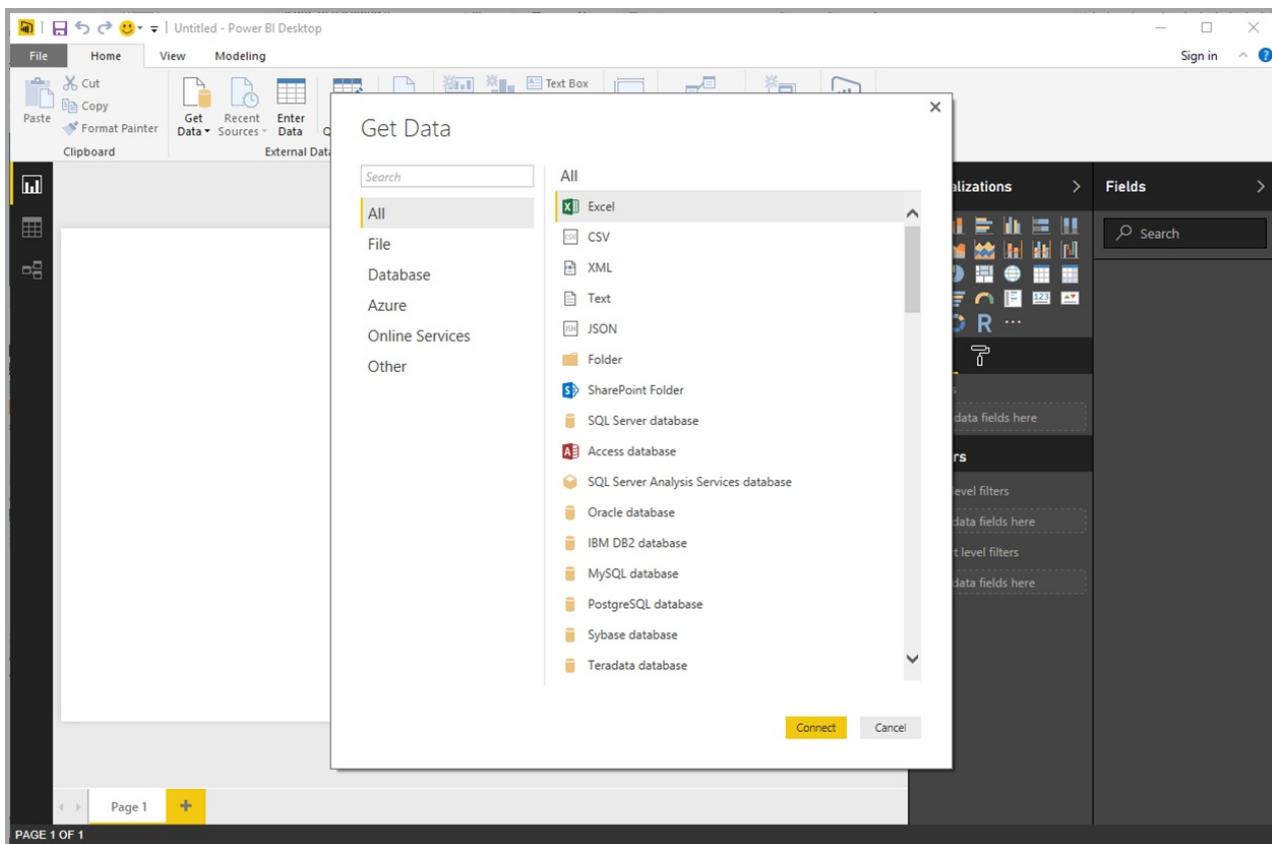
[M Type Reference](#)

What is Power Query?

3 minutes to read • [Edit Online](#)

Power Query is the Microsoft Data Connectivity and Data Preparation technology that enables business users to seamlessly access data stored in hundreds of data sources and reshape it to fit their needs, with an easy to use, engaging, and no-code user experience.

Supported data sources include a wide range of file types, databases, Microsoft Azure services, and many other third-party online services. **Power Query** also provides a **Custom Connectors SDK** so that third parties can create their own data connectors and seamlessly plug them into Power Query.



The **Power Query Editor** is the primary data preparation experience, allowing users to apply over 300 different data transformations by previewing data and selecting transformations in the user experience. These data transformation capabilities are common across all data sources, regardless of the underlying data source limitations.

Where to use Power Query

Power Query is natively integrated in several Microsoft products, including the following.

PRODUCT NAME	OVERVIEW
Microsoft Power BI	Power Query enables data analysts and report authors to connect and transform data as part of creating Power BI reports using Power BI Desktop .
Microsoft Excel	Power Query enables Excel users to import data from a wide range of data sources into Excel for analytics and visualizations. Starting with Excel 2016, Power Query capabilities are natively integrated and can be found under the " Get & Transform " section of the Data tab in the Excel Desktop ribbon. Excel 2010 and 2013 users can also leverage Power Query by installing the Microsoft Power Query for Excel add-in .
Microsoft SQL Server Data Tools for Visual Studio	Business Intelligence Developers can create Azure Analysis Services and SQL Server Analysis Services tabular models using SQL Server Data Tools for Visual Studio . Within this experience, users can leverage Power Query to access and reshape data as part of defining tabular models.
Microsoft Common Data Service for Apps	Common Data Service (CDS) for Apps lets you securely store and manage data that's used by business applications. Data within CDS for Apps is stored within a set of entities. An entity is a set of records used to store data, similar to how a table stores data within a database. CDS for Apps includes a base set of standard entities that cover typical scenarios, but you can also create custom entities specific to your organization and populate them with data using Power Query . App makers can then use PowerApps to build rich applications using this data.

How Power Query helps with data acquisition

Business users spend up to 80% of their time on data preparation, delaying the time to analysis and decision making. There are many challenges that cause this situation, and **Power Query** helps address many of them.

EXISTING CHALLENGE	HOW DOES POWER QUERY HELP?
Finding and connecting to data is too difficult	Power Query enables connectivity to a wide range (100s) of data sources, including data of all sizes and shapes.
Experiences for data connectivity are too fragmented	Consistency of experience, and parity of query capabilities over all data sources with Power Query.
Data often needs to be reshaped before consumption	Highly interactive and intuitive experience for rapidly and iteratively building queries over any data source, any size.
Any shaping is one-off and not repeatable	When using Power Query to access and transform data, users are defining a repeatable process (query) that can be easily refreshed in the future to get up-to-date data. In the event that the process/query needs to be modified to account for underlying data or schema changes, Power Query provides users with the ability to modify existing queries using the same interactive and intuitive experience that they used when initially defining their queries.
Volume (data sizes), Velocity (rate of change) and Variety (breadth of data sources and data shapes)	Power Query offers the ability to work against a subset of the entire data set in order to define the required data transformations, allowing users to easily filter down and transform their data to a manageable size. Power Query queries can be refreshed manually or by leveraging schedule refresh capabilities in specific products (such as Power BI) or even programmatically (using Excel's Object Model). Power Query provides connectivity to hundreds of data sources and over 350 different types of data transformations for each of these sources, allowing users to work with data from any source and in any shape.

Next steps

Next, learn how to use Power Query in **Power BI Desktop**.

- [Quickstart: Using Power Query in Power BI Desktop](#)

Quickstart: Using Power Query in Power BI Desktop

5 minutes to read • [Edit Online](#)

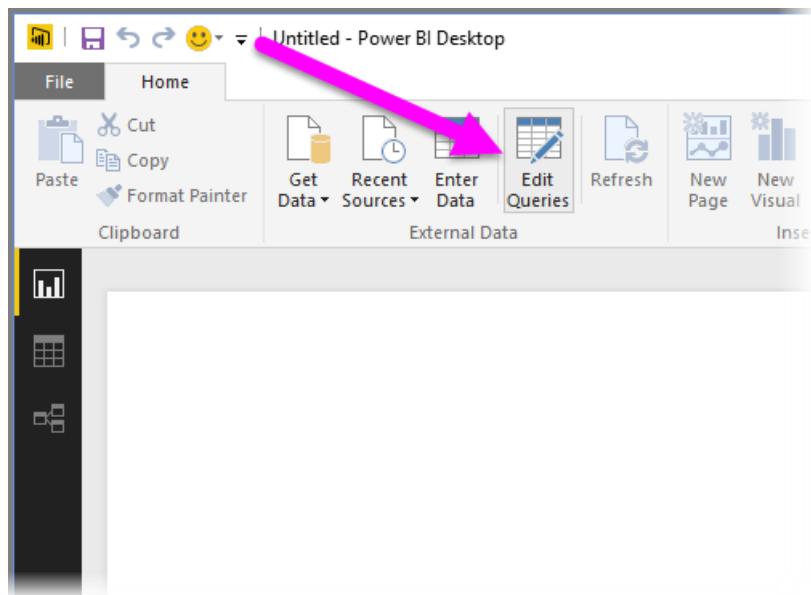
With **Power Query** in **Power BI** you can connect to many different data sources, transform the data into the shape you want, and quickly be ready to create reports and insights. When using Power BI Desktop, **Power Query** functionality is provided in the **Power Query Editor**.

Let's get acquainted with **Power Query Editor**.

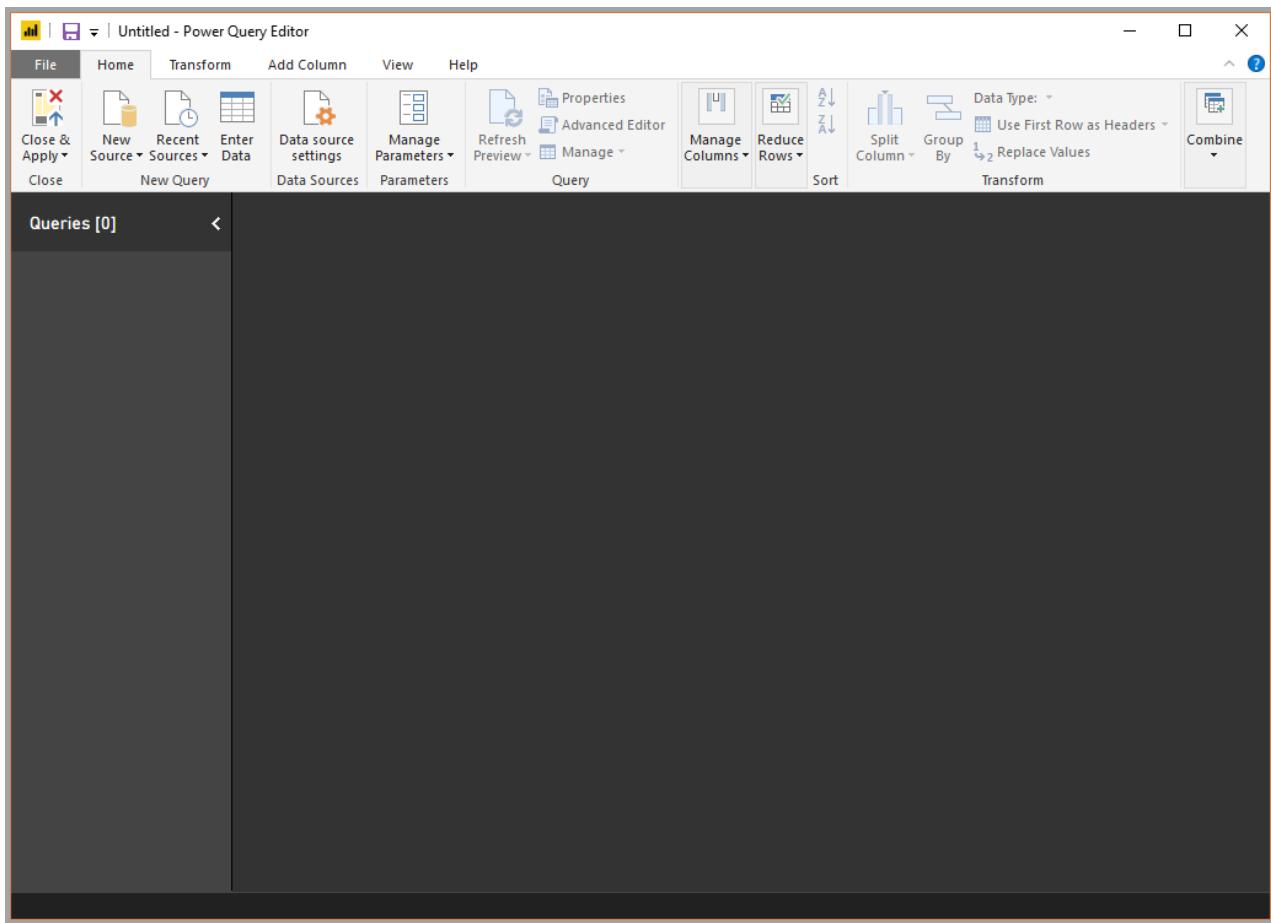
If you're not signed up for Power BI, you can [sign up for a free trial](#) before you begin. Also, Power BI Desktop is [free to download](#).

Using Power Query Editor

Power Query is made available in **Power BI Desktop** through **Power Query Editor**. To launch Power Query Editor, select **Edit Queries** from the **Home** tab of Power BI Desktop.



With no data connections, **Power Query Editor** appears as a blank pane, ready for data.



Once a query is loaded, **Power Query Editor** view becomes more interesting. If you connect to the following Web data source, **Power Query Editor** loads information about the data, which you can then begin to shape.

<https://www.bankrate.com/finance/retirement/best-places-retire-how-state-ranks.aspx>

Here's how **Power Query Editor** appears once a data connection is established:

- In the ribbon, many buttons are now active to interact with the data in the query.
- In the left pane, queries are listed and available for selection, viewing, and shaping.
- In the center pane, data from the selected query is displayed and available for shaping.
- The **Query Settings** window appears, listing the query's properties and applied steps.

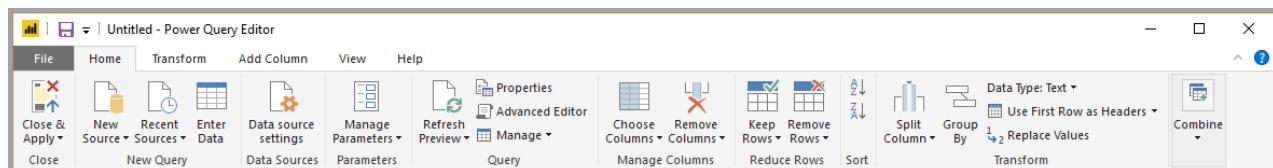
The screenshot shows the Power Query Editor interface. The ribbon at the top has tabs for File, Home (which is selected), Transform, Add Column, View, and Help. The Home tab contains icons for Close & Apply, New Source, Recent Sources, Enter Data, Data source settings, Manage Parameters, Refresh Preview, Advanced Editor, Properties, Choose Columns, Remove Columns, Keep Rows, Remove Rows, Sort, Split Column, Group By, Data Type (set to Text), Use First Row as Headers, Replace Values, and Transform. A preview window shows a table with columns: State, Overall rank, Cost of living, Crime, Culture, and Health care quality. The table lists 27 US states from 1 (South Dakota) to 27 (Vermont). The 'Overall rank' column contains values like 1, 2, 3, etc. The 'Cost of living' column contains values like 19, 25, 12, etc. The 'Culture' column contains values like 10, 15, 31, etc. The 'Health care quality' column contains values like 7, 9, 26, etc. The 'Transform' section of the ribbon shows a formula: = Table.TransformColumnTypes(Data0,{{"State", type text}, {"Overall rank", type number}, {"Cost of living", type number}, {"Crime", type number}, {"Culture", type number}, {"Health care quality", type number}}). The 'QUERY SETTINGS' pane on the right shows 'PROPERTIES' with a name field containing 'Ranking of best and worst states for future' and 'All Properties'. It also shows 'APPLIED STEPS' with 'Source' and 'Navigation' checked, and a step labeled 'Changed Type' which is highlighted with a pink circle. The status bar at the bottom right says 'PREVIEW DOWNLOADED AT 3:30 PM'.

The following sections describe each of these four areas—the ribbon, the queries pane, the data view, and the Query Settings pane.

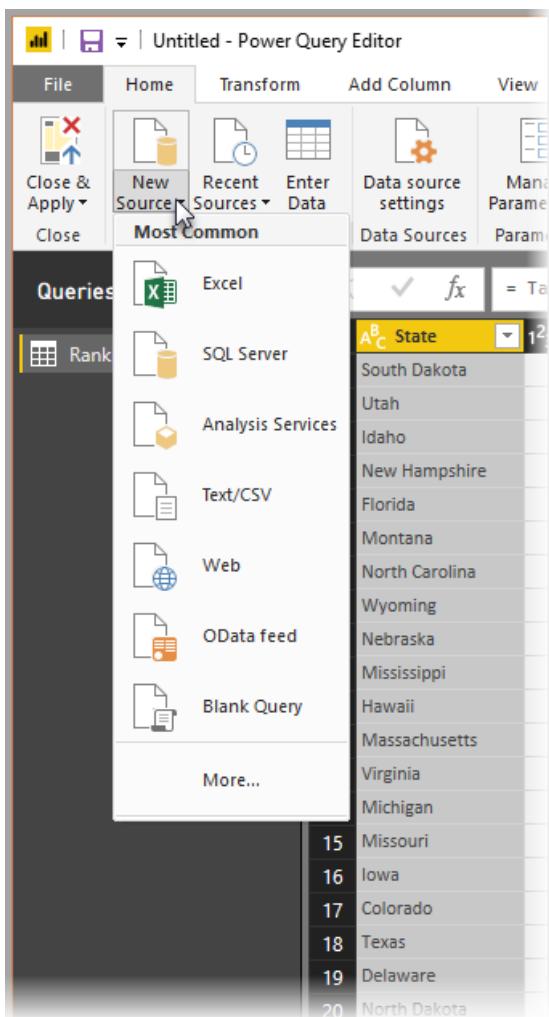
The query ribbon

The ribbon in **Power Query Editor** consists of five tabs—**Home**, **Transform**, **Add Column**, **View**, and **Help**.

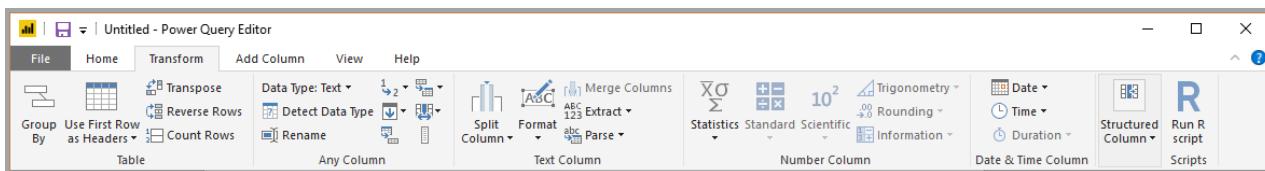
The **Home** tab contains the common query tasks, including the first step in any query, which is **Get Data**. The following image shows the **Home** ribbon.



To connect to data and begin the query building process, select the **Get Data** button. A menu appears, providing the most common data sources.



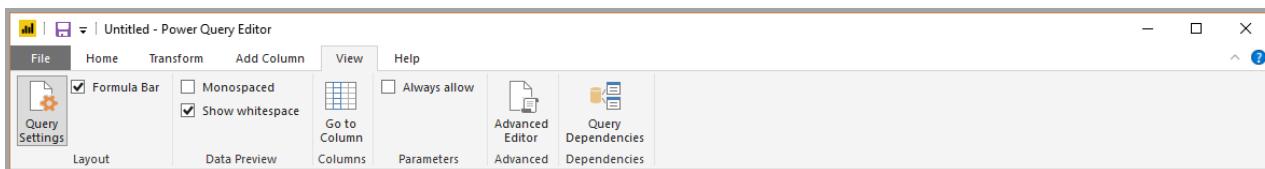
The **Transform** tab provides access to common data transformation tasks, such as adding or removing columns, changing data types, splitting columns, and other data-driven tasks. The following image shows the **Transform** tab.



The **Add Column** tab provides additional tasks associated with adding a column, formatting column data, and adding custom columns. The following image shows the **Add Column** tab.



The **View** tab on the ribbon is used to toggle whether certain panes or windows are displayed. It's also used to display the Advanced Editor. The following image shows the **View** tab.



It's useful to know that many of the tasks available from the ribbon are also available by right-clicking a column, or other data, in the center pane.

The left pane

The left pane displays the number of active queries, as well as the name of the query. When you select a query from the left pane, its data is displayed in the center pane, where you can shape and transform the data to meet your needs. The following image shows the left pane with multiple queries.

A screenshot of the Power Query Editor interface. The left pane is titled 'Queries [3]' and lists three items: 'Ranking of best and worst states', 'States of the United States', and 'Products_by_Categories'. The 'Ranking of best and worst states' query is selected, highlighted with a yellow background. The center pane displays a table with the following data:

	State	Overall rank	Cost of living	Crime
1	South Dakota	1	19	
2	Utah	2	25	
3	Idaho	3	12	
4	New Hampshire	4	43	
5	Florida	5	27	
6	Montana	6	23	
7	North Carolina	6	12	
8	Wyoming	8	28	
9	Nebraska	9	17	
10	Mississippi	10	1	
11	Hawaii	11	48	
12	Massachusetts	12	46	
13	Virginia	13	30	
14	Michigan	14	4	

The center (data) pane

In the center pane, or Data pane, data from the selected query is displayed. This is where much of the work of the Query view is accomplished.

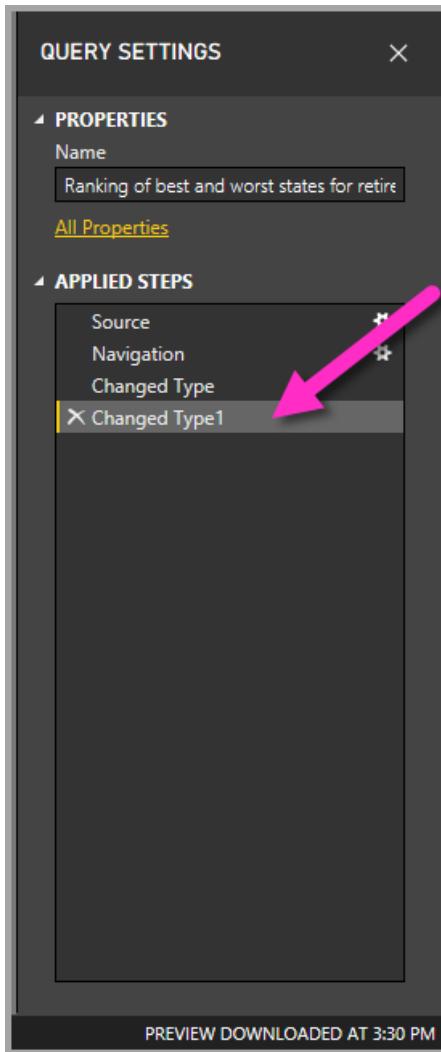
In the following image, the Web data connection established earlier is displayed, the **Overall score** column is selected, and its header is right-clicked to show the available menu items. Notice that many of these right-click menu items are the same as buttons in the ribbon tabs.

The screenshot shows the Power Query Editor interface with a query named "Ranking of best and worst..." containing 9 columns and 50 rows. A context menu is open over the "Health care quality" column, specifically at the step "Change Type". The "Applied Steps" pane on the right shows the step "Source" followed by "Navigation". The "Change Type" step is highlighted, and its sub-menu is displayed, with "Whole Number" being the selected option.

When you select a right-click menu item (or a ribbon button), Query applies the step to the data, and saves it as part of the query itself. The steps are recorded in the **Query Settings** pane in sequential order, as described in the next section.

The query settings pane

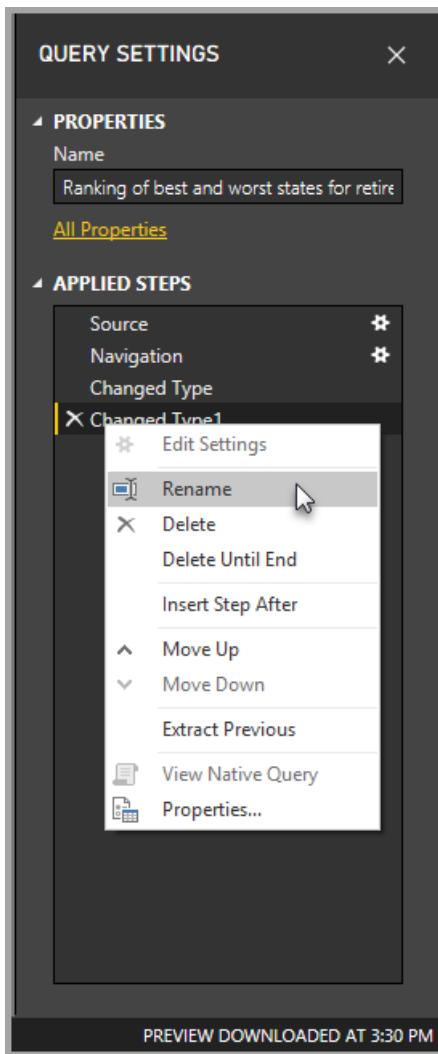
The **Query Settings** pane is where all steps associated with a query are displayed. For example, in the following image, the **Applied Steps** section of the **Query Settings** pane reflects the fact that the type of the **Overall score** column has changed.



As additional shaping steps are applied to the query, they are captured in the **Applied Steps** section.

It's important to know that the underlying data is *not* changed; rather, Power Query Editor adjusts and shapes its view of the data, and any interaction with the underlying data occurs based on Power Query Editor's shaped and modified view of that data.

In the **Query Settings** pane, you can rename steps, delete steps, or reorder the steps as you see fit. To do so, right-click the step in the **Applied Steps** section, and choose from the menu that appears. All query steps are carried out in the order they appear in the **Applied Steps** pane.



The Advanced Editor

If you want to see the code that **Power Query Editor** is creating with each step, or want to create your own shaping code, you can use the **Advanced Editor**. To launch the advanced editor, select **View** from the ribbon, then select **Advanced Editor**. A window appears, showing the existing query code.

```

let
    Source = Web.Page(Web.Contents("http://www.bankrate.com/finance/retirement/best-places-retire-how-state-1")),
    Data0 = Source{0}[Data],
    #"Changed Type" = Table.TransformColumnTypes(Data0,{{"State", type text}, {"Overall rank", Int64.Type}, {"Health care quality", type text}}),
    #"Changed Type1" = Table.TransformColumnTypes(#"Changed Type",{{"Health care quality", type text}}),
in
    #"Changed Type1"

```

No syntax errors have been detected.

Done Cancel

You can directly edit the code in the **Advanced Editor** window. To close the window, select the **Done** or **Cancel** button.

Saving your work

When your query is where you want it, you can have Power Query Editor apply the changes to the data model into Power BI Desktop, and close Power Query Editor. To do that, select **Close & Apply** from Power Query Editor's **File** menu.

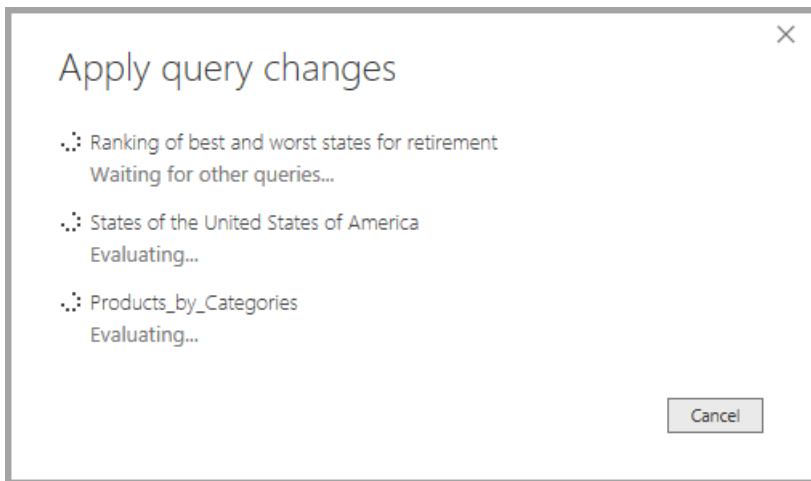
File Home Transform Add Column View Help

Close & Apply

Close the Query Editor window and apply any pending changes.

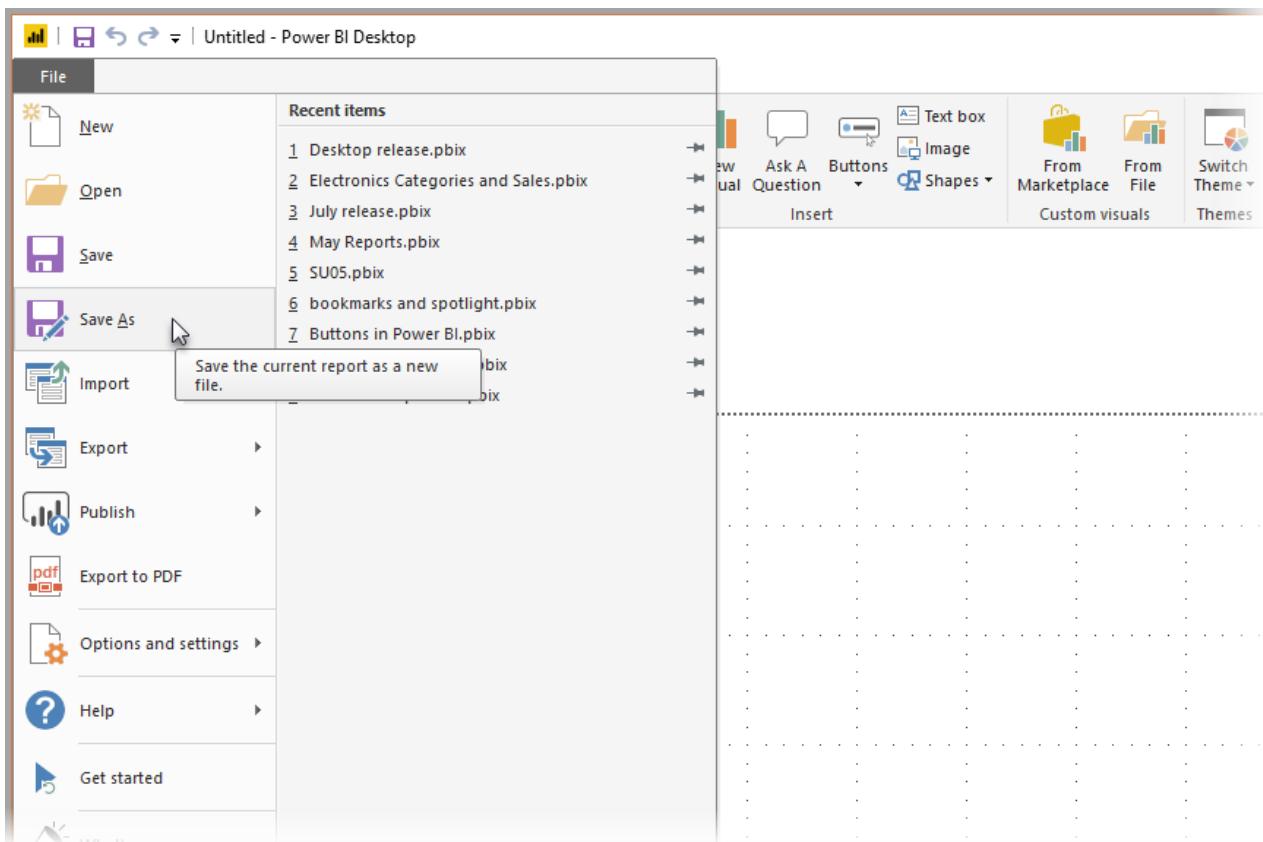
	Cost of living	Crime	Culture	Health care quality
1	19	21	10 12	
2	25	22	15 10	
3	12	4	31 8	
4	43	1	9 5	
5	27	33	26 36	
6	23	26	7 19	
7	12	29	40 30	
8	28	9	16 22	
9	17	18	25 12	
10	1	23	48 26	
11	48	35	3 17	
12	46	14	2 3	
13	20	8	17 24	

As progress is made, Power BI Desktop provides a dialog to display its status.



Once you have your query where you want it, or if you just want to make sure your work is saved, Power BI Desktop can save your work in a .pbix file.

To save your work, select **File > Save** (or **File > Save As**), as shown in the following image.



Next steps

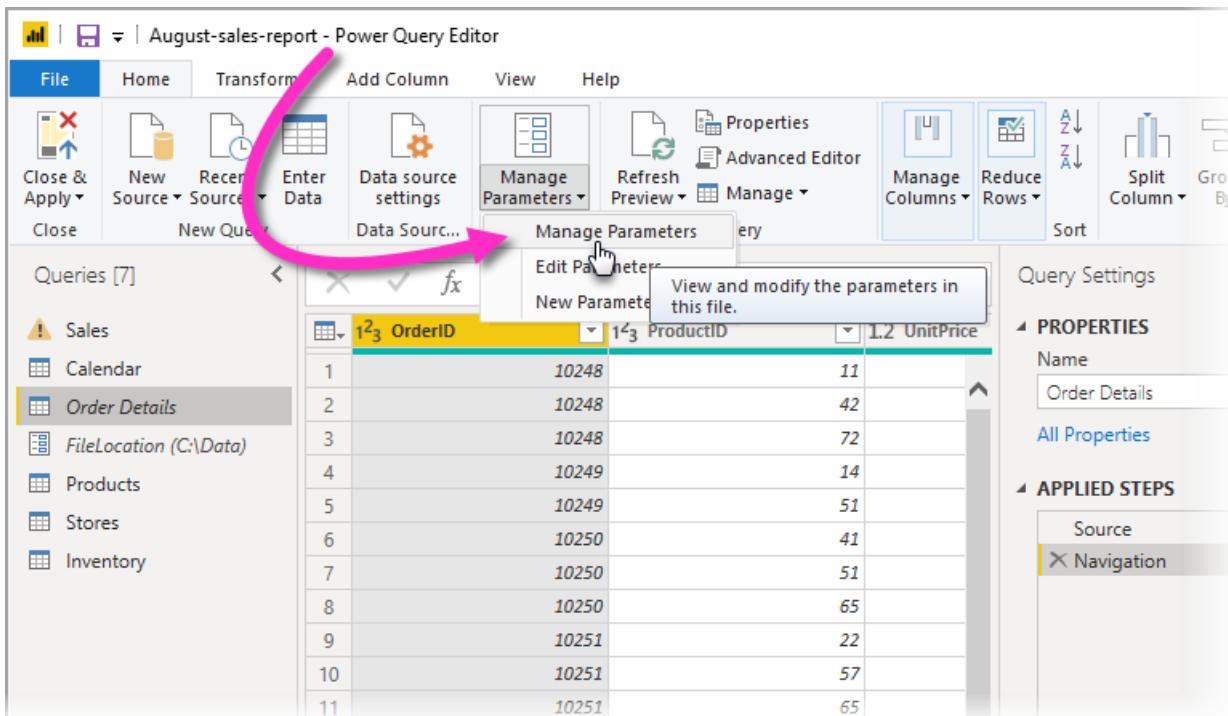
In this quickstart you learned how to use Power Query Editor in Power BI Desktop, and how to connect to data sources. To learn more, continue with the tutorial on shaping and transforming data with Power Query.

[Power Query tutorial](#)

Using Query Parameters in Power BI Desktop

3 minutes to read • [Edit Online](#)

With **Power Query** and **Power BI Desktop**, you can add **Query Parameters** to a report and make elements of the report dependent on those parameters. For example, you could use Query Parameters to automatically have a report create a filter, load a data model or a data source reference, generate a measure definition, and many other abilities. Query Parameters let users open a report, and by providing values for its Query Parameters, jump-start creating that report with just a few clicks.

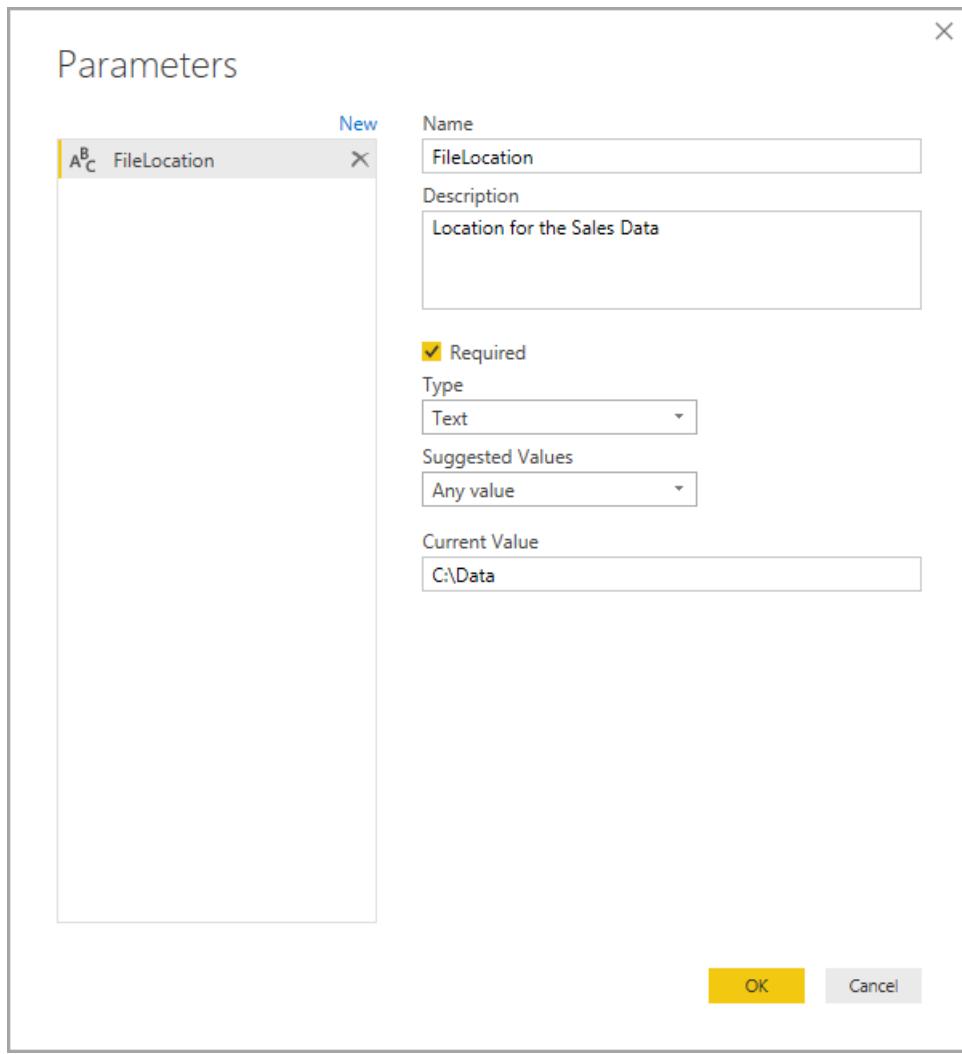


You can have one parameter, or multiple parameters for any report. Let's take a look at how to create parameters in Power BI Desktop.

Creating Query Parameters

To create or access Query Parameters in a Power BI Desktop report, go to the **Home** ribbon and select **Edit Queries > Edit Parameters** to bring up the **Power Query Editor** window.

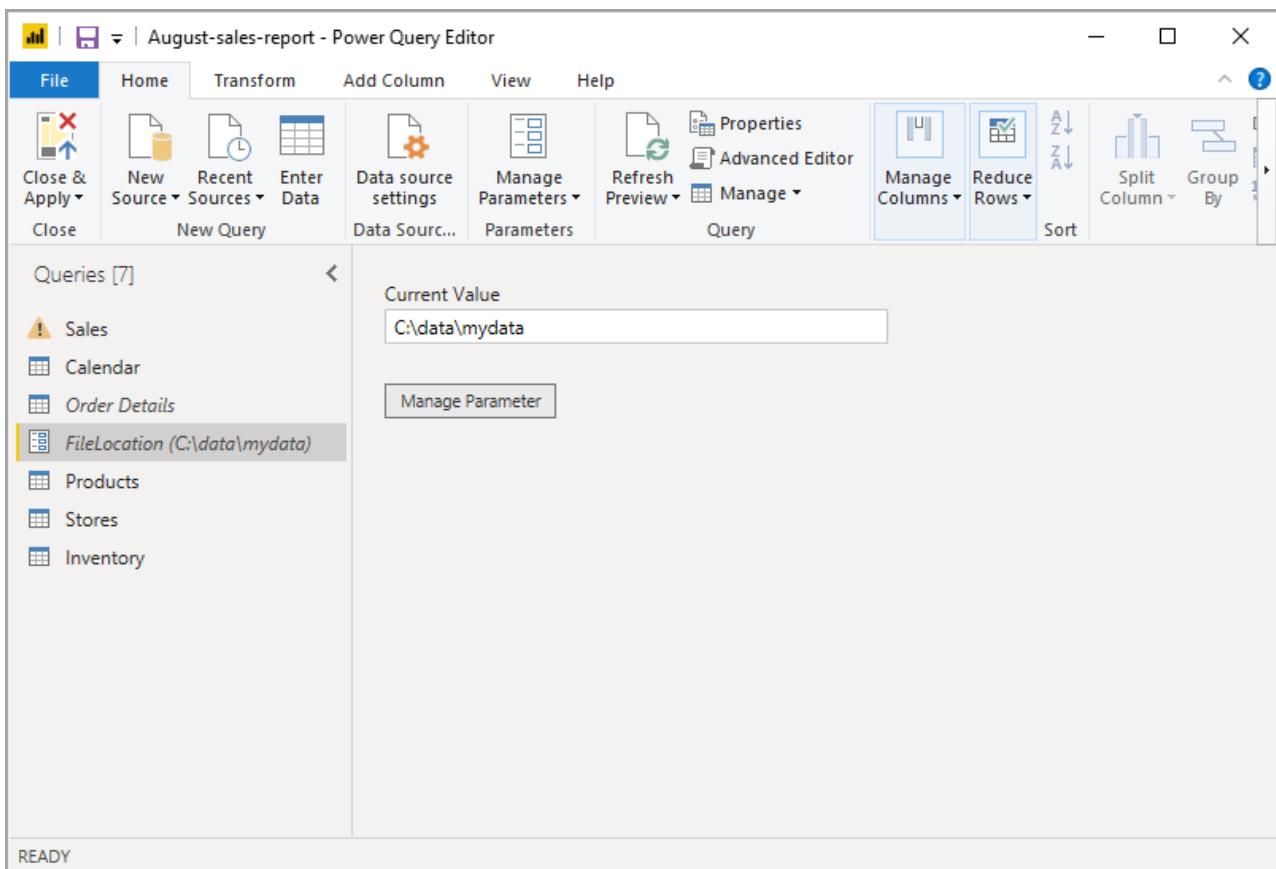
In the **Power Query Editor** window, from the **Home** ribbon select **Manage Parameters** to bring up the **Parameters** dialog.



The **Parameters** dialog has the following elements that let you create new parameters, or specify metadata and setting for each parameter:

- **Name**—Provide a name for this parameter that lets you easily recognize and differentiate it from other parameters you might create.
- **Description**—The description is displayed next to the parameter name when parameter information is displayed, helping users who are specifying the parameter value to understand its purpose, and its semantics.
- **Required**—The checkbox indicates whether subsequent users can specify whether a value for the parameter must be provided.
- **Type**—Applies a Data Type restriction to the input value for the parameter. For example, users can define a parameter of type *Text*, or *Date/Time*. If you want to provide the greatest flexibility for users, you can specify *Any value* from the list of available types in the drop-down.
- **Suggested Values**—You can further restrict the entries that users can select or provide for a given parameter. For example, you could specify that the Data Type for a parameter is *Text*, and then restrict the acceptable values for that parameter to a static list of *Text* values. Users then can pick one of the available values when specifying the parameter value.
- **Default Value**—Sets the default value for a parameter.
- **Current Value**—Specifies the parameter's value for the current report.

Once you define these values for a parameter and select **OK**, a new query is defined for each parameter that was created, shown in the **Power Query Editor** dialog.

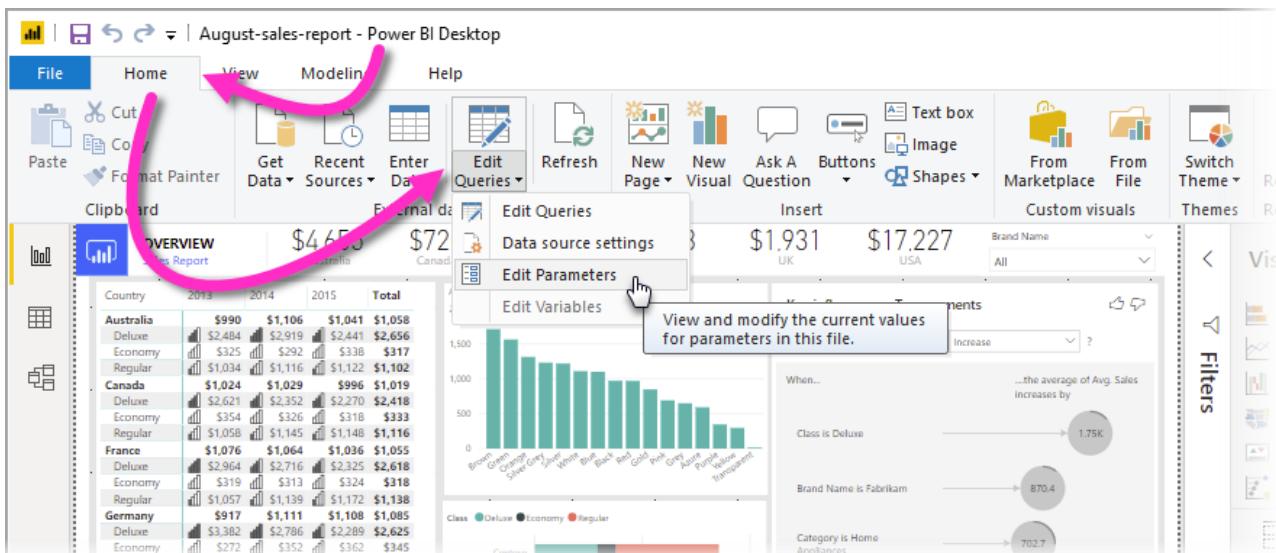


Using Query Parameters

Once you've defined Query Parameters, you can specify how they're being referenced or used by other queries. For reports that have parameters, users accessing or using the report are prompted for parameter input with one or more dialog boxes.

Once information about the parameter value has been provided, select **Close & Apply** in the **Power Query Editor** ribbon to have the data loaded into the data model. Once the selection or data for parameters have been provided, you can reference the parameters from DAX expressions within the report, or in any other way you might reference a parameter value.

If you want to change the value for a Query Parameter, you can do that within the report by selecting **Edit Parameters** from the **Edit Queries** button, found on the **Home** ribbon of **Power BI Desktop**.



Selecting **Edit Parameters** brings up a window that allows you to provide a different value for the parameter. Providing a different value and then selecting **OK** refreshes the report data, and any visuals, based on the new

parameter values.



Query Parameter values are currently available only in **Power BI Desktop**.

Next steps

There are all sorts of things you can do with Power Query and Power BI Desktop. For more information, check out the following resources:

- [Query Overview with Power BI Desktop](#)
- [Data Types in Power BI Desktop](#)
- [Shape and Combine Data with Power BI Desktop](#)
- [Common Query Tasks in Power BI Desktop](#)
- [Using templates in Power BI Desktop](#)

Tutorial: Shape and combine data using Power Query

15 minutes to read • [Edit Online](#)

With **Power Query**, you can connect to many different types of data sources, then shape the data to meet your needs, enabling you to create visual reports using **Power BI Desktop** that you can share with others. *Shaping* data means transforming the data—such as renaming columns or tables, changing text to numbers, removing rows, setting the first row as headers, and so on. *Combining* data means connecting to two or more data sources, shaping them as needed, and then consolidating them into one useful query.

In this tutorial, you'll learn to:

- Shape data using **Power Query Editor**
- Connect to a data source
- Connect to another data source
- Combine those data sources, and create a data model to use in reports

This tutorial demonstrates how to shape a query using **Power Query Editor**, technology that's incorporated into **Power BI Desktop**, and learn some common data tasks.

It's useful to know that the **Power Query Editor** in Power BI Desktop makes ample use of right-click menus, as well as the ribbon. Most of what you can select in the **Transform** ribbon is also available by right-clicking an item (such as a column) and choosing from the menu that appears.

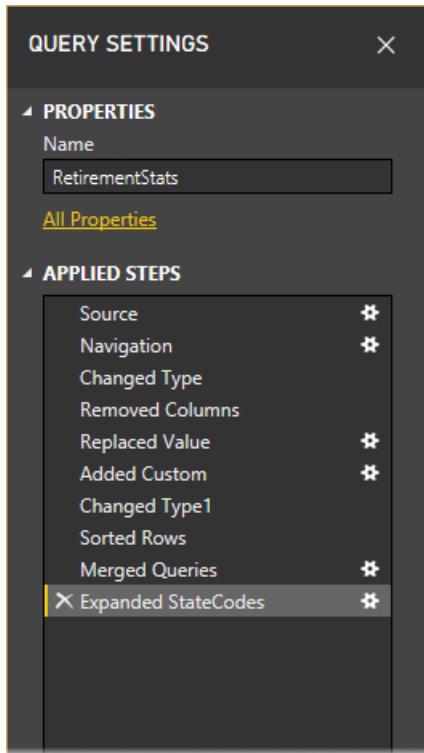
If you're not signed up for Power BI, you can [sign up for a free trial](#) before you begin. Also, Power BI Desktop is [free to download](#).

Shape data

When you shape data in the **Power Query Editor**, you're providing step-by-step instructions (that Power Query Editor carries out for you) to adjust the data as Power Query Editor loads and presents it. The original data source is not affected; only this particular view of the data is adjusted, or *shaped*.

The steps you specify (such as rename a table, transform a data type, or delete columns) are recorded by Power Query Editor. Each time this query connects to the data source, those steps are carried out so that the data is always shaped the way you specify. This process occurs whenever you use the Power Query Editor feature of Power BI Desktop, or for anyone who uses your shared query, such as on the **Power BI** service. Those steps are captured, sequentially, in the **Query Settings** pane, under **Applied Steps**.

The following image shows the **Query Settings** pane for a query that has been shaped—you'll go through each of those steps in the next few paragraphs.



Using the retirement data from the [Using Power Query in Power BI Desktop](#) quickstart article, which you found by connecting to a Web data source, you can shape that data to fit your needs.

For starters, you can add a custom column to calculate rank based on all data being equal factors, and compare this to the existing column *Rank*. Here's the **Add Column** ribbon, with an arrow pointing toward the **Custom Column** button, which lets you add a custom column.

In the **Custom Column** dialog, in **New column name**, enter **New Rank**, and in **Custom column formula**, enter the following:

```
([Cost of living] + [Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8
```

Make sure the status message reads 'No syntax errors have been detected.' and select **OK**.

Custom Column

New column name

New Rank

Custom column formula:

```
= ([Cost of living] + [Weather] + [Health care quality] +  
[Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8
```

Available columns:

Cost of living

Weather

Health care quality

Crime

Tax

Culture

Senior

Well-being

<< Insert

[Learn about Power BI Desktop formulas](#)

✓ No syntax errors have been detected.

OK

Cancel

To keep column data consistent, you can transform the new column values to whole numbers. Just right-click the column header, and select **Change Type > Whole Number** to change them.

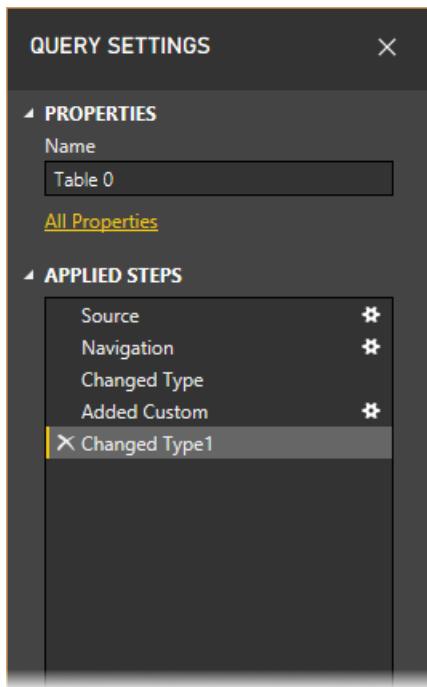
If you need to choose more than one column, first select a column then hold down **SHIFT**, select additional adjacent columns, and then right-click a column header to change all selected columns. You can also use the **CTRL** key to choose non-adjacent columns.

The screenshot shows a Power Query Editor window with a table containing columns for 'Tax', 'Culture', 'Senior', 'Well-being', and 'New Rank'. A context menu is open over the 'Well-being' column, with the 'Change Type' option selected. Other options visible in the menu include Copy, Remove, Remove Other Columns, Duplicate Column, Add Column From Examples..., Remove Duplicates, Remove Errors, Transform, Replace Values..., Replace Errors..., Group By..., Fill, Unpivot Columns, Unpivot Other Columns, Unpivot Only Selected Columns, Rename..., Move, Drill Down, and Add as New Query.

You can also *transform* column data types from the **Transform** ribbon. Here's the **Transform** ribbon, with an arrow pointing toward the **Data Type** button, which lets you transform the current data type to another.

The screenshot shows the Power Query Editor ribbon with the 'Transform' tab selected. A pink arrow points to the 'Data Type' button, which is currently set to 'Text'. Other buttons in the ribbon include File, Home, Transform, Add Column, View, Transpose, Reverse Rows, Detect Data Type, Rename, Split Column, Format, Parse, Statistics, Trigonometry, Standard, Rounding, Scientific, Information, Date, Time, Duration, and Structured Column.

Note that in **Query Settings**, the **Applied Steps** reflect any shaping steps applied to the data. If you want to remove any step from the shaping process, you simply select the **X** to the left of the step. In the following image, **Applied Steps** reflects the steps so far, which includes connecting to the website (**Source**), selecting the table (**Navigation**), and, while loading the table, Power Query Editor automatically changing text-based number columns from **Text** to **Whole Number (Changed Type)**. The last two steps show your previous actions with **Added Custom** and **Changed Type1**.



Before you can work with this query, you need to make a few changes to get its data where you want it:

- *Adjust the rankings by removing a column*—you've decided **Cost of living** is a non-factor in your results. After removing this column, you find the issue that the data remains unchanged, though it's easy to fix using Power BI Desktop, and doing so demonstrates a cool feature of **Applied Steps** in Query.
- *Fix a few errors*—since you removed a column, you need to readjust your calculations in the **New Rank** column. This involves changing a formula.
- *Sort the data*—based on the **New Rank** and **Rank** columns.
- *Replace data*—this tutorial will highlight how to replace a specific value and the need of inserting an **Applied Step**.
- *Change the table name*—**Table 0** is not a useful descriptor, but changing it is simple.

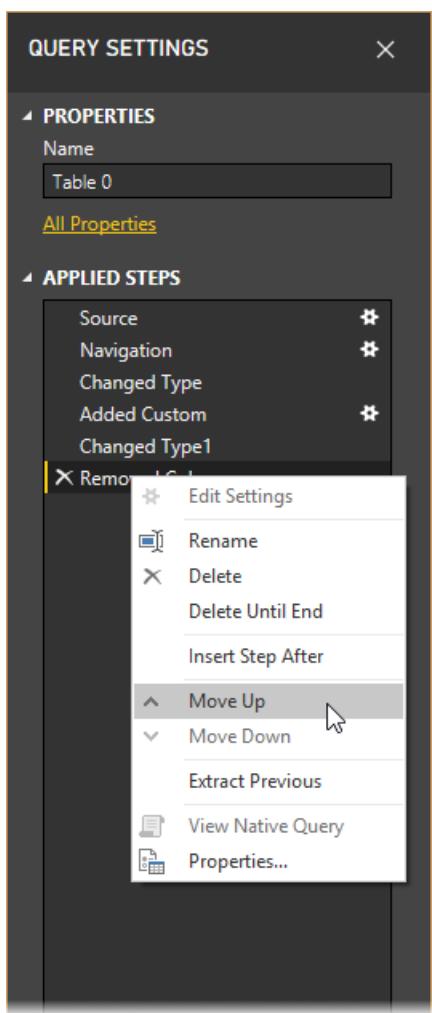
To remove the **Cost of living** column, simply select the column and choose the **Home** tab from the ribbon, and then **Remove Columns**, as shown in the following figure.

Rank	State	Health care quality	Crime
1	New Hampshire	40	45
2	Colorado	33	20
3	Maine	38	44
4	Iowa	14	36
5	Minnesota	30	47
6	Virginia	31	16
7	Massachusetts	45	29
8	South Dakota	26	40
9	Wisconsin	24	42
10	Idaho	5	31
11	Utah	16	19

Notice the *New Rank* values have not changed; this is due to the ordering of the steps. Since Power Query Editor

records the steps sequentially, yet independently of each other, you can move each **Applied Step** up or down in the sequence. Just right-click any step and Power Query Editor provides a menu that lets you do the following:

Rename, Delete, Delete Until End (remove the current step, and all subsequent steps too), **Move Up**, or **Move Down**. Go ahead and move up the last step *Removed Columns* to just above the *Added Custom* step.



Next, select the *Added Custom* step. Notice the data now shows *Error*, which you'll need to address.

The screenshot shows the Power Query Editor interface. On the left is a table with columns: 'culture', 'Senior', 'Well-being', and 'New Rank'. The 'New Rank' column contains numerous 'Error' values. On the right is the 'QUERY SETTINGS' pane, which includes sections for 'PROPERTIES' (Name: Table 0) and 'APPLIED STEPS'. The 'APPLIED STEPS' section lists steps like 'Source', 'Navigation', 'Changed Type', 'Removed Columns', and 'Added Custom' (which has a 'Changed Type1' step). A pink arrow points from the error cells in the table to the 'Added Custom' step in the 'Applied Steps' list.

There are a few ways to get more information about each error. You can select the cell (without selecting the word **Error**), or select the word **Error** directly. If you select the cell *without* selecting the word **Error** directly, Power Query Editor displays the error information on the bottom of the window.

The screenshot shows the Power Query Editor with an error message at the bottom: 'Expression.Error: The field 'Cost of living' of the record wasn't found.' Below this message, there is a 'Details:' section with the following information: Rank=1, State=New Hampshire, Weather=45, and Health care quality=4. To the right is the 'Query Settings' pane, specifically the 'APPLIED STEPS' section, which includes steps like 'Source', 'Navigation', 'Changed Type', 'Removed Columns', and 'Added Custom' (with a 'Changed Type1' step). A pink arrow points from the error message at the bottom to the 'Added Custom' step in the 'Applied Steps' list.

If you select the word **Error** directly, Query creates an **Applied Step** in the **Query Settings** pane and displays information about the error. You don't want to go this route, so select **Cancel**.

To fix the errors, select the *New Rank* column, then display the column's data formula by opening the **View** ribbon and selecting the **Formula Bar** checkbox.

The screenshot shows the Power Query Editor interface. At the top, there's a ribbon with tabs like File, Home, Transform, Add Column, View, and Help. Under the View tab, there are several options: 'Formula Bar' (which is checked), 'Monospaced' (unchecked), 'Show whitespace' (checked), 'Always allow' (unchecked), 'Advanced Editor' (button), and 'Query Dependencies' (button). Below the ribbon is a 'Queries [1]' pane on the left containing 'Table 0'. To the right is a 'Data Preview' pane showing a table with columns: Crime, Tax, Culture, Senior, Well-being, and New Rank. The 'New Rank' column contains error values (e.g., Error, 2, 3, 11, etc.). A pink arrow points from the 'Formula Bar' option in the ribbon down to the formula bar in the preview pane.

Now you can remove the *Cost of living* parameter and decrement the divisor by changing the formula to the following:

```
Table.AddColumn(#"Removed Columns", "New Rank", each ([Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 7)
```

Select the green checkmark to the left of the formula box or press **Enter**, and the data should be replaced by revised values. The **Added Custom** step should now complete *with no errors*.

NOTE

You can also **Remove Errors** (using the ribbon or the right-click menu), which removes any rows that have errors. In this case it would've removed all the rows from your data, and you don't want to do that—you probably like your data, and want to keep it in the table.

Now you need to sort the data based on the **New Rank** column. First select the last applied step, **Changed Type1**, to get to the most recent data. Then, select the drop-down located next to the **New Rank** column header and select **Sort Ascending**.

The screenshot shows the Power Query Editor interface. On the left, there's a preview of a table with columns: Culture, Senior, Well-being, and New Rank. The 'New Rank' column is currently selected. A context menu is open over this column, with 'Sort Ascending' option highlighted. To the right of the preview, the 'QUERY SETTINGS' pane is open, showing the table is named 'Table 0' and has an applied step 'Changed Type1'. At the bottom, the formula bar displays the formula: = Table.Sort(#"Changed Type1",{{"New Rank", Order.Ascending}, {"Rank", Order.Ascending}}).

Notice the data is now sorted according to **New Rank**. However, if you look in the **Rank** column, you'll notice the data is not sorted properly in cases where the **New Rank** value is a tie. To fix this, select the **New Rank** column and change the formula in the **Formula Bar** to the following:

```
= Table.Sort(#"Changed Type1",{{"New Rank", Order.Ascending}, {"Rank", Order.Ascending}})
```

Select the green checkmark to the left of the formula box or press **Enter**, and the rows should now be ordered in accordance with both *New Rank* and *Rank*.

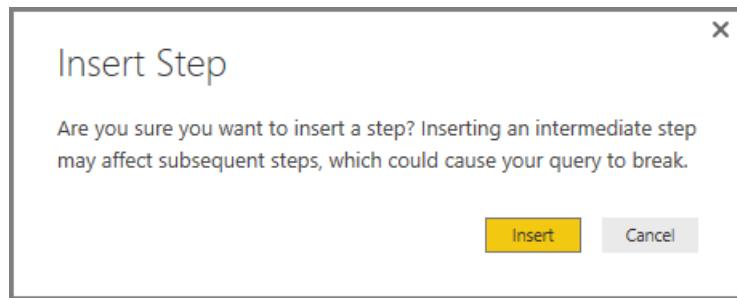
In addition, you can select an **Applied Step** anywhere in the list, and continue shaping the data at that point in the sequence. Power Query Editor will automatically insert a new step directly after the currently selected **Applied Step**. Let's give that a try.

First, select the **Applied Step** prior to adding the custom column—this would be the *Removed Columns* step. Here you'll replace the value of the *Weather* ranking in Arizona. Right-click the appropriate cell that contains Arizona's *Weather* ranking and select **Replace Values** from the menu that appears. Note which **Applied Step** is currently selected (the step prior to the *Added Custom* step).

A screenshot of the Power Query Editor interface. A context menu is open over the 12th row of a table. The table has columns: Rank, State, Weather, Health care quality, Crime, and Tax. The 12th row contains the value '2' in the Rank column. The context menu options are: Copy, Number Filters, Replace Values..., Drill Down, and Add as New Query. The 'Replace Values...' option is highlighted with a cursor.

	Rank	State	Weather	Health care quality	Crime	Tax
1	1	New Hampshire		45	4	3
2	2	Colorado		20	7	26
3	3	Maine		44	1	2
4	4	Iowa		36	9	15
5	5	Minnesota		47	3	14
6	6	Virginia		16	16	5
7	7	Massachusetts		29	5	17
8	8	South Dakota		40	18	20
9	9	Wisconsin		42	2	16
10	10	Idaho		31	20	4
11	11	Utah		19	8	24
12	12	Arizona		2	26	38
13	13	Nebraska			17	18
14	14	Vermont			15	1
15	15	Pennsylvania			14	12
16	16	North Dakota			22	11
17	17	Florida			30	39
18	18	Delaware			6	40
19	19	Rhode Island		26	10	9
20	20	North Carolina		11	12	29
21	21	Wyoming		41	44	8
22	22	Michigan		43	19	22

Since you're inserting a step, Power Query Editor warns you about the danger of doing so—subsequent steps could cause the query to break. You need to be careful, and thoughtful! Since this is a tutorial that's highlighting a really cool feature of Power Query Editor to demonstrate how you can create, delete, insert, and reorder steps, go ahead and select **Insert**.



Change the value to 51 and the data for Arizona is replaced. When you create a new Applied Step, Power Query Editor names it based on the action—in this case, **Replaced Value**. When you have more than one step with the same name in your query, Power Query Editor adds a number (in sequence) to each subsequent **Applied Step** to differentiate between them.

Now select the last **Applied Step**, *Sorted Rows*, and notice the data has changed regarding Arizona's new ranking. This is because you inserted the *Replaced Value* step in the right place, before the *Added Custom* step.

That was a little involved, but it was a good example of how powerful and versatile Power Query Editor can be.

Lastly, you'll want to change the name of that table to something descriptive. When you get to creating reports, it's especially useful to have descriptive table names, especially when you connect to multiple data sources, and they're all listed in the **Fields** pane of the **Report** view.

Changing the table name is easy. In the **Query Settings** pane, under **Properties**, simply type in the new name of the table, as shown in the following image, and select **Enter**. Call this table *RetirementStats*.

The screenshot shows the Microsoft Power Query Editor interface. On the left, there's a preview pane displaying a table with two columns: 'Well-being' and 'New Rank'. The 'Well-being' column contains values like 2, 11, 3, 9, 1, 22, 10, 17, 21, 16, 14, 29, 15, and 13. The 'New Rank' column contains values like 12, 16, 18, 18, 18, 19, 19, 21, 21, 21, 21, 21, 21, 21, and 22. Above the preview, the ribbon has tabs for 'Transform' and 'Combine'. To the right of the preview is a 'QUERY SETTINGS' dialog box. The 'Name' field is set to 'RetirementStats', which is highlighted with a pink arrow. The 'APPLIED STEPS' section lists several steps: Source, Navigation, Changed Type, Removed Columns, Replaced Value, Added Custom, Changed Type1, and Sorted Rows.

You've shaped that data to the extent you need to. Next, you'll connect to another data source and combine data.

Combine data

The data about various states is interesting, and will be useful for building additional analysis efforts and queries. But there's one problem: most data out there uses a two-letter abbreviation for state codes, not the full name of the state. You need some way to associate state names with their abbreviations.

You're in luck. There's another public data source that does just that, but it needs a fair amount of shaping before you can connect it to your retirement table. Here's the Web resource for state abbreviations:

https://en.wikipedia.org/wiki/List_of_U.S._state_abbreviations

From the **Home** ribbon in Power Query Editor, select **New Source > Web** and type the address, select **Connect**, and the Navigator shows what it found on that Web page.

Navigator

The screenshot shows the Microsoft Power BI Navigator dialog box. On the left, there's a search bar with a magnifying glass icon and a dropdown menu labeled "Display Options". Below the search bar is a list of items from a Wikipedia page, with the first item, "Codes and abbreviations for U.S. states, territories and oth...", checked. To the right of the search bar is a "Table View" tab, which is selected, and a "Web View" tab. The main area displays a table titled "Codes and abbreviations for U.S. states, territories and other entities". The table has four columns: "Name and status of region", "Name and status of region2", "ISO", and "ISO". The data includes rows for the United States of America, Alabama, Alaska, Arizona, Arkansas, California, Colorado, Connecticut, Delaware, District of Columbia, Florida, Georgia, Hawaii, Idaho, Illinois, Indiana, Iowa, Kansas, Kentucky, and Louisiana. At the bottom right of the dialog box are "OK" and "Cancel" buttons.

Name and status of region	Name and status of region2	ISO
United States of America	Federal state	US USA 840
Alabama	State	US-AL
Alaska	State	US-AK
Arizona	State	US-AZ
Arkansas	State	US-AR
California	State	US-CA
Colorado	State	US-CO
Connecticut	State	US-CT
Delaware	State	US-DE
District of Columbia	Federal district	US-DC
Florida	State	US-FL
Georgia	State	US-GA
Hawaii	State	US-HI
Idaho	State	US-ID
Illinois	State	US-IL
Indiana	State	US-IN
Iowa	State	US-IA
Kansas	State	US-KS
Kentucky	State (Commonwealth)	US-KY
Louisiana	State	US-LA

Select **Codes and abbreviations...** because that includes the data you want, but it's going to take quite a bit of shaping to pare that table's data down to what you want.

TIP

Is there a faster or easier way to accomplish the steps below? Yes, you could create a *relationship* between the two tables, and shape the data based on that relationship. The following steps are still good to learn for working with tables; just know that relationships can help you quickly use data from multiple tables.

To get this data into shape, take the following steps:

1. Remove the top row—it's a result of the way that Web page's table was created, and you don't need it. From the **Home** ribbon, select **Reduce Rows > Remove Rows > Remove Top Rows**.

The screenshot shows the Power BI Editor interface with the 'Transform' ribbon tab selected. A context menu is open over a table named 'Region Status'. The 'Remove Rows' option is selected, and a submenu is displayed with the 'Remove Top Rows' option highlighted. A tooltip for 'Remove Top Rows' says 'Remove the top N rows from this table.' The 'APPLIED STEPS' pane on the right shows the step 'Changed Type'.

The **Remove Top Rows** window appears, letting you specify how many rows you want to remove.

NOTE

If Power BI accidentally imports the table headers as a row in your data table, you can select **Use First Row As Headers** from the **Home** tab, or from the **Transform** tab in the ribbon, to fix your table.

2. Remove the bottom 26 rows—they’re all the territories, which you don’t need to include. From the **Home** ribbon, select **Reduce Rows > Remove Rows > Remove Bottom Rows**.

The screenshot shows the Power BI Editor interface with the 'Transform' ribbon tab selected. A context menu is open over a table named 'Status'. The 'Remove Rows' option is selected, and a submenu is displayed with the 'Remove Bottom Rows' option highlighted. A tooltip for 'Remove Bottom Rows' says 'Remove the bottom N rows from this table.' The 'APPLIED STEPS' pane on the right shows the step 'Removed Top Rows'.

3. Since the RetirementStats table doesn't have information for Washington DC, you need to filter it from your list. Select the drop-down arrow beside the Region Status column, then clear the checkbox beside **Federal district**.

2 Queries

Region Name Region Status Codes ISO

	Region Name	Region Status	Codes ISO
16	Indiana	State	US-IN IN
17	Iowa	State	US-IA IA
18	Kansas	State	US-KS KS
19	Kentucky	State (Commonwealth)	US-KY KY

- Remove a few unneeded columns—you only need the mapping of the state to its official two-letter abbreviation, so you can remove the following columns: **Column1**, **Column3**, **Column4**, and then **Column6** through **Column11**. First select **Column1**, then hold down the **CTRL** key and select the other columns to be removed (this lets you select multiple, non-contiguous columns). From the Home tab on the ribbon, select **Remove Columns > Remove Columns**.

Choose Columns Remove Columns Reduce Rows Data Type: Text Use First Row As Headers

Manage Remove Columns Remove Other Columns

Remove the currently selected columns from this table.

Query Settings

PROPERTIES

Name
Table[edit]

NOTE

This is a good time to point out that the sequence of applied steps in Power Query Editor is important, and can affect how the data is shaped. It's also important to consider how one step may impact another subsequent step. If you remove a step from the Applied Steps, subsequent steps may not behave as originally intended because of the impact of the query's sequence of steps.

NOTE

When you resize the Power Query Editor window to make the width smaller, some ribbon items are condensed to make the best use of visible space. When you increase the width of the Power Query Editor window, the ribbon items expand to make the most use of the increased ribbon area.

5. Rename the columns, and the table itself—as usual, there are a few ways to rename a column. First select the column, then either select **Rename** from the **Transform** tab on the ribbon, or right-click and select **Rename...** from the menu that appears. The following image has arrows pointing to both options; you only need to choose one.

The screenshot shows the Power Query Editor interface. The ribbon is at the top with tabs: File, Home, Transform (selected), Add Column, View. Under the Transform tab, there are several icons: Transpose, Reverse Rows, Detect Data Type, Count Rows, Rename, Split Column, Format, Merge Columns, Extract, Parse, and Statistics. Below the ribbon is a list of '2 Queries': RetirementStats and Table[edit]. The 'Table[edit]' query is currently selected and displayed in the main pane. It contains a table titled 'United States' with two columns: 'State Name' and 'State Code'. The table has 22 rows, listing all US states. A context menu is open over the first row of the table, with 'Rename...' highlighted. The status bar at the bottom shows '2 COLUMNS, 77 ROWS'.

Rename the columns to *State Name* and *State Code*. To rename the table, just type the name into the **Name** box in the **Query Settings** pane. Call this table *StateCodes*.

Now that you've shaped the StateCodes table the way you want, you'll now combine these two tables, or queries, into one. Since the tables you now have are a result of the queries you applied to the data, they're often referred to as *queries*.

There are two primary ways of combining queries: *merging* and *appending*.

When you have one or more columns that you'd like to add to another query, you **merge** the queries. When you have additional rows of data that you'd like to add to an existing query, you **append** the query.

In this case, you'll want to merge queries. To get started, from the left pane of Power Query Editor, select the query *into which* you want the other query to merge, which in this case is *RetirementStats*. Then select **Combine > Merge Queries** from the **Home** tab on the ribbon.

A screenshot of the Power BI Data Editor interface. On the left, there's a table with columns 'Crime rate' and 'Community well-being'. The 'Combine' button in the top right has a dropdown menu open, with 'Merge Queries' highlighted by a pink arrow. The 'Query Settings' pane on the right shows the query name 'RetirementStats' and applied steps.

You may be prompted to set the privacy levels to ensure the data is combined without including or transferring data you didn't want transferred.

Next the **Merge** window appears, prompting you to select which table you'd like merged into the selected table, and then, the matching columns to use for the merge. Select State from the *RetirementStats* table (query), then select the *StateCodes* query (easy in this case, since there's only one other query—when you connect to many data sources, there are many queries to choose from). When you select the correct matching columns—**State** from *RetirementStats*, and **State Name** from *StateCodes*—the **Merge** window looks like the following, and the **OK** button is enabled.

The Merge dialog box is open. It shows two tables: 'RetirementStats' and 'StateCodes'. The 'RetirementStats' table has columns for Rank, State, Weather, Health care quality, Crime, Tax, Culture, Senior, Well-being, and Net. The 'StateCodes' table has columns for Name and status of region and ANSI, with data for Alabama (AL), Alaska (AK), Arizona (AZ), Arkansas (AR), and California (CA). Below the tables, the 'Join Kind' is set to 'Left Outer (all from first, matching from second)'. A status message at the bottom says 'The selection has matched 50 out of the first 50 rows.' with a green checkmark icon. There are 'OK' and 'Cancel' buttons at the bottom right.

A **NewColumn** is created at the end of the query, which is the contents of the table (query) that was merged with the existing query. All columns from the merged query are condensed into the **NewColumn**, but you can select to **Expand** the table, and include whichever columns you want.

The screenshot shows the Power Query Editor interface. The ribbon at the top has the 'Transform' tab selected. Below the ribbon is a table with several columns. A pink arrow points from the 'NewColumn' dropdown menu to the 'Expand' icon in the ribbon.

To expand the merged table, and select which columns to include, select the expand icon (). The **Expand** window appears.

The screenshot shows the Power Query Editor with the 'Expand' dialog box open. The 'Expand' tab is selected. The 'Search Columns to Expand' field contains 'State'. The 'Expand' radio button is selected. Under 'Select Columns', 'State Name' and 'State Code' are checked. The 'OK' button is highlighted.

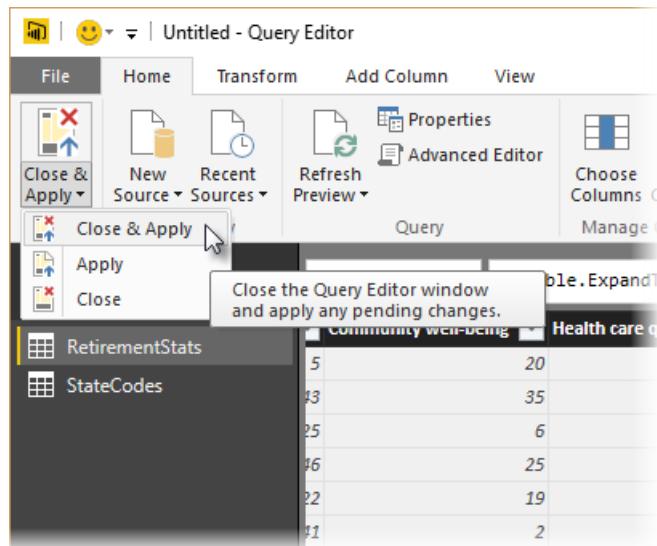
In this case, you only want the **State Code** column, so you'll select only that column, and then select **OK**. Clear the checkbox from *Use original column name as prefix* because you don't need or want that. If you leave that checkbox selected, the merged column would be named **NewColumn.State Code** (the original column name, or **NewColumn**, then a dot, then the name of the column being brought into the query).

NOTE

Want to play around with how to bring in that **NewColumn** table? You can experiment a bit, and if you don't like the results, just delete that step from the **Applied Steps** list in the **Query Settings** pane. Your query returns to the state prior to applying that **Expand** step. It's like a free do-over, which you can do as many times as you like until the expand process looks the way you want it.

You now have a single query (table) that combined two data sources, each of which has been shaped to meet your needs. This query can serve as a basis for lots of additional, interesting data connections, such as housing cost statistics, demographics, or job opportunities in any state.

To apply changes and close Power Query Editor, select **Close & Apply** from the **Home** ribbon tab. The transformed dataset appears in Power BI Desktop, ready to be used for creating reports.



Next steps

There are all sorts of things you can do with Power Query. If you're ready to create your own custom connector, check the following article.

[Creating your first connector: Hello World](#)

Installing the Power Query SDK

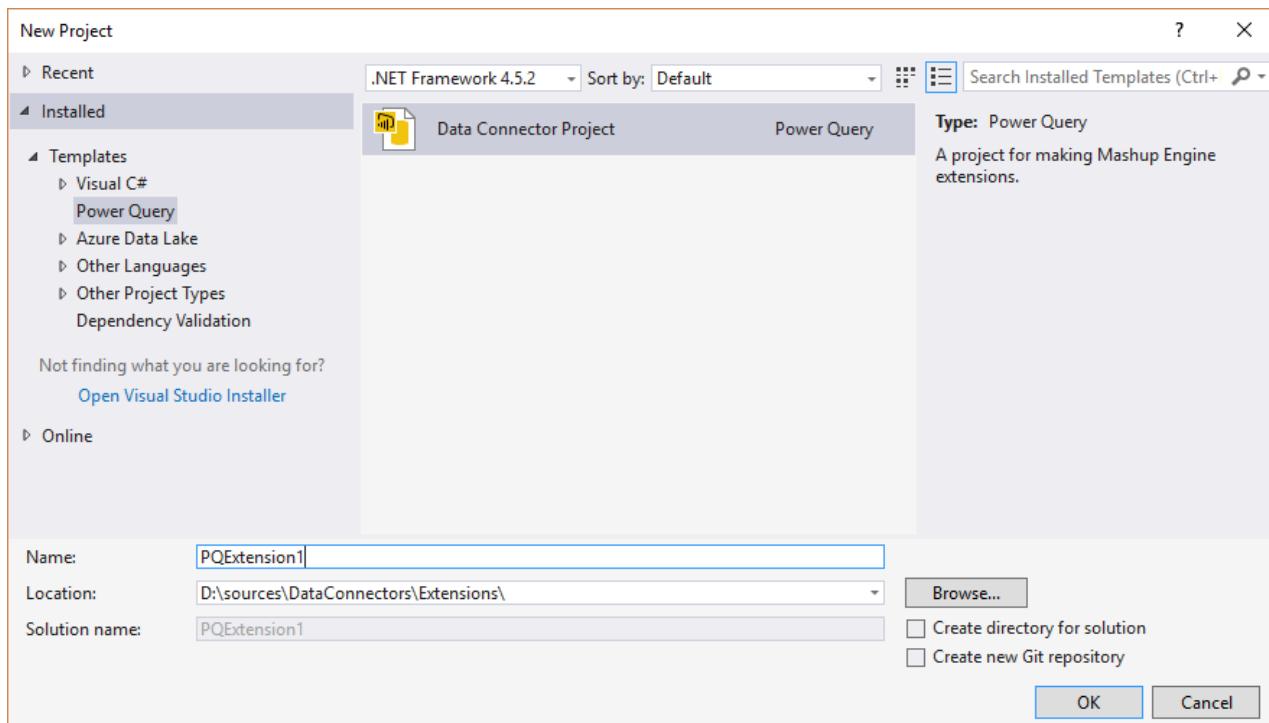
3 minutes to read • [Edit Online](#)

Quickstart

NOTE

The steps to enable extensions changed in the June 2017 version of Power BI Desktop.

1. Install the [Power Query SDK](#) from the Visual Studio Marketplace.
2. Create a new data connector project.
3. Define your connector logic.
4. Build the project to produce an extension file.
5. Copy the extension file into [Documents]/Power BI Desktop/Custom Connectors.
6. Check the option **(Not Recommended) Allow any extension to load without validation or warning** in Power BI Desktop (under *File | Options and settings | Options | Security | Data Extensions*).
7. Restart Power BI Desktop.



Distribution of Data Connectors

Power BI Desktop users can download extension files and place them in a known directory (steps described above). Power BI Desktop will automatically load the extensions on restart. *We are hard at work on Office Store integration to make it easy for users to discover and install data connectors you build. During this preview phase, developers interested in distributing their connectors for use with Power BI can contact us at DataConnectors (at microsoft.com).*

Additional links and resources

- [M Library Functions](#)

- [M Language Specification](#)
- [Power BI Developer Center](#)
- [Data Connector Tutorial](#)

Step by step

Creating a new extension in Visual Studio

Installing the Power Query SDK for Visual Studio will create a new Data Connector project template in Visual Studio.

This creates a new project containing the following files:

- Connector definition file (.pq)
- A query test file (.query.pq)
- A string resource file (resources.resx)
- PNG files of various sizes used to create icons

Your connector definition file will start with an empty Data Source description. See the Data Source Kind section later in this document for details.

Testing in Visual Studio

The Power Query SDK provides basic query execution capabilities, allowing you to test your extension without having to switch over to Power BI Desktop. See [Query File](#) for more details.

Build and deploy from Visual Studio

Building your project will produce your .pqi file.

Data Connector projects don't support custom post build steps to copy the extension file to your [Documents]\Microsoft Power BI Desktop\Custom Connectors directory. If this is something you want to do, you may want to use a third party Visual Studio extension, such as Auto Deploy.

Extension files

Power Query extensions are bundled in a ZIP file and given a .mez file extension. At runtime, Power BI Desktop will load extensions from the [Documents]\Microsoft Power BI Desktop\Custom Connectors.

NOTE

In an upcoming change the default extension will be changed from .mez to .pqi.

Extension file format

Extensions are defined within an M section document. A section document has a slightly different format from the query document(s) generated in Power Query. Code you import from Power Query typically requires modification to fit into a section document, but the changes are minor. Section document differences you should be aware of include:

- They begin with a section declaration (for example, `section HelloWorld;`).
- Each expression ends with a semi-colon (for example, `a = 1;` or `b = let c = 1 + 2 in c;`).
- All functions and variables are local to the section document, unless they are marked as shared. Shared functions become visible to other queries/functions, and can be thought of as the exports for your extension (that is, they become callable from Power Query).

More information about M section documents can be found in the M Language specification.

Query File

In addition to the extension file, Data Connector projects can have a query file (name.query.pq). This file can be used to run test queries within Visual Studio. The query evaluation will automatically include your extension code, without having to register your .pxq file, allowing you to call/test any shared functions in your extension code.

The query file can contain a single expression (for example, `HelloWorld.Contents()`), a `let` expression (such as what Power Query would generate), or a section document.

Starting to Develop Custom Connectors

2 minutes to read • [Edit Online](#)

To get you up to speed with Power Query, this page lists some of the most common questions.

What software do I need to get started with the Power Query SDK?

You need to install the [Power Query SDK](#) in addition to Visual Studio. To be able to test your connectors, you should also have Power BI installed.

What can you do with a Connector?

Data Connectors allow you to create new data sources or customize and extend an existing source. Common use cases include:

- Creating a business analyst-friendly view for a REST API.
- Providing branding for a source that Power Query supports with an existing connector (such as an OData service or ODBC driver).
- Implementing OAuth v2 authentication flow for a SaaS offering.
- Exposing a limited or filtered view over your data source to improve usability.
- Enabling DirectQuery for a data source using an ODBC driver.

Data Connectors are currently only supported in Power BI Desktop.

Creating your first connector: Hello World

2 minutes to read • [Edit Online](#)

Hello World sample

This sample provides a simple data source extension that can be run in Visual Studio, and loaded in Power BI Desktop. As an overview, this sample shows the following:

- Exporting function (`HelloWorld.Contents`), which takes an option text parameter.
- Defining a data source kind that:
 - Declares that it uses Implicit (anonymous) authentication.
 - Uses string resources that allow for localization.
 - Declaring UI metadata so the extension can show up in the Power BI Desktop Get Data dialog.

Following the instructions in [Installing the PowerQuery SDK](#), create a new project called "HelloWorld" and copy in the following M code, and then follow the rest of the instructions to be able to open it in PowerBI.

In the following connector definition you will find:

- A `section` statement.
- A data source function with metadata establishing it as a data source definition with the Kind `HelloWorld` and Publish `HelloWorld.Publish`.
- An `Authentication` record declaring that implicit (anonymous) is the only authentication type for this source.
- A publish record declaring that this connection is in Beta, what text to load from the resx file, the source image, and the source type image.
- A record associating icon sizes with specific pngs in the build folder.

```

[DataSource.Kind="HelloWorld", Publish="HelloWorld.Publish"]
shared HelloWorld.Contents = (optional message as text) =>
    let
        message = if (message <> null) then message else "Hello world"
    in
        message;

HelloWorld = [
    Authentication = [
        Implicit = []
    ],
    Label = Extension.LoadString("DataSourceLabel")
];

HelloWorld.Publish = [
    Beta = true,
    ButtonText = { Extension.LoadString("FormulaTitle"), Extension.LoadString("FormulaHelp") },
    SourceImage = HelloWorld(Icons,
    SourceTypeImage = HelloWorld(Icons
];

HelloWorld(Icons = [
    Icon16 = { Extension.Contents("HelloWorld16.png"), Extension.Contents("HelloWorld20.png"), Extension.Contents("HelloWorld24.png"), Extension.Contents("HelloWorld32.png") },
    Icon32 = { Extension.Contents("HelloWorld32.png"), Extension.Contents("HelloWorld40.png"), Extension.Contents("HelloWorld48.png"), Extension.Contents("HelloWorld64.png") }
];

```

Once you've built the file and copied it to the correct directory, following the instructions in [Installing the PowerQuery SDK](#) tutorial, open PowerBI. You can search for "hello" to find your connector in the **Get Data** dialog.

This step will bring up an authentication dialog. Since there's no authentication options and the function takes no parameters, there's no further steps in these dialogs.

Press **Connect** and the dialog will tell you that it's a "Preview connector", since `Beta` is set to true in the query. Since there's no authentication, the authentication screen will present a tab for Anonymous authentication with no fields. Press **Connect** again to finish.

Finally, the query editor will come up showing what you expect—a function that returns the text "Hello world".

For the fully implemented sample, see the [Hello World Sample](#) in the Data Connectors sample repo.

Tutorial: Shape and combine data using Power Query

15 minutes to read • [Edit Online](#)

With **Power Query**, you can connect to many different types of data sources, then shape the data to meet your needs, enabling you to create visual reports using **Power BI Desktop** that you can share with others. *Shaping* data means transforming the data—such as renaming columns or tables, changing text to numbers, removing rows, setting the first row as headers, and so on. *Combining* data means connecting to two or more data sources, shaping them as needed, and then consolidating them into one useful query.

In this tutorial, you'll learn to:

- Shape data using **Power Query Editor**
- Connect to a data source
- Connect to another data source
- Combine those data sources, and create a data model to use in reports

This tutorial demonstrates how to shape a query using **Power Query Editor**, technology that's incorporated into **Power BI Desktop**, and learn some common data tasks.

It's useful to know that the **Power Query Editor** in Power BI Desktop makes ample use of right-click menus, as well as the ribbon. Most of what you can select in the **Transform** ribbon is also available by right-clicking an item (such as a column) and choosing from the menu that appears.

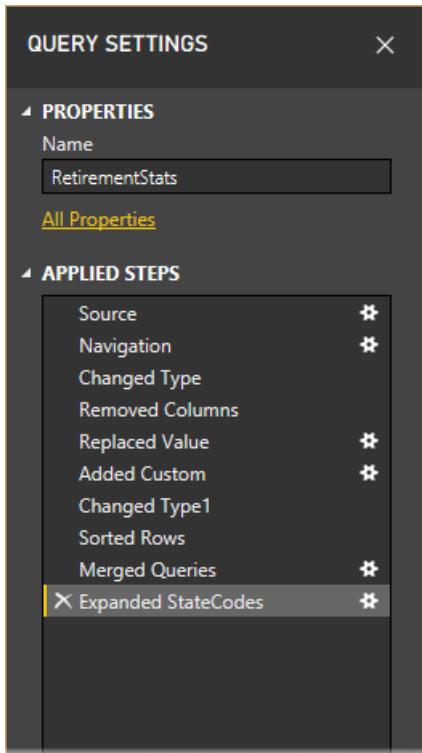
If you're not signed up for Power BI, you can [sign up for a free trial](#) before you begin. Also, Power BI Desktop is [free to download](#).

Shape data

When you shape data in the **Power Query Editor**, you're providing step-by-step instructions (that Power Query Editor carries out for you) to adjust the data as Power Query Editor loads and presents it. The original data source is not affected; only this particular view of the data is adjusted, or *shaped*.

The steps you specify (such as rename a table, transform a data type, or delete columns) are recorded by Power Query Editor. Each time this query connects to the data source, those steps are carried out so that the data is always shaped the way you specify. This process occurs whenever you use the Power Query Editor feature of Power BI Desktop, or for anyone who uses your shared query, such as on the **Power BI** service. Those steps are captured, sequentially, in the **Query Settings** pane, under **Applied Steps**.

The following image shows the **Query Settings** pane for a query that has been shaped—you'll go through each of those steps in the next few paragraphs.



Using the retirement data from the [Using Power Query in Power BI Desktop](#) quickstart article, which you found by connecting to a Web data source, you can shape that data to fit your needs.

For starters, you can add a custom column to calculate rank based on all data being equal factors, and compare this to the existing column *Rank*. Here's the **Add Column** ribbon, with an arrow pointing toward the **Custom Column** button, which lets you add a custom column.

The screenshot shows the 'Untitled - Power Query Editor' window. The ribbon is visible with tabs like File, Home, Transform, Add Column, View, and Help. A pink arrow points to the 'Custom Column' button in the 'Add Column' group. The main area shows a table with columns: Health care quality, Crime, Tax, Culture, Senior, and Well-being. The formula bar shows the M code: `= Table.TransformColumnTypes(Data0,{{"Rank", Int64.Type}, {"State", type text}})`. The 'Queries [1]' pane shows 'Table 0' and the 'Custom Column' dialog, which is open with the message 'Create a new column in this table, based on a custom formula.' The formula input field contains the M code: `([Cost of living] + [Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8`.

In the **Custom Column** dialog, in **New column name**, enter **New Rank**, and in **Custom column formula**, enter the following:

```
([Cost of living] + [Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8
```

Make sure the status message reads 'No syntax errors have been detected.' and select **OK**.

Custom Column

New column name
New Rank

Custom column formula:
`= ([Cost of living] + [Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8`

Available columns:
Cost of living
Weather
Health care quality
Crime
Tax
Culture
Senior
Well-being

<< Insert

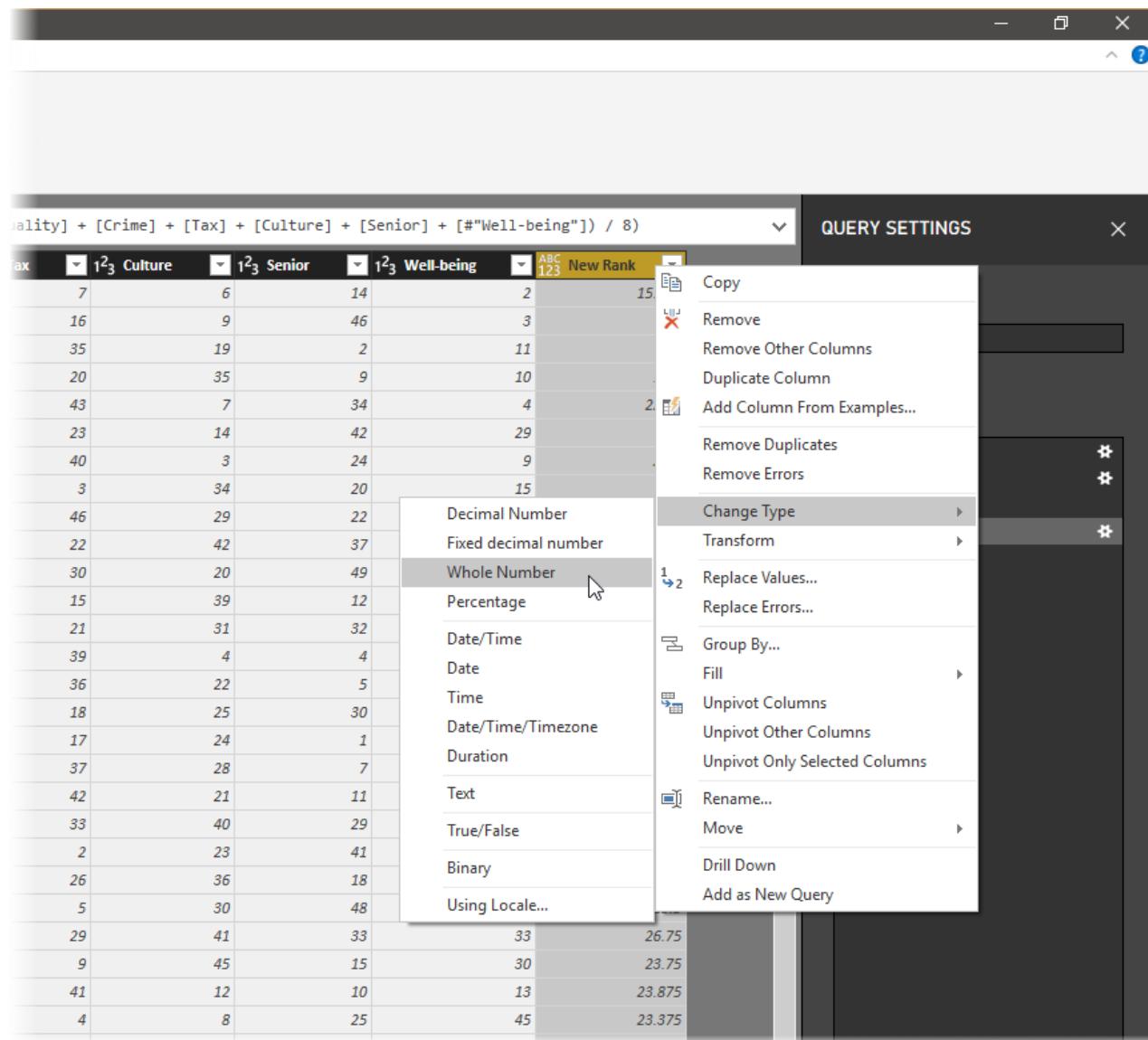
[Learn about Power BI Desktop formulas](#)

✓ No syntax errors have been detected.

OK Cancel

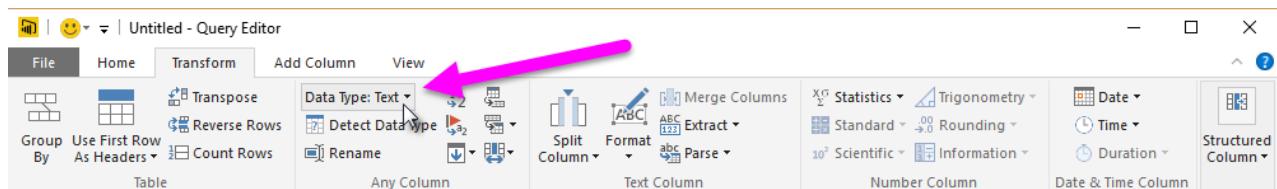
To keep column data consistent, you can transform the new column values to whole numbers. Just right-click the column header, and select **Change Type > Whole Number** to change them.

If you need to choose more than one column, first select a column then hold down **SHIFT**, select additional adjacent columns, and then right-click a column header to change all selected columns. You can also use the **CTRL** key to choose non-adjacent columns.

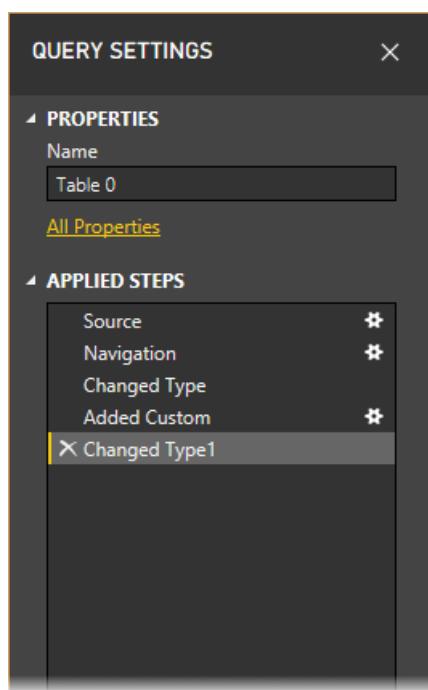


The screenshot shows a Power BI desktop interface with a data grid containing columns: Cost of living, Weather, Health care quality, Crime, Tax, Culture, Senior, and Well-being. A new column, "New Rank", is being created with the formula `= ([Cost of living] + [Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 8`. A context menu is open over the "New Rank" column header, with the "Whole Number" option highlighted under the "Change Type" submenu. Other options in the submenu include Decimal Number, Fixed decimal number, Percentage, Date/Time, Date, Time, Date/Time/Timezone, Duration, Text, True/False, Binary, and Using Locale... The "QUERY SETTINGS" tab is visible at the top right of the ribbon.

You can also *transform* column data types from the **Transform** ribbon. Here's the **Transform** ribbon, with an arrow pointing toward the **Data Type** button, which lets you transform the current data type to another.



Note that in **Query Settings**, the **Applied Steps** reflect any shaping steps applied to the data. If you want to remove any step from the shaping process, you simply select the **X** to the left of the step. In the following image, **Applied Steps** reflects the steps so far, which includes connecting to the website (**Source**), selecting the table (**Navigation**), and, while loading the table, Power Query Editor automatically changing text-based number columns from *Text* to *Whole Number* (**Changed Type**). The last two steps show your previous actions with **Added Custom** and **Changed Type1**.



Before you can work with this query, you need to make a few changes to get its data where you want it:

- *Adjust the rankings by removing a column*—you've decided **Cost of living** is a non-factor in your results. After removing this column, you find the issue that the data remains unchanged, though it's easy to fix using Power BI Desktop, and doing so demonstrates a cool feature of **Applied Steps** in Query.
- *Fix a few errors*—since you removed a column, you need to readjust your calculations in the **New Rank** column. This involves changing a formula.
- *Sort the data*—based on the **New Rank** and **Rank** columns.
- *Replace data*—this tutorial will highlight how to replace a specific value and the need of inserting an **Applied Step**.
- *Change the table name*—**Table 0** is not a useful descriptor, but changing it is simple.

To remove the **Cost of living** column, simply select the column and choose the **Home** tab from the ribbon, and then **Remove Columns**, as shown in the following figure.

The screenshot shows the Power Query Editor interface. In the ribbon, the 'Transform' tab is selected. A pink arrow points to the 'Remove Columns' button in the 'Manage Columns' group. The 'QUERY SETTING' pane on the right shows the 'APPLIED STEPS' section with 'Removed Columns' highlighted.

Notice the *New Rank* values have not changed; this is due to the ordering of the steps. Since Power Query Editor records the steps sequentially, yet independently of each other, you can move each **Applied Step** up or down in the sequence. Just right-click any step and Power Query Editor provides a menu that lets you do the following:

Rename, Delete, Delete Until End (remove the current step, and all subsequent steps too), **Move Up**, or **Move Down**. Go ahead and move up the last step *Removed Columns* to just above the *Added Custom* step.

The screenshot shows the 'QUERY SETTINGS' pane. In the 'APPLIED STEPS' list, 'Removed Columns' is selected. A context menu is open, with 'Move Up' highlighted.

Next, select the *Added Custom* step. Notice the data now shows *Error*, which you'll need to address.

The screenshot shows the Power Query Editor interface. On the left is a table with columns: Culture, Senior, Well-being, and New Rank. The New Rank column contains numerous 'Error' values. On the right is the 'QUERY SETTINGS' pane, which includes sections for 'PROPERTIES' (Name: Table 0) and 'APPLIED STEPS'. The 'APPLIED STEPS' section lists several steps, with 'Added Custom' being the most recent, which corresponds to the error in the New Rank column.

There are a few ways to get more information about each error. You can select the cell (without selecting the word **Error**), or select the word **Error** directly. If you select the cell *without* selecting the word **Error** directly, Power Query Editor displays the error information on the bottom of the window.

The screenshot shows the Power Query Editor interface with an error message displayed at the bottom left: 'Expression.Error: The field 'Cost of living' of the record wasn't found.' Below this message, there are details: Rank=1, State=New Hampshire, Weather=45, and Health care quality=4. On the right, the 'QUERY SETTINGS' pane shows the 'APPLIED STEPS' section, which includes 'Added Custom' and 'Changed Type1'.

If you select the word *Error* directly, Query creates an **Applied Step** in the **Query Settings** pane and displays information about the error. You don't want to go this route, so select **Cancel**.

To fix the errors, select the *New Rank* column, then display the column's data formula by opening the **View** ribbon and selecting the **Formula Bar** checkbox.

The screenshot shows the Power Query Editor interface. At the top, there's a ribbon with tabs like File, Home, Transform, Add Column, View, and Help. Under the View tab, there are several options: 'Formula Bar' (which is checked), 'Monospaced' (unchecked), 'Show whitespace' (checked), 'Always allow' (unchecked), 'Go to Column' (button), 'Columns' (button), 'Parameters' (button), 'Advanced' (button), 'Query Dependencies' (button). Below the ribbon is a 'Queries [1]' pane on the left containing 'Table 0'. On the right is a 'Data Preview' pane showing a table with columns: Crime, Tax, Culture, Senior, Well-being, and New Rank. The 'New Rank' column contains values from 2 to 29, each followed by an 'Error' message. A pink arrow points from the 'Query Settings' button in the ribbon to the 'Formula Bar' checkbox. Another pink arrow points from the 'fx' button in the formula bar to the formula itself.

Now you can remove the *Cost of living* parameter and decrement the divisor by changing the formula to the following:

```
Table.AddColumn(#"Removed Columns", "New Rank", each ([Weather] + [Health care quality] + [Crime] + [Tax] + [Culture] + [Senior] + [#"Well-being"]) / 7)
```

Select the green checkmark to the left of the formula box or press **Enter**, and the data should be replaced by revised values. The **Added Custom** step should now complete *with no errors*.

NOTE

You can also **Remove Errors** (using the ribbon or the right-click menu), which removes any rows that have errors. In this case it would've removed all the rows from your data, and you don't want to do that—you probably like your data, and want to keep it in the table.

Now you need to sort the data based on the **New Rank** column. First select the last applied step, **Changed Type1**, to get to the most recent data. Then, select the drop-down located next to the **New Rank** column header and select **Sort Ascending**.

The screenshot shows the Power Query Editor interface. On the left, there's a preview of a table with columns: Culture, Senior, Well-being, and New Rank. The 'New Rank' column is currently selected, indicated by a yellow background. In the center, a context menu is open over the 'New Rank' column, with options like 'Sort Ascending', 'Sort Descending', 'Clear Sort', 'Clear Filter', 'Remove Empty', and 'Number Filters'. On the right, the 'QUERY SETTINGS' pane is open, showing the properties of the current step, which is named 'Table 0'. Under 'APPLIED STEPS', the step 'Changed Type1' is listed and selected.

Notice the data is now sorted according to **New Rank**. However, if you look in the **Rank** column, you'll notice the data is not sorted properly in cases where the **New Rank** value is a tie. To fix this, select the **New Rank** column and change the formula in the **Formula Bar** to the following:

```
= Table.Sort#"Changed Type1",{{"New Rank", Order.Ascending}, {"Rank", Order.Ascending}}
```

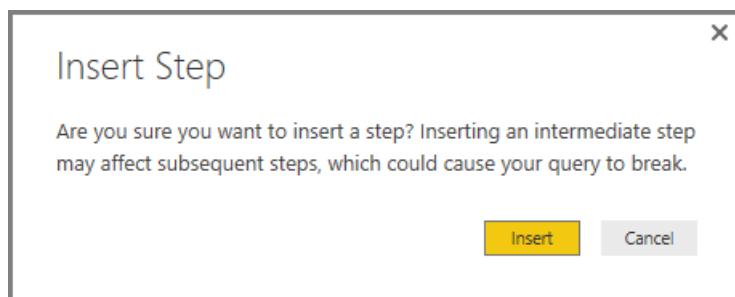
Select the green checkmark to the left of the formula box or press **Enter**, and the rows should now be ordered in accordance with both *New Rank* and *Rank*.

In addition, you can select an **Applied Step** anywhere in the list, and continue shaping the data at that point in the sequence. Power Query Editor will automatically insert a new step directly after the currently selected **Applied Step**. Let's give that a try.

First, select the **Applied Step** prior to adding the custom column—this would be the *Removed Columns* step. Here you'll replace the value of the *Weather* ranking in Arizona. Right-click the appropriate cell that contains Arizona's *Weather* ranking and select **Replace Values** from the menu that appears. Note which **Applied Step** is currently selected (the step prior to the *Added Custom* step).

	Rank	State	Weather	Health care quality	Crime	Tax
1	1	New Hampshire		45	4	3
2	2	Colorado		20	7	26
3	3	Maine		44	1	2
4	4	Iowa		36	9	15
5	5	Minnesota		47	3	14
6	6	Virginia		16	16	5
7	7	Massachusetts		29	5	17
8	8	South Dakota		40	18	20
9	9	Wisconsin		42	2	16
10	10	Idaho		31	20	4
11	11	Utah		19	8	24
12	12	Arizona		2	26	38
13	13	Nebraska			17	18
14	14	Vermont			15	1
15	15	Pennsylvania			14	12
16	16	North Dakota			22	11
17	17	Florida			30	39
18	18	Delaware			6	40
19	19	Rhode Island		26	10	9
20	20	North Carolina		11	12	29
21	21	Wyoming		41	44	8
22	22	Michigan		43	19	22

Since you're inserting a step, Power Query Editor warns you about the danger of doing so—subsequent steps could cause the query to break. You need to be careful, and thoughtful! Since this is a tutorial that's highlighting a really cool feature of Power Query Editor to demonstrate how you can create, delete, insert, and reorder steps, go ahead and select **Insert**.



Change the value to 51 and the data for Arizona is replaced. When you create a new Applied Step, Power Query Editor names it based on the action—in this case, **Replaced Value**. When you have more than one step with the same name in your query, Power Query Editor adds a number (in sequence) to each subsequent **Applied Step** to differentiate between them.

Now select the last **Applied Step**, *Sorted Rows*, and notice the data has changed regarding Arizona's new ranking. This is because you inserted the *Replaced Value* step in the right place, before the *Added Custom* step.

That was a little involved, but it was a good example of how powerful and versatile Power Query Editor can be.

Lastly, you'll want to change the name of that table to something descriptive. When you get to creating reports, it's especially useful to have descriptive table names, especially when you connect to multiple data sources, and they're all listed in the **Fields** pane of the **Report** view.

Changing the table name is easy. In the **Query Settings** pane, under **Properties**, simply type in the new name of the table, as shown in the following image, and select **Enter**. Call this table *RetirementStats*.

The screenshot shows the Power Query Editor interface. On the left, there's a table with two columns: 'Well-being' and 'New Rank'. The rows contain data like (2, 12), (11, 16), (3, 18), etc. On the right, a 'QUERY SETTINGS' dialog is open. It has sections for 'PROPERTIES' (with 'Name' set to 'RetirementStats') and 'APPLIED STEPS' (listing steps like Source, Navigation, Changed Type, etc.). A pink arrow points to the 'Name' input field in the properties section.

You've shaped that data to the extent you need to. Next, you'll connect to another data source and combine data.

Combine data

The data about various states is interesting, and will be useful for building additional analysis efforts and queries. But there's one problem: most data out there uses a two-letter abbreviation for state codes, not the full name of the state. You need some way to associate state names with their abbreviations.

You're in luck. There's another public data source that does just that, but it needs a fair amount of shaping before you can connect it to your retirement table. Here's the Web resource for state abbreviations:

https://en.wikipedia.org/wiki/List_of_U.S._state_abbreviations

From the **Home** ribbon in Power Query Editor, select **New Source > Web** and type the address, select **Connect**, and the Navigator shows what it found on that Web page.

Navigator

The screenshot shows the Microsoft Power BI Navigator window. At the top, there's a search bar and a 'Display Options' dropdown. Below that is a tree view of a Wikipedia page titled 'List of U.S. state abbreviations'. The 'Codes and abbreviations for U.S. states, territories and oth...' node is selected. The main content area displays a table with four columns: 'Name and status of region', 'Name and status of region2', and 'ISO'. The table lists various US states with their corresponding abbreviations and ISO codes. At the bottom right are 'OK' and 'Cancel' buttons.

	Name and status of region	Name and status of region2	ISO
iso	United States of America	Federal state	US USA 840
iso	Alabama	State	US-AL
iso	Alaska	State	US-AK
iso	Arizona	State	US-AZ
iso	Arkansas	State	US-AR
iso	California	State	US-CA
iso	Colorado	State	US-CO
iso	Connecticut	State	US-CT
iso	Delaware	State	US-DE
iso	District of Columbia	Federal district	US-DC
iso	Florida	State	US-FL
iso	Georgia	State	US-GA
iso	Hawaii	State	US-HI
iso	Idaho	State	US-ID
iso	Illinois	State	US-IL
iso	Indiana	State	US-IN
iso	Iowa	State	US-IA
iso	Kansas	State	US-KS
iso	Kentucky	State (Commonwealth)	US-KY
iso	Louisiana	State	US-LA

Select **Codes and abbreviations...** because that includes the data you want, but it's going to take quite a bit of shaping to pare that table's data down to what you want.

TIP

Is there a faster or easier way to accomplish the steps below? Yes, you could create a *relationship* between the two tables, and shape the data based on that relationship. The following steps are still good to learn for working with tables; just know that relationships can help you quickly use data from multiple tables.

To get this data into shape, take the following steps:

1. Remove the top row—it's a result of the way that Web page's table was created, and you don't need it. From the **Home** ribbon, select **Reduce Rows > Remove Rows > Remove Top Rows**.

The screenshot shows the Power BI Editor interface with the Transform ribbon tab selected. A table named "Region Status" is open. A context menu is open over the table, with "Remove Rows" selected. A sub-menu is open under "Remove Rows" with "Remove Top Rows" highlighted. A tooltip explains "Remove the top N rows from this table." The "Query Settings" pane on the right shows the table name as "Table[edit]".

The **Remove Top Rows** window appears, letting you specify how many rows you want to remove.

NOTE

If Power BI accidentally imports the table headers as a row in your data table, you can select **Use First Row As Headers** from the **Home** tab, or from the **Transform** tab in the ribbon, to fix your table.

2. Remove the bottom 26 rows—they’re all the territories, which you don’t need to include. From the **Home** ribbon, select **Reduce Rows > Remove Rows > Remove Bottom Rows**.

The screenshot shows the Power BI Editor interface with the Transform ribbon tab selected. A table named "Status" is open. A context menu is open over the table, with "Remove Rows" selected. A sub-menu is open under "Remove Rows" with "Remove Bottom Rows" highlighted. A tooltip explains "Remove the bottom N rows from this table." The "Query Settings" pane on the right shows the table name as "Table[edit]".

3. Since the RetirementStats table doesn’t have information for Washington DC, you need to filter it from your list. Select the drop-down arrow beside the Region Status column, then clear the checkbox beside **Federal district**.

The screenshot shows the Power Query Editor interface with a '2 Queries' ribbon tab. On the left, there are two queries: 'RetirementStats' and 'StateCodes'. The 'StateCodes' query is currently selected. A pink arrow points to the filter dialog for the 'Region Status' column. The dialog lists several filter options: '(Select All)', 'Federal district' (which is currently unchecked), 'Federal state' (checked), 'State' (checked), and 'State (Commonwealth)' (checked). The 'OK' button at the bottom right of the dialog is highlighted.

- Remove a few unneeded columns—you only need the mapping of the state to its official two-letter abbreviation, so you can remove the following columns: **Column1**, **Column3**, **Column4**, and then **Column6** through **Column11**. First select **Column1**, then hold down the **CTRL** key and select the other columns to be removed (this lets you select multiple, non-contiguous columns). From the Home tab on the ribbon, select **Remove Columns > Remove Columns**.

The screenshot shows the Power Query Editor ribbon with the 'Transform' tab selected. The 'Remove Columns' button is highlighted with a mouse cursor. A tooltip above the button reads 'Remove the currently selected columns from this table.' To the right, a 'Query Settings' dialog is open, showing the 'Name' field set to 'Table[edit]'. The main workspace below shows a table with four columns: 'Codes ANSI2', 'Codes USPS', 'Codes USCG', and 'Abbreviations Old'. The first three columns contain data like '00', 'AL', 'AK', etc., while the fourth column contains abbreviations like 'U.S.', 'Ala.', 'Alaska', 'Ariz.'

NOTE

This is a good time to point out that the sequence of applied steps in Power Query Editor is important, and can affect how the data is shaped. It's also important to consider how one step may impact another subsequent step. If you remove a step from the Applied Steps, subsequent steps may not behave as originally intended because of the impact of the query's sequence of steps.

NOTE

When you resize the Power Query Editor window to make the width smaller, some ribbon items are condensed to make the best use of visible space. When you increase the width of the Power Query Editor window, the ribbon items expand to make the most use of the increased ribbon area.

5. Rename the columns, and the table itself—as usual, there are a few ways to rename a column. First select the column, then either select **Rename** from the **Transform** tab on the ribbon, or right-click and select **Rename...** from the menu that appears. The following image has arrows pointing to both options; you only need to choose one.

The screenshot shows the Power Query Editor interface. The ribbon is at the top with tabs like File, Home, Transform, Add Column, View, etc. The Transform tab is currently selected. Below the ribbon, there's a toolbar with various icons for operations like Transpose, Detect Data Type, and Merge Columns. A pink arrow points from the 'Rename' icon in this toolbar to a context menu. This context menu is open over a column labeled 'United States' in a table. The menu items include Copy, Remove, Remove Other Columns, Duplicate Column, Remove Duplicates, Remove Errors, Change Type, Transform, Replace Values..., Replace Errors..., Split Column, Group By..., Fill, Unpivot Columns, Unpivot Other Columns, Rename... (which is highlighted with a cursor), Move, Drill Down, and Add as New Query. Another pink arrow points from the 'Rename...' item in the menu to the 'Rename...' item in the ribbon's Transform tab.

Row	State	Code
1	Alabama	AL
2	Alaska	AK
3	Arizona	AZ
4	Arkansas	AR
5	California	CA
6	Colorado	CO
7	Connecticut	CT
8	Delaware	DE
9	District of Columbia	DC
10	Florida	FL
11	Georgia	GA
12	Hawaii	HI
13	Idaho	ID
14	Illinois	IL
15	Indiana	IN
16	Iowa	IA
17	Kansas	KS
18	Kentucky	KY
19	Louisiana	LA
20	Maine	ME
21	Maryland	MD
22	Massachusetts	MA

Rename the columns to *State Name* and *State Code*. To rename the table, just type the name into the **Name** box in the **Query Settings** pane. Call this table *StateCodes*.

Now that you've shaped the StateCodes table the way you want, you'll now combine these two tables, or queries, into one. Since the tables you now have are a result of the queries you applied to the data, they're often referred to as *queries*.

There are two primary ways of combining queries: *merging* and *appending*.

When you have one or more columns that you'd like to add to another query, you **merge** the queries. When you have additional rows of data that you'd like to add to an existing query, you **append** the query.

In this case, you'll want to merge queries. To get started, from the left pane of Power Query Editor, select the query *into which* you want the other query to merge, which in this case is *RetirementStats*. Then select **Combine > Merge Queries** from the **Home** tab on the ribbon.

The screenshot shows the Power BI Query Editor interface. In the top ribbon, under the 'Transform' tab, there is a 'Combine' button. A pink arrow points from the text 'Merge Queries' to this button. Below the ribbon, a context menu is open over a table named 'RetirementStats'. The menu has sections for 'PROPERTIES' (with 'Name' set to 'RetirementStats') and 'APPLIED STEPS' (with 'Source' selected). The 'Merge Queries' option is listed under the 'Combine' section of the menu.

You may be prompted to set the privacy levels to ensure the data is combined without including or transferring data you didn't want transferred.

Next the **Merge** window appears, prompting you to select which table you'd like merged into the selected table, and then, the matching columns to use for the merge. Select State from the *RetirementStats* table (query), then select the *StateCodes* query (easy in this case, since there's only one other query—when you connect to many data sources, there are many queries to choose from). When you select the correct matching columns—**State** from *RetirementStats*, and **State Name** from *StateCodes*—the **Merge** window looks like the following, and the **OK** button is enabled.

The screenshot shows the 'Merge' dialog box. At the top, it says 'Merge' and 'Select a table and matching columns to create a merged table.' Below this, there are two tables: 'RetirementStats' and 'StateCodes'. The 'RetirementStats' table has columns: Rank, State, Weather, Health care quality, Crime, Tax, Culture, Senior, Well-being, and Ne. The 'StateCodes' table has columns: Name and status of region and ANSI. Under 'Join Kind', it says 'Left Outer (all from first, matching from second)'. At the bottom, a message says 'The selection has matched 50 out of the first 50 rows.' with a checkmark icon. There are 'OK' and 'Cancel' buttons at the bottom right.

A **NewColumn** is created at the end of the query, which is the contents of the table (query) that was merged with the existing query. All columns from the merged query are condensed into the **NewColumn**, but you can select to **Expand** the table, and include whichever columns you want.

The screenshot shows the Power Query Editor interface. The ribbon at the top has a 'Transform' tab selected. Below the ribbon is a merged table with four columns: 'e quality', 'Tax rate', 'Weather', and 'NewColumn'. The 'NewColumn' column contains the value 'Table'. To the right of the table is a 'Query Settings' pane. Under the 'PROPERTIES' section, the 'Name' is set to 'RetirementStats'. Under the 'APPLIED STEPS' section, there are steps for 'Source', 'Navigation', 'Changed Type', and 'Removed Columns'.

To expand the merged table, and select which columns to include, select the expand icon (). The **Expand** window appears.

The screenshot shows the Power Query Editor with the 'Expand' dialog box open. The dialog box has a radio button for 'Expand' (which is selected) and another for 'Aggregate'. Below that is a list with three checked checkboxes: '(Select All Columns)', 'State Name', and 'State Code'. At the bottom of the dialog box is a checkbox for 'Use original column name as prefix'. The 'OK' button is highlighted in yellow. To the right of the dialog box is the 'Query Settings' pane, which is identical to the one in the first screenshot.

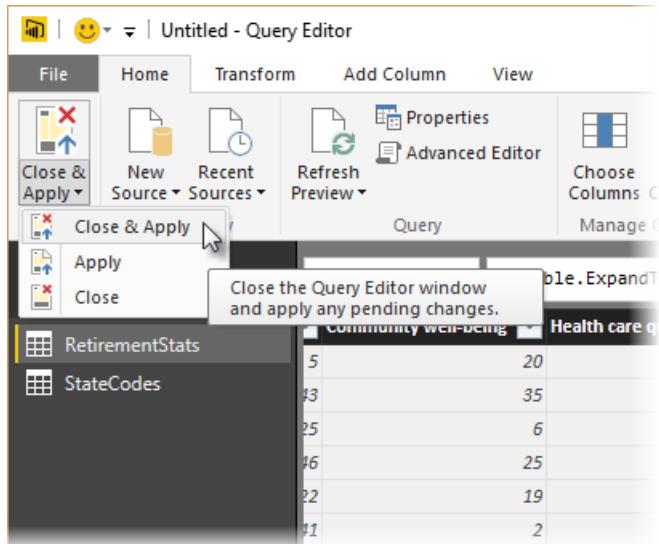
In this case, you only want the **State Code** column, so you'll select only that column, and then select **OK**. Clear the checkbox from *Use original column name as prefix* because you don't need or want that. If you leave that checkbox selected, the merged column would be named **NewColumn.State Code** (the original column name, or **NewColumn**, then a dot, then the name of the column being brought into the query).

NOTE

Want to play around with how to bring in that **NewColumn** table? You can experiment a bit, and if you don't like the results, just delete that step from the **Applied Steps** list in the **Query Settings** pane. Your query returns to the state prior to applying that **Expand** step. It's like a free do-over, which you can do as many times as you like until the expand process looks the way you want it.

You now have a single query (table) that combined two data sources, each of which has been shaped to meet your needs. This query can serve as a basis for lots of additional, interesting data connections, such as housing cost statistics, demographics, or job opportunities in any state.

To apply changes and close Power Query Editor, select **Close & Apply** from the **Home** ribbon tab. The transformed dataset appears in Power BI Desktop, ready to be used for creating reports.



Next steps

There are all sorts of things you can do with Power Query. If you're ready to create your own custom connector, check the following article.

[Creating your first connector: Hello World](#)

TripPin Tutorial

2 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

This tutorial uses a public OData service ([TripPin](#)) as a reference source. Although this lesson requires the use of the M engine's OData functions, subsequent lessons will use [Web.Contents](#), making it applicable to (most) REST APIs.

Prerequisites

The following applications will be used throughout this tutorial:

- [Power BI Desktop](#), May 2017 release or later
- [Power Query SDK for Visual Studio](#)
- [Fiddler](#)—Optional, but recommended for viewing and debugging requests to your REST service

It's strongly suggested that you review:

- [Installing the PowerQuery SDK](#)
- [Starting to Develop Custom Connectors](#)
- [Creating your first connector: Hello World](#)
- [Handling Data Access](#)
- [Handling Authentication](#)

Parts

PART	LESSON	DETAILS
1	OData	Create a simple Data Connector over an OData service
2	Rest	Connect to a REST API that returns a JSON response
3	Nav Tables	Provide a navigation experience for your source
4	Data Source Paths	How credentials are identified for your data source
5	Paging	Read with a paged response from a web service
6	Enforcing Schema	Enforce table structure and column data types
7	Advanced Schema	Dynamically enforce table structure using M types and external metadata

PART	LESSON	DETAILS
8	Diagnostics	Add detailed tracing to the connector
9	Test Connection	Implement a TestConnection handler to enable refresh through the gateway
10	Query Folding (part 1)	Implement basic query folding handlers

TripPin Part 1 - Data Connector for an OData Service

5 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

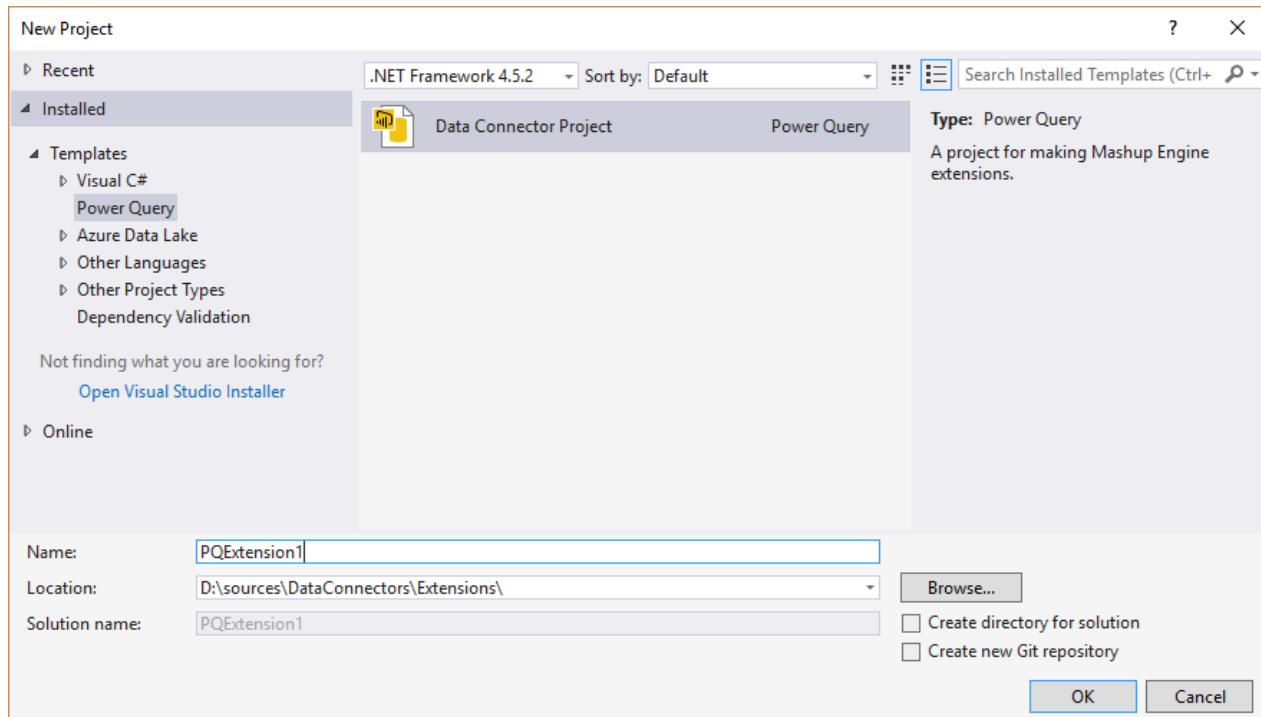
In this lesson, you will:

- Create a new Data Connector project using the Visual Studio SDK
- Author a base function to pull data from a source
- Test your connector in Visual Studio
- Register your connector in Power BI Desktop

Creating a Basic OData Connector

In this section, you will create a new Data Connector project, provide some basic information, and test it in Visual Studio.

Open Visual Studio, and create a new Project. Under the Power Query folder, select the Data Connector project. For this sample, set the project name to `TripPin`.



Open the `TripPin.pq` file and paste in the following connector definition.

```

section TripPin;

[DataSource.Kind="TripPin", Publish="TripPin.Publish"]
shared TripPin.Feed = Value.ReplaceType(TripPinImpl, type function (url as Uri.Type) as any);

TripPinImpl = (url as text) =>
    let
        source = OData.Feed(url)
    in
        source;

// Data Source Kind description
TripPin = [
    Authentication = [
        Anonymous = []
    ],
    Label = "TripPin Part 1 - OData"
];

// Data Source UI publishing description
TripPin.Publish = [
    Beta = true,
    Category = "Other",
    ButtonText = { "TripPin OData", "TripPin OData" }
];

```

This connector definition contains:

- A Data Source definition record for the TripPin connector
- A declaration that Implicit (Anonymous) is the only authentication type for this source
- A function (`TripPinImpl`) with an implementation that calls `OData.Feed`
- A shared function (`TripPin.Feed`) that sets the parameter type to `Uri.Type`
- A Data Source publishing record that will allow the connector to appear in the Power BI **Get Data** dialog

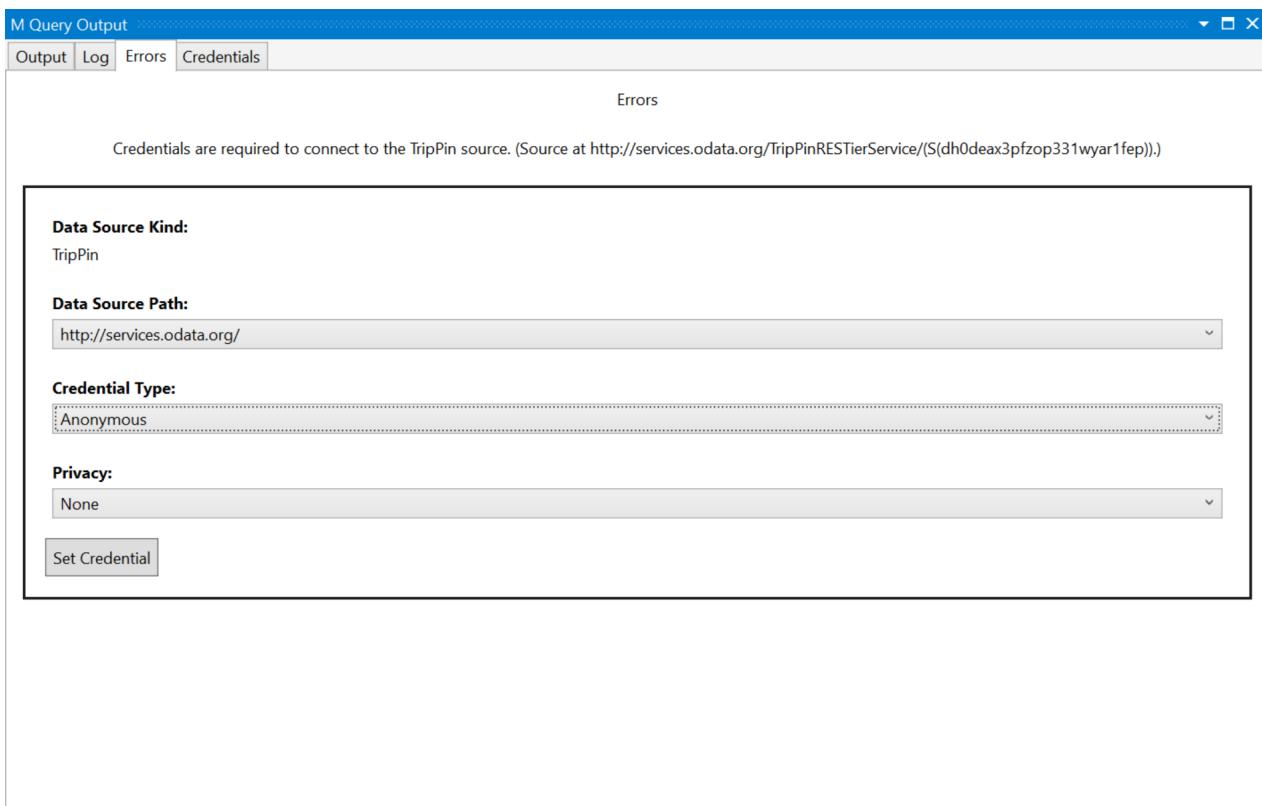
Open the TripPin.query.pq file. Replace the current contents with a call to your exported function.

```
TripPin.Feed("https://services.odata.org/v4/TripPinService/")
```

Select the **Start** button to launch the M Query utility.

The .query.pq file is used to test out your extension without having to deploy it to your Power BI Desktop's bin folder. Selecting the **Start** button (or press **F5**) automatically compiles your extension and launches the M Query utility.

Running your query for the first time results in a credential error. In Power Query, the hosting application would convert this error into a credential prompt. In Visual Studio, you'll receive a similar prompt that calls out which data source is missing credentials and its data source path. Select the shortest of the data source paths (<https://services.odata.org/>)—this will apply your credential to all URLs under this path. Select the **Anonymous** credential type, and then select **Set Credential**.



Select **OK** to close the dialog, and then select the **Start** button once again. You see a query execution status dialog, and finally a Query Result table showing the data returned from your query.

The screenshot shows the 'MQuery Output' dialog with a yellow header bar. Below the header, there are tabs for 'Output' (selected), 'Log', 'Errors', and 'Credentials'. The main area is titled 'Query Result' and contains a table with three columns: 'Name', 'Data', and 'Signature'.

Name	Data	Signature
Airlines	[Table]	table
Airports	[Table]	table
GetNearestAirport	[Function]	function (lat as number, lon as number) as record
GetPersonWithMostFriends	[Function]	function () as record
Me	[Table]	singleton
NewComePeople	[Table]	table
People	[Table]	table

You can try out a few different OData URLs in the test file to see what how different results are returned. For example:

- <https://services.odata.org/v4/TripPinService/Me>
- [https://services.odata.org/v4/TripPinService/GetPersonWithMostFriends\(\)](https://services.odata.org/v4/TripPinService/GetPersonWithMostFriends())
- <https://services.odata.org/v4/TripPinService/People>

The TripPin.query.pq file can contain single statements, let statements, or full section documents.

```

let
    Source = TripPin.Feed("https://services.odata.org/v4/TripPinService/"),
    People = Source{[Name="People"]}[Data],
    SelectColumns = Table.SelectColumns(People, {"UserName", "FirstName", "LastName"})
in
    SelectColumns

```

Open [Fiddler](#) to capture HTTP traffic, and run the query. You should see a few different requests to services.odata.org, generated by the mashup container process. You can see that accessing the root URL of the service results in a 302 status and a redirect to the longer version of the URL. Following redirects is another behavior you get “for free” from the base library functions.

One thing to note if you look at the URLs is that you can see the query folding that happened with the `SelectColumns` statement.

```
https://services.odata.org/v4/TripPinService/People?$select=UserName%2CFirstName%2CLastName
```

If you add more transformations to your query, you can see how they impact the generated URL.

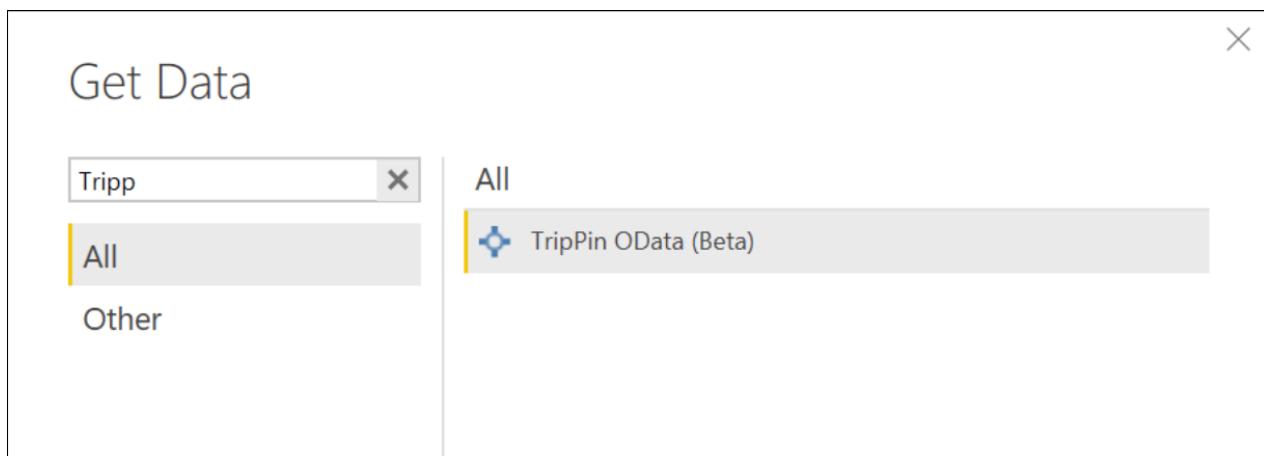
This behavior is important to note. Even though you did not implement explicit folding logic, your connector inherits these capabilities from the [OData.Feed](#) function. M statements are compose-able—filter contexts will flow from one function to another, whenever possible. This is similar in concept to the way data source functions used within your connector inherit their authentication context and credentials. In later lessons, you'll replace the use of [OData.Feed](#), which has native folding capabilities, with [Web.Contents](#), which does not. To get the same level of capabilities, you'll need to use the `Table.View` interface and implement your own explicit folding logic.

Loading Your Extension in Power BI Desktop

To use your extension in Power BI Desktop, you'll need to copy your connector project's output file (TripPin.mez) to your Custom Connectors directory.

1. In Visual Studio, select **Build | Build Solution (F6)** from the menu bar. This will generate the .mez file for your project. By default, this will go in your project's bin\Debug folder.
2. Create a `[My Documents]\Power BI Desktop\Custom Connectors` directory.
3. Copy the extension file into this directory.
4. Enable the **Custom data connectors** preview feature in Power BI Desktop (under **File > Options and settings > Custom data connectors**).
5. Restart Power BI Desktop.
6. Select **Get Data > More** to bring up the **Get Data** dialog.

You should be able to locate your extension by typing its name into the search box.

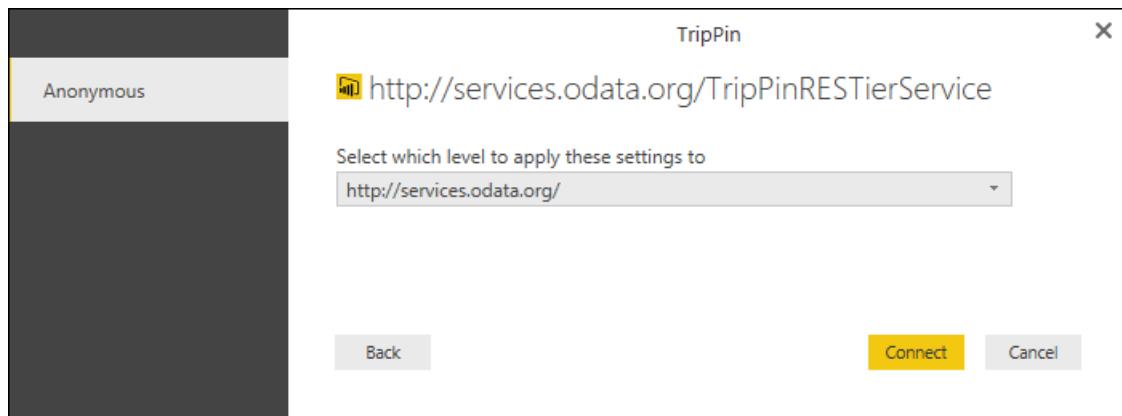


Double click on the function name and the function invocation dialog will appear. Enter the root URL of the

service (<https://services.odata.org/v4/TripPinService/>), and select **OK**.



Since this is the first time you are accessing this data source, you'll receive a prompt for credentials. Check that the shortest URL is selected, and then select **Connect**.



Notice that instead of getting a simple table of data, the navigator appears. This is because the **OData.Feed** function returns a table with special metadata on top of it that the Power Query experience knows to display as a navigation table. This walkthrough will cover how you can create and customize your own navigation table in a future lesson.

The screenshot shows the Navigator tool interface. On the left, there's a tree view of available services and tables. The 'Airlines' table under the 'http://services.odata.org/TripPinRESTierService/' service is selected. On the right, the 'Airlines' table is displayed in a grid format with three rows:

AirlineCode	Name
AA	American Airlines
FM	Shanghai Airline
MU	China Eastern Airlines

At the bottom right are 'Load', 'Edit', and 'Cancel' buttons.

Select the **Me** table, and then select **Edit**. Notice that the columns already have types assigned (well, most of them). This is another feature of the underlying [OData.Feed](#) function. If you watch the requests in [Fiddler](#), you'll see that you've fetched the service's \$metadata document. The engine's OData implementation does this automatically to determine the service's schema, data types, and relationships.

The screenshot shows the EntityDataSource Editor for the 'Me' table. The table has one row with the following data:

UserName	FirstName	LastName	MiddleName	Gender	Age	Emails
aprilcline	April	Cline	null	Female	null	List

Conclusion

This lesson walked you through the creation of a simple connector based on the [OData.Feed](#) library function. As you saw, very little logic is needed to enable a fully functional connector over the `odata` base function. Other extensibility enabled functions, such as [ODBC.DataSource](#), provide similar capabilities.

In the next lesson, you'll replace the use of [OData.Feed](#) with a less capable function—[Web.Contents](#). Each lesson will implement more connector features, including paging, metadata/schema detection, and query folding to the OData query syntax, until your custom connector supports the same range of capabilities as [OData.Feed](#).

TripPin Part 2 - Data Connector for a REST Service

7 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Create a base function that calls out to a REST API using [Web.Contents](#)
- Learn how to set request headers and process a JSON response
- Use Power BI Desktop to wrangle the response into a user friendly format

This lesson converts the OData based connector for the [TripPin service](#) (created in the [previous lesson](#)) to a connector that resembles something you'd create for any RESTful API. OData is a RESTful API, but one with a fixed set of conventions. The advantage of OData is that it provides a schema, data retrieval protocol, and standard query language. Taking away the use of [OData.Feed](#) will require us to build these capabilities into the connector ourselves.

Recap of the OData Connector

Before you remove the OData functions from your connector, let's do a quick review of what it currently does (mostly behind the scenes) to retrieve data from the service.

Open the TripPin connector project from [Part 1](#) in Visual Studio. Open the Query file and paste in the following query:

```
TripPin.Feed("https://services.odata.org/v4/TripPinService/Me")
```

Open Fiddler and then select the Start button in Visual Studio.

In Fiddler, you'll see three requests to the server:

#	Result	Protocol	Host	URL
25	200	HTTP	services.odata.org	/TripPinRESTTierService/(S(rjbl4y1givihcajsba2xiorr))/Me
26	200	HTTP	services.odata.org	/TripPinRESTTierService/(S(rjbl4y1givihcajsba2xiorr))/\$metadata
27	204	HTTP	services.odata.org	/TripPinRESTTierService/(S(rjbl4y1givihcajsba2xiorr))/Me/BestFriend

- `/Me` —the actual URL you are requesting.
- `/$metadata` —a call automatically made by the `OData.Feed` function to determine schema and type information about the response.
- `/Me/BestFriend` —one of the fields that was (eagerly) pulled when you listed the `/Me` singleton. In this case the call resulted in a `204 No Content` status.

M evaluation is mostly lazy. In most cases, data values are only retrieved/pulled when they are needed. There are scenarios (like the `/Me/BestFriend` case) where a value is pulled eagerly. This tends to occur when type information is needed for a member, and the engine has no other way to determine the type than to retrieve the value and inspect it. Making things lazy (that is, avoiding eager pulls) is one of the key aspects to making an M connector performant.

Note the request headers that were sent along with the requests and the JSON format of the response of the `/Me` request.

```
{
    "@odata.context": "https://services.odata.org/v4/TripPinService/$metadata#Me",
    "UserName": "aprilcline",
    "FirstName": "April",
    "LastName": "Cline",
    "MiddleName": null,
    "Gender": "Female",
    "Age": null,
    "Emails": [ "April@example.com", "April@contoso.com" ],
    "FavoriteFeature": "Feature1",
    "Features": [ ],
    "AddressInfo": [
        {
            "Address": "P.O. Box 555",
            "City": {
                "Name": "Lander",
                "CountryRegion": "United States",
                "Region": "WY"
            }
        }
    ],
    "HomeAddress": null
}
```

When the query finishes evaluating, the M Query Output window should show the Record value for the Me singleton.

Query Result		
Name	Value	
UserName	aprilcline	
FirstName	April	
LastName	Cline	
MiddleName		
Gender	Female	
Age		
Emails	[List]	
AddressInfo	[List]	
HomeAddress		
FavoriteFeature	Feature1	
Features	[List]	
Friends	[Table]	
BestFriend		
Trips	[Table]	
GetFavoriteAirline		
GetFriendsTrips		
UpdatePersonLastName		

If you compare the fields in the output window with the fields returned in the raw JSON response, you'll notice a mismatch. The query result has additional fields (`Friends`, `Trips`, `GetFriendsTrips`) that don't appear anywhere in the JSON response. The `OData.Feed` function automatically appended these fields to the record based on the schema returned by `$metadata`. This is a good example of how a connector might augment and/or reformat the response from the service to provide a better user experience.

Creating a Basic REST Connector

You'll now be adding a new exported function to your connector that calls `Web.Contents`.

To be able to make successful web requests to the OData service, however, you'll have to set some [standard OData headers](#). You'll do this by defining a common set of headers as a new variable in your connector:

```
DefaultRequestHeaders = [
    #"Accept" = "application/json;odata.metadata=minimal", // column name and values only
    #"OData-MaxVersion" = "4.0"                            // we only support v4
];
```

You'll change your implementation of your `TripPin.Feed` function so that rather than using `OData.Feed`, it uses `Web.Contents` to make a web request, and parses the result as a JSON document.

```
TripPinImpl = (url as text) =>
let
    source = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
    json = Json.Document(source)
in
    json;
```

You can now test this out in Visual Studio using the query file. The result of the /Me record now resembles the raw JSON that you saw in the Fiddler request.

If you watch Fiddler when running the new function, you'll also notice that the evaluation now makes a single web request, rather than three. Congratulations—you've achieved a 300% performance increase! Of course, you've now lost all the type and schema information, but there's no need to focus on that part just yet.

Update your query to access some of the TripPin Entities/Tables, such as:

- <https://services.odata.org/v4/TripPinService/Airlines>
- <https://services.odata.org/v4/TripPinService/Airports>
- <https://services.odata.org/v4/TripPinService/Me/Trips>

You'll notice that the paths that used to return nicely formatted tables now return a top level "value" field with an embedded [List]. You'll need to do some transformations on the result to make it usable for Power BI scenarios.

The screenshot shows the MQuery Output interface. At the top, there's a yellow header bar with a close button (X). Below it is a toolbar with tabs: Output (which is selected), Log, Errors, and Credentials. The main area is titled 'Query Result' and contains a table with two rows. The first row has columns 'Name' and 'Value'. The second row has a single column 'value' with the value '[List]'. The table has a light blue header row.

Name	Value
@odata.context	http://services.odata.org/TripPinRESTierService/(S(njbl4y1givihcajsba2xiorr))/\$.metadata#Collection(Microsoft.OData.Service
value	[List]

Authoring Transformations in Power Query

While it is certainly possible to author your M transformations by hand, most people prefer to use Power Query to shape their data. You'll open your extension in Power BI Desktop and use it to design queries to turn the output into a more user friendly format. Rebuild your solution, copy the new extension file to your Custom Data Connectors directory, and relaunch Power BI Desktop.

Start a new Blank Query, and paste the following into the formula bar:

```
= TripPin.Feed("https://services.odata.org/v4/TripPinService/Airlines")
```

Be sure to include the = sign.

Manipulate the output until it looks like the original OData feed—a table with two columns: AirlineCode and Name.

	AirlineCode	Name
1	AA	American Airlines
2	FM	Shanghai Airline
3	MU	China Eastern Airlines

The resulting query should look something like this:

```
let
    Source = TripPin.Feed("https://services.odata.org/v4/TripPinService/Airlines"),
    value = Source[value],
    toTable = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    expand = Table.ExpandRecordColumn(toTable, "Column1", {"AirlineCode", "Name"}, {"AirlineCode", "Name"})
in
    expand
```

Give the query a name ("Airlines").

Create a new Blank Query. This time, use the `TripPin.Feed` function to access the /Airports entity. Apply transforms until you get something similar to the share shown below. The matching query can also be found below—give this query a name ("Airports") as well.

	Name	IcaoCode	IataCode	Address	City	CountryRegion	Region	Latitude	Longitude
1	San Francisco International Airport	KSFO	SFO	South McDonnell Road, San Francisco, CA 94128	San Francisco	United States	California	37.61888889	-122.3747222
2	Los Angeles International Airport	KLAX	LAX	1 World Way, Los Angeles, CA, 90045	Los Angeles	United States	California	33.9425	-118.4080556
3	Shanghai Hongqiao International Airport	ZSSS	SHA	Hongqiao Road 2550, Changning District	Shanghai	China	Shanghai	31.19777778	121.33611111
4	Beijing Capital International Airport	ZBAA	PEK	Airport Road, Chaoyang District, Beijing, 100621	Beijing	China	Beijing	40.08	116.5844444
5	John F. Kennedy International Airport	KJFK	JFK	Jamaica, New York, NY 11430	New York City	United States	New York	40.63972222	-73.77888889

```
let
    Source = TripPin.Feed("https://services.odata.org/v4/TripPinService/Airports"),
    value = Source[value],
    #"Converted to Table" = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"Expanded Column1" = Table.ExpandRecordColumn(#"Converted to Table", "Column1", {"Name", "IcaoCode", "IataCode", "Location"}, {"Name", "IcaoCode", "IataCode", "Location"}),
    #"Expanded Location" = Table.ExpandRecordColumn(#"Expanded Column1", "Location", {"Address", "Loc", "City"}, {"Address", "Loc", "City"}),
    #"Expanded City" = Table.ExpandRecordColumn(#"Expanded Location", "City", {"Name", "CountryRegion", "Region"}, {"Name.1", "CountryRegion", "Region"}),
    #"Renamed Columns" = Table.RenameColumns(#"Expanded City", {"Name.1", "City"}),
    #"Expanded Loc" = Table.ExpandRecordColumn(#"Renamed Columns", "Loc", {"coordinates"}, {"coordinates"}),
    #"Added Custom" = Table.AddColumn(#"Expanded Loc", "Latitude", each [coordinates]{1}),
    #"Added Custom1" = Table.AddColumn(#"Added Custom", "Longitude", each [coordinates]{0}),
    #"Removed Columns" = Table.RemoveColumns(#"Added Custom1", {"coordinates"}),
    #"Changed Type" = Table.TransformColumnTypes(#"Removed Columns", {"Name", type text}, {"IcaoCode", type text}, {"IataCode", type text}, {"Address", type text}, {"City", type text}, {"CountryRegion", type text}, {"Region", type text}, {"Latitude", type number}, {"Longitude", type number})
in
    #"Changed Type"
```

You can repeat this process for additional paths under the service. Once you're ready, move onto the next step of creating a (mock) navigation table.

Simulating a Navigation Table

Now you are going to build a table (using M code) that presents your nicely formatted TripPin entities.

Start a new Blank Query and bring up the Advanced Editor.

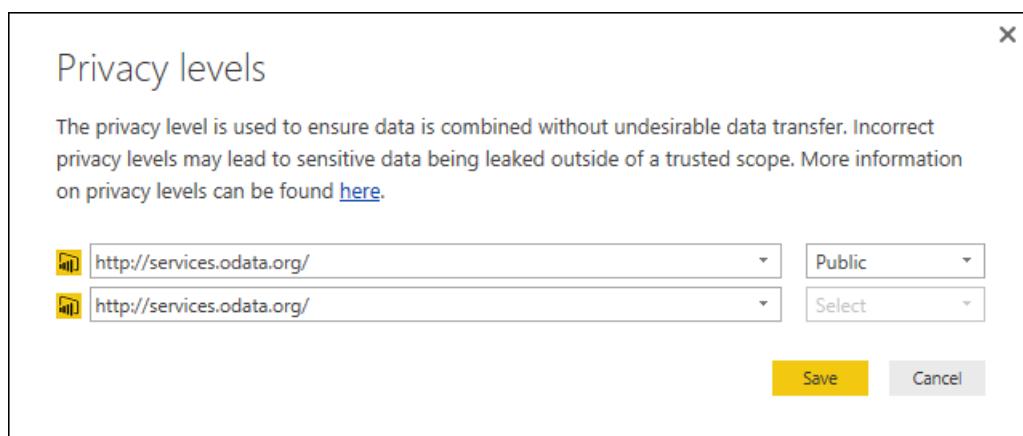
Paste in the following query:

```
let
    source = #table({ "Name", "Data"}, {
        { "Airlines", Airlines },
        { "Airports", Airports }
    })
in
    source
```

If you have not set your Privacy Levels setting to "Always ignore Privacy level settings" (also known as "Fast Combine") you'll see a privacy prompt.



Privacy prompts appear when you're combining data from multiple sources and have not yet specified a privacy level for the source(s). Select the **Continue** button and set the privacy level of the top source to **Public**.



Select **Save** and your table will appear. While this isn't a navigation table yet, it provides the basic functionality you need to turn it into one in a subsequent lesson.

Data combination checks do not occur when accessing multiple data sources from within an extension. Since all data source calls made from within the extension inherit the same authorization context, it is assumed they are "safe" to combine. Your extension will always be treated as a single data source when it comes to data combination rules. Users would still receive the regular privacy prompts when combining your source with other M sources.

If you run Fiddler and click the **Refresh Preview** button in the Query Editor, you'll notice separate web requests for each item in your navigation table. This indicates that an eager evaluation is occurring, which isn't ideal when building navigation tables with a lot of elements. Subsequent lessons will show how to build a proper navigation

table that supports lazy evaluation.

Conclusion

This lesson showed you how to build a simple connector for a REST service. In this case, you turned an existing OData extension into a standard REST extension (using [Web.Contents](#)), but the same concepts apply if you were creating a new extension from scratch.

In the next lesson, you'll take the queries created in this lesson using Power BI Desktop and turn them into a true navigation table within the extension.

TripPin Part 3 - Navigation Tables

4 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Create a navigation table for a fixed set of queries
- Test the navigation table in Power BI Desktop

This lesson adds a navigation table to the TripPin connector created in the [previous lesson](#). When your connector used the `OData.Feed` function ([Part 1](#)), you received the navigation table “for free”, as derived from the OData service’s \$metadata document. When you moved to the `Web.Contents` function ([Part 2](#)), you lost the built-in navigation table. In this lesson, you’ll take a set of fixed queries you created in Power BI Desktop and add the appropriate metadata for Power Query to popup the **Navigator** dialog for your data source function.

See the [Navigation Table documentation](#) for more information about using navigation tables.

Defining Fixed Queries in the Connector

A simple connector for a REST API can be thought of as a fixed set of queries, each returning a table. These tables are made discoverable through the connector’s navigation table. Essentially, each item in the navigator is associated with a specific URL and set of transformations.

You’ll start by copying the queries you wrote in Power BI Desktop (in the previous lesson) into your connector file. Open the TripPin Visual Studio project, and paste the Airlines and Airports queries into the TripPin.pq file. You can then turn those queries into functions that take a single text parameter:

```

GetAirlinesTable = (url as text) as table =>
let
    source = TripPin.Feed(url & "Airlines"),
    value = source[value],
    toTable = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    expand = Table.ExpandRecordColumn(toTable, "Column1", {"AirlineCode", "Name"}, {"AirlineCode", "Name"})
in
    expand;

GetAirportsTable = (url as text) as table =>
let
    source = TripPin.Feed(url & "Airports"),
    value = source[value],
    #"Converted to Table" = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"Expanded Column1" = Table.ExpandRecordColumn(#"Converted to Table", "Column1", {"Name", "IcaoCode", "IataCode", "Location"}, {"Name", "IcaoCode", "IataCode", "Location"}),
    #"Expanded Location" = Table.ExpandRecordColumn(#"Expanded Column1", "Location", {"Address", "Loc", "City"}, {"Address", "Loc", "City"}),
    #"Expanded City" = Table.ExpandRecordColumn(#"Expanded Location", "City", {"Name", "CountryRegion", "Region"}, {"Name.1", "CountryRegion", "Region"}),
    #"Renamed Columns" = Table.RenameColumns(#"Expanded City", {"Name.1", "City"}),
    #"Expanded Loc" = Table.ExpandRecordColumn(#"Renamed Columns", "Loc", {"coordinates"}, {"coordinates"}),
    #"Added Custom" = Table.AddColumn(#"Expanded Loc", "Latitude", each [coordinates]{1}),
    #"Added Custom1" = Table.AddColumn(#"Added Custom", "Longitude", each [coordinates]{0}),
    #"Removed Columns" = Table.RemoveColumns(#"Added Custom1", {"coordinates"}),
    #"Changed Type" = Table.TransformColumnTypes(#"Removed Columns", {"Name", type text}, {"IcaoCode", type text}, {"IataCode", type text}, {"Address", type text}, {"City", type text}, {"CountryRegion", type text}, {"Region", type text}, {"Latitude", type number}, {"Longitude", type number})
in
    #"Changed Type";

```

Next you'll import the mock navigation table query you wrote that creates a fixed table linking to these data set queries. Call it `TripPinNavTable`:

```

TripPinNavTable = (url as text) as table =>
let
    source = #table({ "Name", "Data"}, [
        { "Airlines", GetAirlinesTable(url) },
        { "Airports", GetAirportsTable(url) }
    ])
in
    source;

```

Finally you'll declare a new shared function, `TripPin.Contents`, that will be used as your main data source function. You'll also remove the `Publish` value from `TripPin.Feed` so that it no longer shows up in the **Get Data** dialog.

```

[DataSource.Kind="TripPin"]
shared TripPin.Feed = Value.ReplaceType(TripPinImpl, type function (url as Uri.Type) as any);

[DataSource.Kind="TripPin", Publish="TripPin.Publish"]
shared TripPin.Contents = Value.ReplaceType(TripPinNavTable, type function (url as Uri.Type) as any);

```

NOTE

Your extension can mark multiple functions as `shared`, with or without associating them with a `DataSource.Kind`. However, when you associate a function with a specific `DataSource.Kind`, each function **must** have the same set of *required* parameters, with the same name and type. This is because the data source function parameters are combined to make a 'key' used for looking up cached credentials.

You can test your `TripPin.Contents` function using your `TripPin.query.pq` file. Running the following test query will give you a credential prompt, and a simple table output.

```
TripPin.Contents("https://services.odata.org/v4/TripPinService/")
```

The screenshot shows the 'M Query Output' window. At the top, there are tabs for 'Output', 'Log', 'Errors', and 'Credentials'. The 'Output' tab is selected. Below the tabs, it says 'Query Result'. A table is displayed with two rows:

Name	Data
Airlines	[Table]
Airports	[Table]

Creating a Navigation Table

You'll use the handy `Table.ToNavigationTable` function to format your static table into something that Power Query will recognize as a Navigation Table.

```
Table.ToNavigationTable = (
    table as table,
    keyColumns as list,
    nameColumn as text,
    dataColumn as text,
    itemKindColumn as text,
    itemNameColumn as text,
    isLeafColumn as text
) as table =>
    let
        tableType = Value.Type(table),
        newTableType = Type.AddTableKey(tableType, keyColumns, true) meta
        [
            NavigationTable.NameColumn = nameColumn,
            NavigationTable.DataColumn = dataColumn,
            NavigationTable.ItemKindColumn = itemKindColumn,
            Preview.DelayColumn = itemNameColumn,
            NavigationTable.IsLeafColumn = isLeafColumn
        ],
        navigationTable = Value.ReplaceType(table, newTableType)
    in
        navigationTable;
```

After copying this into your extension file, you'll update your `TripPinNavTable` function to add the navigation table

fields.

```
TripPinNavTable = (url as text) as table =>
let
    source = #table({ "Name", "Data", "ItemKind", "ItemName", "IsLeaf"}, {
        { "Airlines", GetAirlinesTable(url), "Table", "Table", true },
        { "Airports", GetAirportsTable(url), "Table", "Table", true }
    }),
    navTable = Table.ToNavigationTable(source, { "Name"}, "Name", "Data", "ItemKind", "ItemName", "IsLeaf")
in
    navTable;
```

Running your test query again will give you a similar result as last time—with a few more columns added.

The screenshot shows the 'M Query Output' window in Visual Studio. The 'Output' tab is selected. Below it, a table titled 'Query Result' is displayed with the following schema:

Name	Data	ItemKind	ItemName	IsLeaf
Airlines	[Table]	Table	Table	True
Airports	[Table]	Table	Table	True

NOTE

You will not see the **Navigator** window appear in Visual Studio. The **M Query Output** window always displays the underlying table.

If you copy your extension over to your Power BI Desktop custom connector and invoke the new function from the **Get Data** dialog, you'll see your navigator appear.

The screenshot shows the 'Navigator' window in Power BI Desktop. On the left, there is a navigation tree with the following structure:

- http://services.odata.org/TripPinRESTierService...
 - Airlines
 - Airports

On the right, there is a detailed view of the 'Airlines' table:

AirlineCode	Name
AA	American Airlines
FM	Shanghai Airline
MU	China Eastern Airlines

If you right click on the root of the navigation tree and select **Edit**, you'll see the same table as you did within Visual Studio.

The screenshot shows the Microsoft Power Query Editor interface. The top ribbon has tabs for File, Home, Transform, Add Column, View, and a Help icon. The Home tab is selected. The ribbon contains various icons for file operations like Close & Apply, New Source, Refresh, and Transform. Below the ribbon, there are sections for Data Sources, Parameters, and Query settings. The main area displays a query titled "TripPin.Contents("http://services.odata.org/TripPinRESTierService/(\$").Tables[0]" which retrieves data from a navigation table. The preview pane shows two rows of data with columns: Name, Data, ItemKind, ItemName, and IsLeaf. The first row is labeled "Airlines" and the second "Airports". The preview also indicates that the source is a Table. On the right side, the "Query Settings" pane is open, showing the "Properties" section with the URL "http://services.odata.org/TripPinRESTierS" and the "Applied Steps" section which lists "Source".

Conclusion

In this tutorial, you added a [Navigation Table](#) to your extension. Navigation Tables are a key feature that make connectors easier to use. In this example your navigation table only has a single level, but the Power Query UI supports displaying navigation tables that have multiple dimensions (even when they are ragged).

TripPin Part 4 - Data Source Paths

6 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Simplify the connection logic for your connector
- Improve the navigation table experience

This lesson simplifies the connector built in the [previous lesson](#) by removing its required function parameters, and improving the user experience by moving to a dynamically generated navigation table.

For an in-depth explanation of how credentials are identified, see the [Data Source Paths section](#) of [Handling Authentication](#).

Data Source Paths

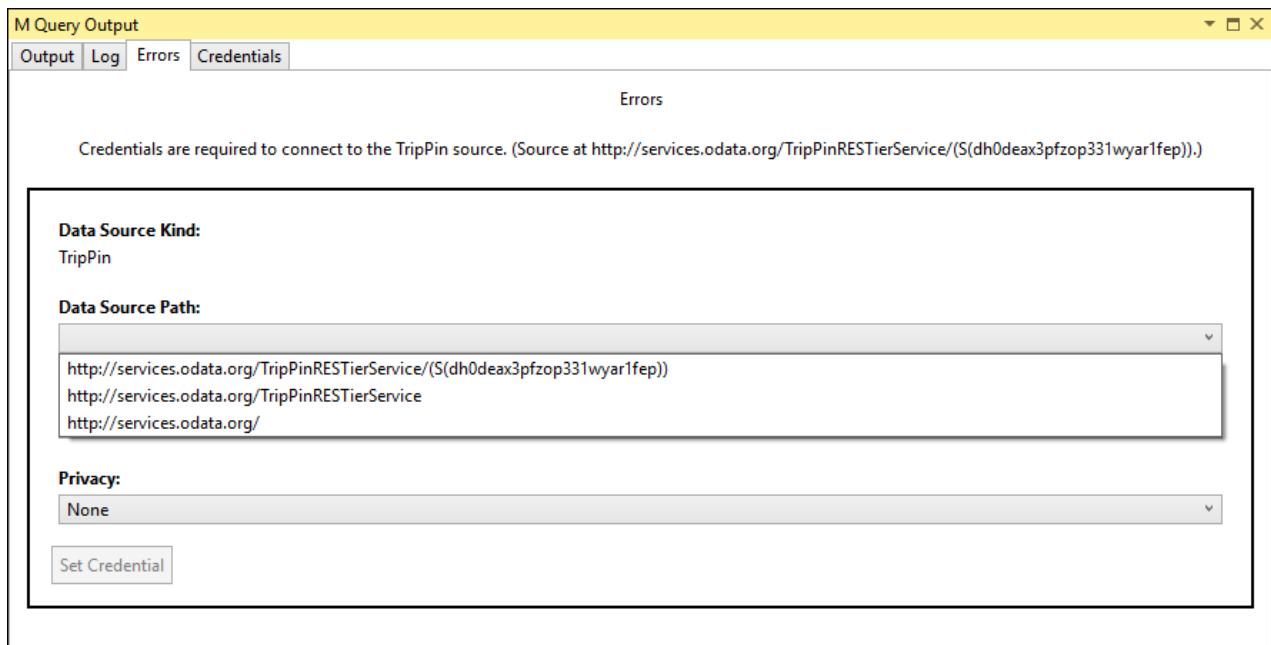
When invoking a [data source function](#), the M engine identifies which credentials to use during an evaluation by doing a lookup based on the [Data Source Kind](#) and [Data Source Path](#) values.

In the [previous lesson](#) you shared two data source functions, both with a single *Uri.Type* parameter.

```
[DataSource.Kind="TripPin"]
shared TripPin.Feed = Value.ReplaceType(TripPinImpl, type function (url as Uri.Type) as any);

[DataSource.Kind="TripPin", Publish="TripPin.Publish"]
shared TripPin.Contents = Value.ReplaceType(TripPinNavTable, type function (url as Uri.Type) as any);
```

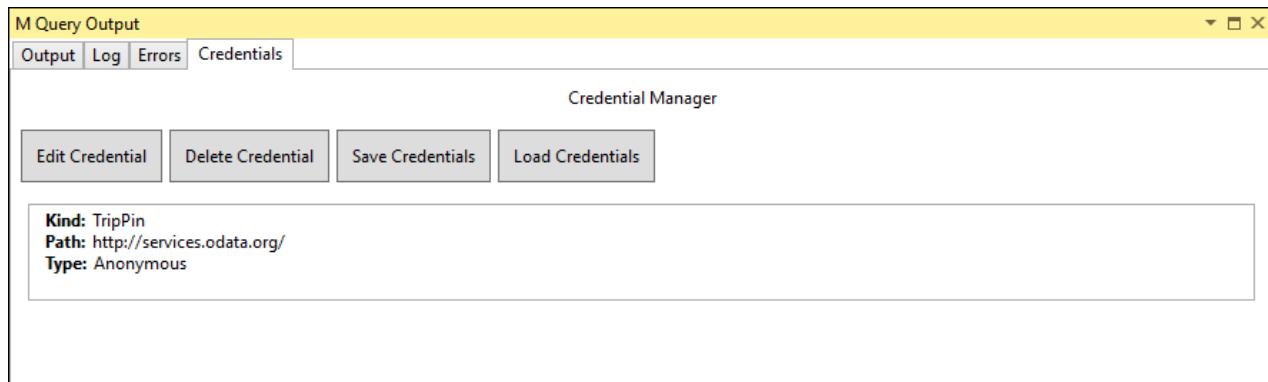
The first time you run a query that uses one of the functions, you'll receive a credential prompt with drop downs that lets you select a path and an authentication type.



If you run the same query again, with the same parameters, the M engine is able to locate the cached credentials,

and no credential prompt is shown. If you modify the `url` argument to your function so that the base path no longer matches, a new credential prompt is displayed for the new path.

You can see any cached credentials on the Credentials table in the **M Query Output** window.



Depending on the type of change, modifying the parameters of your function will likely result in a credential error.

Simplifying the Connector

You'll now simplify your connector by removing the parameters for your data source function (`TripPin.Contents`). You'll also remove the `shared` qualifier for `TripPin.Feed`, and leave it as an internal-only function.

One of the design philosophies of Power Query is to keep the initial data source dialog as simple as possible. If at all possible, you should provide the user with choices at the Navigator level, rather than on the connection dialog. If a user provided value can be determined programmatically, consider adding it as the top level of your navigation table rather than a function parameter.

For example, when connecting to a relational database, you might need server, database, and table names. Once you know the server to connect to, and credentials have been provided, you could use the database's API to fetch a list of databases, and a list of tables contained within each database. In this case, to keep your initial connect dialog as simple as possible, only the server name should be a required parameter—`Database` and `Table` would be levels of your navigation table.

Since the TripPin service has a fixed URL endpoint, you don't need to prompt the user for any values. You'll remove the `url` parameter from your function, and define a `BaseUrl` variable in your connector.

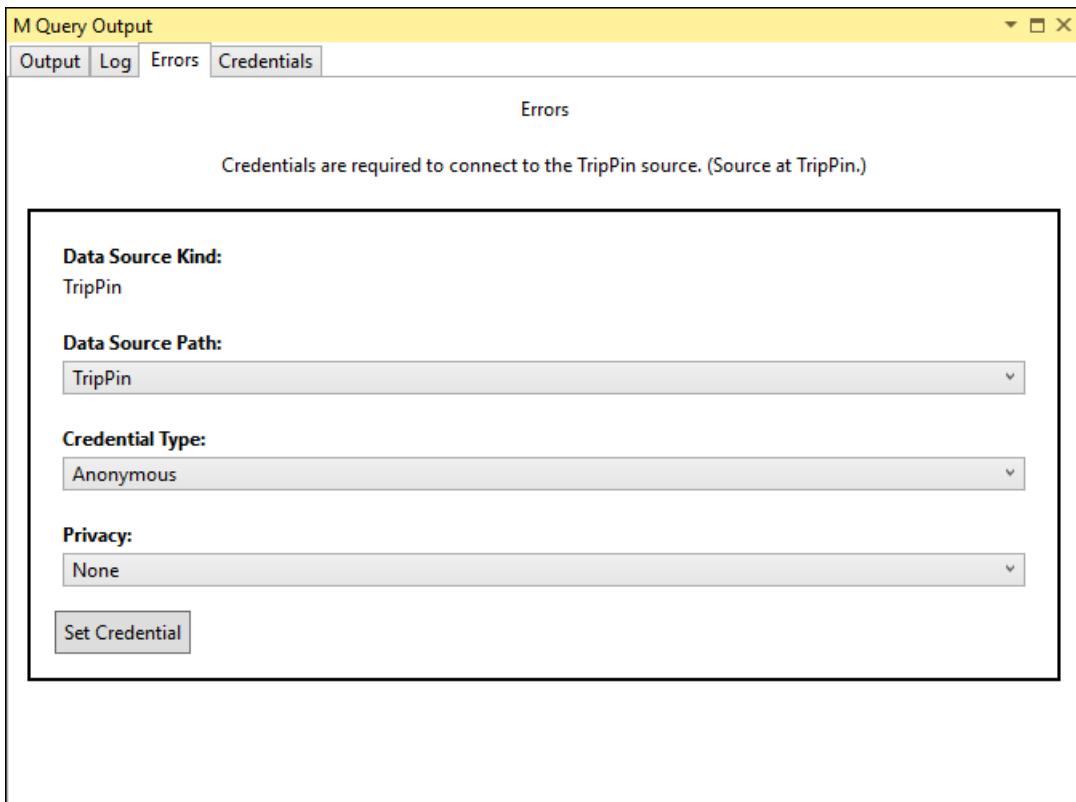
```
BaseUrl = "https://services.odata.org/v4/TripPinService/";

[DataSource.Kind="TripPin", Publish="TripPin.Publish"]
shared TripPin.Contents = () => TripPinNavTable(BaseUrl) as table;
```

You'll keep the `TripPin.Feed` function, but no longer make it shared, no longer associate it with a Data Source Kind, and simplify its declaration. From this point on, you'll only use it internally within this section document.

```
TripPin.Feed = (url as text) =>
let
    source = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
    json = Json.Document(source)
in
    json;
```

If you update the `TripPin.Contents()` call in your `TripPin.query.pq` file and run it in Visual Studio, you'll see a new credential prompt. Note that there is now a single Data Source Path value—TripPin.



Improving the Navigation Table

In the [first tutorial](#) you used the built-in `odata` functions to connect to the TripPin service. This gave you a really nice looking navigation table, based on the TripPin service document, with no additional code on your side. The `OData.Feed` function automatically did the hard work for you. Since you're "roughing it" by using `Web.Contents` rather than `OData.Feed`, you'll need to recreate this navigation table yourself.

You're going to make the following changes:

1. Define a list of items to show in your navigation table
2. Do away with the entity specific functions (`GetAirlineTables` and `GetAirportsTable`)

Generating a Navigation Table from a List

You'll list the entities you want to expose in the navigation table, and build the appropriate URL to access them. Since all of the entities are under the same root path, you'll be able build these URLs dynamically.

To simplify the example, you'll only expose the three entity sets (Airlines, Airports, People), which would be exposed as Tables in M, and skip the singleton (Me) which would be exposed as a Record. You'll skip adding the functions until a later lesson.

```
RootEntities = {
    "Airlines",
    "Airports",
    "People"
};
```

You then update your `TripPinNavTable` function to build the table a column at a time. The [Data] column for each entity is retrieved by calling `TripPin.Feed` with the full URL to the entity.

```
TripPinNavTable = (url as text) as table =>
let
    entitiesAsTable = Table.FromList(RootEntities, Splitter.SplitByNothing()),
    rename = Table.RenameColumns(entitiesAsTable, {[{"Column1", "Name"}]}),
    // Add Data as a calculated column
    withData = Table.AddColumn(rename, "Data", each TripPin.Feed(Uri.Combine(url, [Name])), Uri.Type),
    // Add ItemKind and ItemName as fixed text values
    withItemKind = Table.AddColumn(withData, "ItemKind", each "Table", type text),
    withItemName = Table.AddColumn(withItemKind, "ItemName", each "Table", type text),
    // Indicate that the node should not be expandable
    withIsLeaf = Table.AddColumn(withItemName, "IsLeaf", each true, type logical),
    // Generate the nav table
    navTable = Table.ToNavigationTable(withIsLeaf, {"Name", "Name", "Data", "ItemKind", "ItemName",
    "IsLeaf"})
in
    navTable;
```

When dynamically building URL paths, make sure you're clear where your forward slashes (/) are! Note that `Uri.Combine` uses the following rules when combining paths:

- When the `relativeUri` parameter starts with a /, it will replace the entire path of the `baseUri` parameter
- If the `relativeUri` parameter *does not* start with a / and `baseUri` ends with a /, the path is appended
- If the `relativeUri` parameter *does not* start with a / and `baseUri` *does not* end with a /, the last segment of the path is replaced

The following image shows examples of this:

	baseUri	relativeUri	Uri.Combine
1	http://services.odata.org/TripPinRESTierService/	/Airports	http://services.odata.org/Airports
2	http://services.odata.org/TripPinRESTierService/	Airports	http://services.odata.org/TripPinRESTierService/Airports
3	http://services.odata.org/TripPinRESTierService	Airports	http://services.odata.org/Airports

Remove the Entity Specific Functions

To make your connector easier to maintain, you'll remove the entity specific formatting functions you used in the previous lesson—`GetAirlineTables` and `GetAirportsTable`. Instead, you'll update `TripPin.Feed` to process the JSON response in a way that will work for all of your entities. Specifically, you take the `value` field of the returned OData JSON payload, and convert it from a list of records to a table.

```

TripPin.Feed = (url as text) =>
let
    source = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
    json = Json.Document(source),
    // The response is a JSON record - the data we want is a list of records in the "value" field
    value = json[value],
    asTable = Table.FromList(value, Splitter.SplitByNothing()),
    // expand all columns from the record
    fields = Record.FieldNames(Table.FirstValue(asTable, [Empty = null])),
    expandAll = Table.ExpandRecordColumn(asTable, "Column1", fields)
in
    expandAll;

```

NOTE

A disadvantage of using a generic approach to process your entities is that you lose the nice formating and type information for your entities. A later section in this tutorial shows how to enforce schema on REST API calls.

Conclusion

In this tutorial, you cleaned up and simplified your connector by fixing your Data Source Path value, and moving to a more flexible format for your navigation table. After completing these steps (or using the sample code in this directory), the `TripPin.Contents` function returns a navigation table in Power BI Desktop.

The screenshot shows the Power BI Navigator window. On the left, there's a sidebar with a search bar, a 'Display Options' dropdown, and a tree view under 'TripPin Data Source Paths [3]' showing 'Airlines', 'Airports' (which is selected and highlighted in grey), and 'People'. On the right, there's a table titled 'Airports' with four columns: 'Name', 'IcaoCode', 'IataCode', and 'Location'. The table contains five rows of data:

Name	IcaoCode	IataCode	Location
San Francisco International Airport	KSFO	SFO	Record
Los Angeles International Airport	KLAX	LAX	Record
Shanghai Hongqiao International Airport	ZSSS	SHA	Record
Beijing Capital International Airport	ZBAA	PEK	Record
John F. Kennedy International Airport	KJFK	JFK	Record

At the bottom right of the Navigator window, there are three buttons: 'Load', 'Edit', and 'Cancel'.

TripPin Part 5 - Paging

7 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Add paging support to the connector

Many Rest APIs will return data in "pages", requiring clients to make multiple requests to stitch the results together. Although there are some common conventions for pagination (such as [RFC 5988](#)), it generally varies from API to API. Thankfully, TripPin is an OData service, and the [OData standard](#) defines a way of doing pagination using [odata.nextLink](#) values returned in the body of the response.

To simplify [previous iterations](#) of the connector, the `TripPin.Feed` function was not *page aware*. It simply parsed whatever JSON was returned from the request and formatted it as a table. Those familiar with the OData protocol might have noticed that a number of incorrect assumptions were made on the [format of the response](#) (such as assuming there is a `value` field containing an array of records).

In this lesson you'll improve your response handling logic by making it page aware. Future tutorials will make the page handling logic more robust and able to handle multiple response formats (including errors from the service).

NOTE

You do not need to implement your own paging logic with connectors based on [OData.Feed](#), as it handles it all for you automatically.

Paging Checklist

When implementing paging support, you'll need to know the following things about your API:

- How do you request the next page of data?
- Does the paging mechanism involve calculating values, or do you extract the URL for the next page from the response?
- How do you know when to stop paging?
- Are there parameters related to paging that you should be aware of? (such as "page size")

The answer to these questions will impact the way you implement your paging logic. While there is some amount of code reuse across paging implementations (such as the use of [Table.GenerateByPage](#), most connectors will end up requiring custom logic.

NOTE

This lesson contains paging logic for an OData service, which follows a specific format. Check the documentation for your API to determine the changes you'll need to make in your connector to support its paging format.

Overview of OData Paging

OData paging is driven by [nextLink annotations](#) contained within the response payload. The nextLink value contains the URL to the next page of data. You'll know if there is another page of data by looking for an `odata.nextLink` field in outermost object in the response. If there's no `odata.nextLink` field, you've read all of your data.

```
{  
    "odata.context": "...",  
    "odata.count": 37,  
    "value": [  
        { },  
        { },  
        { }  
    ],  
    "odata.nextLink": "...?$skiptoken=342r89"  
}
```

Some OData services allow clients to supply a [max page size preference](#), but it is up to the service whether or not to honor it. Power Query should be able to handle responses of any size, so you don't need to worry about specifying a page size preference—you can support whatever the service throws at you.

More information about [Server-Driven Paging](#) can be found in the OData specification.

Testing TripPin

Before fixing your paging implementation, confirm the current behavior of the extension from the [previous tutorial](#). The following test query will retrieve the People table and add an index column to show your current row count.

```
let  
    source = TripPin.Contents(),  
    data = source{[Name="People"]}[Data],  
    withRowCount = Table.AddIndexColumn(data, "Index")  
in  
    withRowCount
```

Turn on fiddler, and run the query in Visual Studio. You'll notice that the query returns a table with 8 rows (index 0 to 7).

Query Result										
@odata.id	@odata.etag	@odata.editLink	UserName	FirstName	LastName	Emails	AddressInfo	Gender	Concurrency	Index
			russellwhyte	Russell	Whyte	[List]	[List]	Male	636348013653063483	0
			scottketchum	Scott	Ketchum	[List]	[List]	Male	636348013653063483	1
			ronaldmundy	Ronald	Mundy	[List]	[List]	Male	636348013653063483	2
			javieralfred	Javier	Alfred	[List]	[List]	Male	636348013653063483	3
			willieashmore	Willie	Ashmore	[List]	[List]	Male	636348013653063483	4
			vincentcalabrese	Vincent	Calabrese	[List]	[List]	Male	636348013653063483	5
			clydegues	Clyde	Guess	[List]	[List]	Male	636348013653063483	6
			keithpinckney	Keith	Pinckney	[List]	[List]	Male	636348013653063483	7

If you look at the body of the response from fiddler, you'll see that it does in fact contain an `@odata.nextLink` field, indicating that there are more pages of data available.

```
{
    "@odata.context": "https://services.odata.org/V4/TripPinService/$metadata#People",
    "@odata.nextLink": "https://services.odata.org/v4/TripPinService/People?%24skiptoken=8",
    "value": [
        { },
        { },
        { },
        { }
    ]
}
```

Implementing Paging for TripPin

You're now going to make the following changes to your extension:

1. Import the common `Table.GenerateByPage` function
2. Add a `GetAllPagesByNextLink` function that uses `Table.GenerateByPage` to glue all pages together
3. Add a `GetPage` function that can read a single page of data
4. Add a `GetNextLink` function to extract the next URL from the response
5. Update `TripPin.Feed` to use the new page reader functions

NOTE

As stated earlier in this tutorial, paging logic will vary between data sources. The implementation here tries to break up the logic into functions that should be reusable for sources that use *next links* returned in the response.

Table.GenerateByPage

The `Table.GenerateByPage` function can be used to efficiently combine multiple 'pages' of data into a single table. It does this by repeatedly calling the function passed in as the `getNextPage` parameter, until it receives a `null`. The function parameter must take a single argument, and return a `nullable table`.

```
getNextPage = (lastPage) as nullable table => ...
```

Each call to `getNextPage` receives the output from the previous call.

```
// The getNextPage function takes a single argument and is expected to return a nullable table
Table.GenerateByPage = (getNextPage as function) as table =>
    let
        listOfPages = List.Generate(
            () => getNextPage(null), // get the first page of data
            (lastPage) => lastPage <> null, // stop when the function returns null
            (lastPage) => getNextPage(lastPage) // pass the previous page to the next function call
        ),
        // concatenate the pages together
        tableOfPages = Table.FromList(listOfPages, Splitter.SplitByNothing(), {"Column1"}),
        firstRow = tableOfPages{0}?
    in
        // if we didn't get back any pages of data, return an empty table
        // otherwise set the table type based on the columns of the first page
        if (firstRow = null) then
            Table.FromRows({})
        else
            Value.ReplaceType(
                Table.ExpandTableColumn(tableOfPages, "Column1", Table.ColumnNames(firstRow[Column1])),
                Value.Type(firstRow[Column1])
            );
    
```

Some notes about `Table.GenerateByPage`:

- The `getNextPage` function will need to retrieve the next page URL (or page number, or whatever other values are used to implement the paging logic). This is generally done by adding `meta` values to the page before returning it.
- The columns and table type of the combined table (i.e. all pages together) are derived from the first page of data. The `getNextPage` function should normalize each page of data.
- The first call to `getNextPage` receives a null parameter.
- `getNextPage` must return null when there are no pages left.

Implementing GetAllPagesByNextLink

The body of your `GetAllPagesByNextLink` function implements the `getNextPage` function argument for `Table.GenerateByPage`. It will call the `GetPage` function, and retrieve the URL for the next page of data from the `NextLink` field of the `meta` record from the previous call.

```
// Read all pages of data.
// After every page, we check the "NextLink" record on the metadata of the previous request.
// Table.GenerateByPage will keep asking for more pages until we return null.
GetAllPagesByNextLink = (url as text) as table =>
    Table.GenerateByPage((previous) =>
        let
            // if previous is null, then this is our first page of data
            nextLink = if (previous = null) then url else Value.Metadata(previous)[NextLink]?,
            // if NextLink was set to null by the previous call, we know we have no more data
            page = if (nextLink <> null) then GetPage(nextLink) else null
        in
            page
    );

```

Implementing GetPage

Your `GetPage` function will use `Web.Contents` to retrieve a single page of data from the TripPin service, and convert the response into a table. It passes the response from `Web.Contents` to the `GetNextLink` function to extract the URL of the next page, and sets it on the `meta` record of the returned table (page of data).

This implementation is a slightly modified version of the `TripPin.Feed` call from the previous tutorials.

```
GetPage = (url as text) as table =>
    let
        response = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
        body = Json.Document(response),
        nextLink = GetNextLink(body),
        data = Table.FromRecords(body[value])
    in
        data meta [NextLink = nextLink];

```

Implementing GetNextLink

Your `GetNextLink` function simply checks the body of the response for an `@odata.nextLink` field, and returns its value.

```
// In this implementation, 'response' will be the parsed body of the response after the call to Json.Document.
// Look for the '@odata.nextLink' field and simply return null if it doesn't exist.
GetNextLink = (response) as nullable text => Record.FieldOrDefault(response, "@odata.nextLink");
```

Putting it all Together

The final step to implement your paging logic is to update `TripPin.Feed` to use the new functions. For now, you're simply calling through to `GetAllPagesByNextLink`, but in subsequent tutorials, you'll be adding new capabilities (such as enforcing a schema, and query parameter logic).

```
TripPin.Feed = (url as text) as table => GetAllPagesByNextLink(url);
```

If you re-run the same [test query](#) from earlier in the tutorial, you should now see the page reader in action. You should also see that you have 20 rows in the response rather than 8.

Query Result											
@odata.id	@odata.etag	@odata.editLink	UserName	FirstName	LastName	Emails	AddressInfo	Gender	Concurrency	Index	
			russellwhyte	Russell	Whyte	[List]	[List]	Male	636348013653063483	0	
			scottketchum	Scott	Ketchum	[List]	[List]	Male	636348013653063483	1	
			ronaldmundy	Ronald	Mundy	[List]	[List]	Male	636348013653063483	2	
			javieralfred	Javier	Alfred	[List]	[List]	Male	636348013653063483	3	
			willieashmore	Willie	Ashmore	[List]	[List]	Male	636348013653063483	4	
			vincentcalabrese	Vincent	Calabrese	[List]	[List]	Male	636348013653063483	5	
			clydegueess	Clyde	Guess	[List]	[List]	Male	636348013653063483	6	
			keithpinckney	Keith	Pinckney	[List]	[List]	Male	636348013653063483	7	
			marshallgaray	Marshall	Garay	[List]	[List]	Male	636348013653063483	8	
			ryantheriault	Ryan	Theriault	[List]	[List]	Male	636348013653063483	9	
			elainestewart	Elaine	Stewart	[List]	[List]	Female	636348013653063483	10	
			salliesampson	Sallie	Sampson	[List]	[List]	Female	636348013653063483	11	
			jonirosales	Joni	Rosales	[List]	[List]	Female	636348013653063483	12	
			georginabarlow	Georgina	Barlow	[List]	[List]	Female	636348013653063483	13	
			angelhuffman	Angel	Huffman	[List]	[List]	Female	636348013653063483	14	
			laurelosborn	Laurel	Osborn	[List]	[List]	Female	636348013653063483	15	
			sandyosborn	Sandy	Osborn	[List]	[List]	Female	636348013653063483	16	
			ursulabright	Ursula	Bright	[List]	[List]	Female	636348013653063483	17	
			genevievereeves	Genevieve	Reeves	[List]	[List]	Female	636348013653063483	18	
			kristakemp	Krista	Kemp	[List]	[List]	Female	636348013653063483	19	

If you look at the requests in fiddler, you should now see separate requests for each page of data.

#	Overall_Elapsed	Result	Protocol	Host	URL
{S1} 1	0:00:00.114	200	HTTP	services.odata.org	/v4/TripPinService/People
{S1} 2	0:00:00.366	200	HTTP	services.odata.org	/v4/TripPinService/People
{S1} 3	0:00:00.038	200	HTTP	services.odata.org	/v4/TripPinService/People?%24skiptoken=8
{S1} 4	0:00:00.043	200	HTTP	services.odata.org	/v4/TripPinService/People?%24skiptoken=16

NOTE

You'll notice duplicate requests for the first page of data from the service, which is not ideal. The extra request is a result of the M engine's schema checking behavior. Ignore this issue for now and resolve it in the [next tutorial](#), where you'll apply an explicit schema.

Conclusion

This lesson showed you how to implement pagination support for a Rest API. While the logic will likely vary between APIs, the pattern established here should be reusable with minor modifications.

In the next lesson, you'll look at how to apply an explicit schema to your data, going beyond the simple `text` and `number` data types you get from `Json.Document`.

TripPin Part 6 - Schema

11 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Define a fixed schema for a REST API
- Dynamically set data types for columns
- Enforce a table structure to avoid transformation errors due to missing columns
- Hide columns from the result set

One of the big advantages of an OData service over a standard REST API is its [\\$metadata definition](#). The \$metadata document describes the data found on this service, including the schema for all of its Entities (Tables) and Fields (Columns). The `OData.Feed` function uses this schema definition to automatically set data type information—so instead of getting all text and number fields (like you would from `Json.Document`), end users will get dates, whole numbers, times, and so on, providing a better overall user experience.

Many REST APIs don't have a way to programmatically determine their schema. In these cases, you'll need to include schema definitions within your connector. In this lesson you'll define a simple, hardcoded schema for each of your tables, and enforce the schema on the data you read from the service.

NOTE

The approach described here should work for many REST services. [Future lessons](#) will build upon this approach by recursively enforcing schemas on structured columns (record, list, table), and provide sample implementations that can programmatically generate a schema table from CSDL or [JSON Schema](#) documents.

Overall, enforcing a schema on the data returned by your connector has multiple benefits, such as:

- Setting the correct data types
- Removing columns that don't need to be shown to end users (such as internal IDs or state information)
- Ensuring that each page of data has the same shape by adding any columns that might be missing from a response (a common way for REST APIs to indicate a field should be null)

Viewing the Existing Schema with Table.Schema

The connector created in the [previous lesson](#) displays three tables from the TripPin service—`Airlines`, `Airports`, and `People`. Run the following query to view the `Airlines` table:

```
let
    source = TripPin.Contents(),
    data = source{[Name="Airlines"]}[Data]
in
    data
```

In the results you'll see four columns returned:

- `@odata.id`

- @odata.editLink
- AirlineCode
- Name

@odata.id	@odata.editLink	AirlineCode	Name
		AA	American Airlines
		FM	Shanghai Airline
		MU	China Eastern Airlines
		AF	Air France
		AZ	Alitalia
		AC	Air Canada
		OS	Austrian Airlines
		TK	Turkish Airlines
		JL	Japan Airlines
		SQ	Singapore Airlines
		KE	Korean Air
		CZ	China Southern
		AK	AirAsia
		HX	Hong Kong Airlines
		EK	Emirates

The "@odata.*" columns are part of OData protocol, and not something you'd want or need to show to the end users of your connector. `AirlineCode` and `Name` are the two columns you'll want to keep. If you look at the schema of the table (using the handy [Table.Schema](#) function), you can see that all of the columns in the table have a data type of `Any.Type`.

```
let
  source = TripPin.Contents(),
  data = source{[Name="Airlines"]}[Data]
in
  Table.Schema(data)
```

Name	Position	TypeName	Kind	IsNullable
@odata.id	0	Any.Type	any	<input checked="" type="checkbox"/>
@odata.editLink	1	Any.Type	any	<input checked="" type="checkbox"/>
AirlineCode	2	Any.Type	any	<input checked="" type="checkbox"/>
Name	3	Any.Type	any	<input checked="" type="checkbox"/>
				<input type="checkbox"/>

[Table.Schema](#) returns a lot of metadata about the columns in a table, including names, positions, type information, and many advanced properties, such as Precision, Scale, and MaxLength. Future lessons will provide design patterns for setting these advanced properties, but for now you need only concern yourself with the ascribed type (`TypeName`), primitive type (`Kind`), and whether the column value might be null (`IsNullable`).

Defining a Simple Schema Table

Your schema table will be composed of two columns:

COLUMN	DETAILS
Name	The name of the column. This must match the name in the results returned by the service.
Type	The M data type you're going to set. This can be a primitive type (<code>text</code> , <code>number</code> , <code>datetime</code> , and so on), or an ascribed type (<code>Int64.Type</code> , <code>Currency.Type</code> , and so on).

The hardcoded schema table for the `Airlines` table will set its `AirlineCode` and `Name` columns to `text`, and looks like this:

```
Airlines = #table({"Name", "Type"}, {  
    {"AirlineCode", type text},  
    {"Name", type text}  
});
```

The `Airports` table has four fields you'll want to keep (including one of type `record`):

```
Airports = #table({"Name", "Type"}, {  
    {"IcaoCode", type text},  
    {"Name", type text},  
    {"IataCode", type text},  
    {"Location", type record}  
});
```

Finally, the `People` table has seven fields, including lists (`Emails`, `AddressInfo`), a *nullable* column (`Gender`), and a column with an *ascribed type* (`Concurrency`).

```
People = #table({"Name", "Type"}, {  
    {"UserName", type text},  
    {"FirstName", type text},  
    {"LastName", type text},  
    {"Emails", type list},  
    {"AddressInfo", type list},  
    {"Gender", type nullable text},  
    {"Concurrency", Int64.Type}  
})
```

The SchemaTransformTable Helper Function

The `SchemaTransformTable` helper function described below will be used to enforce schemas on your data. It takes the following parameters:

PARAMETER	TYPE	DESCRIPTION
table	table	The table of data you'll want to enforce your schema on.
schema	table	The schema table to read column information from, with the following type: <pre>type table [Name = text, Type = type]</pre>

PARAMETER	TYPE	DESCRIPTION
enforceSchema	number	<p>(optional) An enum that controls behavior of the function.</p> <p>The default value (<code>EnforceSchema.Strict = 1</code>) ensures that the output table will match the schema table that was provided by adding any missing columns, and removing extra columns.</p> <p>The <code>EnforceSchema.IgnoreExtraColumns = 2</code> option can be used to preserve extra columns in the result.</p> <p>When <code>EnforceSchema.IgnoreMissingColumns = 3</code> is used, both missing columns and extra columns will be ignored.</p>

The logic for this function looks something like this:

1. Determine if there are any missing columns from the source table.
2. Determine if there are any extra columns.
3. Ignore structured columns (of type `list`, `record`, and `table`), and columns set to `type any`.
4. Use `Table.TransformColumnTypes` to set each column type.
5. Reorder columns based on the order they appear in the schema table.
6. Set the type on the table itself using `Value.ReplaceType`.

NOTE

The last step to set the table type will remove the need for the Power Query UI to infer type information when viewing the results in the query editor. This removes the double request issue you saw at the [end of the previous tutorial](#).

The following helper code can be copy and pasted into your extension:

```

EnforceSchema.Strict = 1;           // Add any missing columns, remove extra columns, set table type
EnforceSchema.IgnoreExtraColumns = 2; // Add missing columns, do not remove extra columns
EnforceSchema.IgnoreMissingColumns = 3; // Do not add or remove columns

SchemaTransformTable = (table as table, schema as table, optional enforceSchema as number) as table =>
    let
        // Default to EnforceSchema.Strict
        _enforceSchema = if (enforceSchema <> null) then enforceSchema else EnforceSchema.Strict,

        // Applies type transforms to a given table
        EnforceTypes = (table as table, schema as table) as table =>
            let
                map = (t) => if Type.Is(t, type list) or Type.Is(t, type record) or t = type any then null
            else t,
                mapped = Table.TransformColumns(schema, {"Type", map}),
                omitted = Table.SelectRows(mapped, each [Type] <> null),
                existingColumns = Table.ColumnNames(table),
                removeMissing = Table.SelectRows(omitted, each List.Contains(existingColumns, [Name])),
                primitiveTransforms = Table.ToRows(removeMissing),
                changedPrimitives = Table.TransformColumnTypes(table, primitiveTransforms)
            in
                changedPrimitives,

            // Returns the table type for a given schema
            SchemaToTableType = (schema as table) as type =>
                let
                    toList = List.Transform(schema[Type], (t) => [Type=t, Optional=false]),
                    toRecord = Record.FromList(toList, schema[Name]),
                    toType = Type.ForRecord(toRecord, false)
                in
                    type table (toType),

                // Determine if we have extra/missing columns.
                // The enforceSchema parameter determines what we do about them.
                schemaNames = schema[Name],
                foundNames = Table.ColumnNames(table),
                addNames = List.RemoveItems(schemaNames, foundNames),
                extraNames = List.RemoveItems(foundNames, schemaNames),
                tmp = Text.NewGuid(),
                added = Table.AddColumn(table, tmp, each []),
                expanded = Table.ExpandRecordColumn(added, tmp, addNames),
                result = if List.IsEmpty(addNames) then table else expanded,
                fullList =
                    if (_enforceSchema = EnforceSchema.Strict) then
                        schemaNames
                    else if (_enforceSchema = EnforceSchema.IgnoreMissingColumns) then
                        foundNames
                    else
                        schemaNames & extraNames,

                // Select the final list of columns.
                // These will be ordered according to the schema table.
                reordered = Table.SelectColumns(result, fullList, MissingField.Ignore),
                enforcedTypes = EnforceTypes(reordered, schema),
                withType = if (_enforceSchema = EnforceSchema.Strict) then Value.ReplaceType(enforcedTypes,
SchemaToTableType(schema)) else enforcedTypes
            in
                withType;

```

Updating the TripPin Connector

You'll now make the following changes to your connector to make use of the new schema enforcement code.

1. Define a master schema table (`SchemaTable`) that holds all of your schema definitions.
2. Update the `TripPin.Feed`, `GetPage`, and `GetAllPagesByNextLink` to accept a `schema` parameter.

3. Enforce your schema in `GetPage`.
4. Update your navigation table code to wrap each table with a call to a new function (`GetEntity`)—this will give you more flexibility to manipulate the table definitions in the future.

Master schema table

You'll now consolidate your schema definitions into a single table, and add a helper function (`GetSchemaForEntity`) that lets you look up the definition based on an entity name (for example, `GetSchemaForEntity("Airlines")`)

```
SchemaTable = #table({"Entity", "SchemaTable"}, {
    {"Airlines", #table({"Name", "Type"}, {
        {"AirlineCode", type text},
        {"Name", type text}
    )}),
    {"Airports", #table({"Name", "Type"}, {
        {"IcaoCode", type text},
        {"Name", type text},
        {"IataCode", type text},
        {"Location", type record}
    )}),
    {"People", #table({"Name", "Type"}, {
        {"UserName", type text},
        {"FirstName", type text},
        {"LastName", type text},
        {"Emails", type list},
        {"AddressInfo", type list},
        {"Gender", type nullable text},
        {"Concurrency", Int64.Type}
    )})
});

GetSchemaForEntity = (entity as text) as table => try SchemaTable{[Entity=entity]}[SchemaTable] otherwise
error "Couldn't find entity: '" & entity & "'";
```

Adding schema support to data functions

You'll now add an optional `schema` parameter to the `TripPin.Feed`, `GetPage`, and `GetAllPagesByNextLink` functions. This will allow you to pass down the schema (when you want to) to the paging functions, where it will be applied to the results you get back from the service.

```
TripPin.Feed = (url as text, optional schema as table) as table => ...
GetPage = (url as text, optional schema as table) as table => ...
GetAllPagesByNextLink = (url as text, optional schema as table) as table => ...
```

You'll also update all of the calls to these functions to make sure that you pass the schema through correctly.

Enforcing the schema

The actual schema enforcement will be done in your `GetPage` function.

```
GetPage = (url as text, optional schema as table) as table =>
let
    response = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
    body = Json.Document(response),
    nextLink = GetNextLink(body),
    data = Table.FromRecords(body[value]),
    // enforce the schema
    withSchema = if (schema <> null) then SchemaTransformTable(data, schema) else data
in
    withSchema meta [NextLink = nextLink];
```

[Note] This `GetPage` implementation uses `Table.FromRecords` to convert the list of records in the JSON response to a table. A major downside to using `Table.FromRecords` is that it assumes all records in the list have the same set of fields. This works for the TripPin service, since the OData records are guaranteed to contain the same fields, but this might not be the case for all REST APIs. A more robust implementation would use a combination of `Table.FromList` and `Table.ExpandRecordColumn`. Later tutorials will change the implementation to get the column list from the schema table, ensuring that no columns are lost or missing during the JSON to M translation.

Adding the GetEntity function

The `GetEntity` function will wrap your call to `TripPin.Feed`. It will look up a schema definition based on the entity name, and build the full request URL.

```
GetEntity = (url as text, entity as text) as table =>
    let
        fullUrl = Uri.Combine(url, entity),
        schemaTable = GetSchemaForEntity(entity),
        result = TripPin.Feed(fullUrl, schemaTable)
    in
        result;
```

You'll then update your `TripPinNavTable` function to call `GetEntity`, rather than making all of the calls inline. The main advantage to this is that it will let you continue modifying your entity building code, without having to touch your nav table logic.

```
TripPinNavTable = (url as text) as table =>
    let
        entitiesAsTable = Table.FromList(RootEntities, Splitter.SplitByNothing()),
        rename = Table.RenameColumns(entitiesAsTable, {{"Column1", "Name"}}),
        // Add Data as a calculated column
        withData = Table.AddColumn(rename, "Data", each GetEntity(url, [Name]), type table),
        // Add ItemKind and ItemName as fixed text values
        withItemKind = Table.AddColumn(withData, "ItemKind", each "Table", type text),
        withItemName = Table.AddColumn(withItemKind, "ItemName", each "Table", type text),
        // Indicate that the node should not be expandable
        withIsLeaf = Table.AddColumn(withItemName, "IsLeaf", each true, type logical),
        // Generate the nav table
        navTable = Table.ToNavigationTable(withIsLeaf, {"Name"}, "Name", "Data", "ItemKind", "ItemName",
        "IsLeaf")
    in
        navTable;
```

Putting it all Together

Once all of the code changes are made, compile and re-run the test query that calls `Table.Schema` for the Airlines table.

```
let
    source = TripPin.Contents(),
    data = source{[Name="Airlines"]}[Data]
in
    Table.Schema(data)
```

You now see that your Airlines table only has the two columns you defined in its schema:

Name	Position	TypeName	Kind	IsNullable
AirlineCode	0	Text.Type	text	<input type="checkbox"/>
Name	1	Text.Type	text	<input type="checkbox"/>
				<input type="checkbox"/>

If you run the same code against the People table...

```
let
    source = TripPin.Contents(),
    data = source{[Name="People"]}[Data]
in
    Table.Schema(data)
```

You'll see that the ascribed type you used (`Int64.Type`) was also set correctly.

Name	Position	TypeName	Kind	IsNullable
UserName	0	Text.Type	text	<input type="checkbox"/>
FirstName	1	Text.Type	text	<input type="checkbox"/>
LastName	2	Text.Type	text	<input type="checkbox"/>
Emails	3	List.Type	list	<input type="checkbox"/>
AddressInfo	4	List.Type	list	<input type="checkbox"/>
Gender	5	Text.Type	text	<input checked="" type="checkbox"/>
Concurrency	6	Int64.Type	number	<input type="checkbox"/>
				<input type="checkbox"/>

An important thing to note is that this implementation of `SchemaTransformTable` doesn't modify the types of `list` and `record` columns, but the `Emails` and `AddressInfo` columns are still typed as `list`. This is because `Json.Document` will correctly map JSON arrays to M lists, and JSON objects to M records. If you were to expand the list or record column in Power Query, you'd see that all of the expanded columns will be of type any. Future tutorials will improve the implementation to recursively set type information for nested complex types.

Conclusion

This tutorial provided a sample implementation for enforcing a schema on JSON data returned from a REST service. While this sample uses a simple hardcoded schema table format, the approach could be expanded upon by dynamically building a schema table definition from another source, such as a JSON schema file, or metadata service/endpoint exposed by the data source.

In addition to modifying column types (and values), your code is also setting the correct type information on the table itself. Setting this type information benefits performance when running inside of Power Query, as the user experience always attempts to infer type information to display the right UI queues to the end user, and the inference calls can end up triggering additional calls to the underlying data APIs.

If you view the People table using the [TripPin connector from the previous lesson](#), you'll see that all of the columns have a 'type any' icon (even the columns that contain lists):

= Source{[Name="People"]}[Data]

		ABC 123 @odata.editLink	ABC 123 UserName	ABC 123 FirstName	ABC 123 LastName	ABC 123 Emails	ABC 123 AddressInfo	ABC 123 Gender	ABC 123 Concurrency
1	8951"	http://services.odata.org/V4/TripPinService/People('russellwhyte')	russellwhyte	Russell	Whyte	List	List	Male	6.36389E+17
2	8951"	http://services.odata.org/V4/TripPinService/People('scottketchum')	scottketchum	Scott	Ketchum	List	List	Male	6.36389E+17
3	8951"	http://services.odata.org/V4/TripPinService/People('ronaldmundy')	ronaldmundy	Ronald	Mundy	List	List	Male	6.36389E+17
4	8951"	http://services.odata.org/V4/TripPinService/People('javieralfred')	javieralfred	Javier	Alfred	List	List	Male	6.36389E+17
5	8951"	http://services.odata.org/V4/TripPinService/People('willieashmore')	willieashmore	Willie	Ashmore	List	List	Male	6.36389E+17
6	8951"	http://services.odata.org/V4/TripPinService/People('vincentcalabrese')	vincentcalabrese	Vincent	Calabrese	List	List	Male	6.36389E+17
7	8951"	http://services.odata.org/V4/TripPinService/People('clydeguess')	clydeguess	Clyde	Guess	List	List	Male	6.36389E+17
8	8951"	http://services.odata.org/V4/TripPinService/People('keithpinckney')	keithpinckney	Keith	Pinckney	List	List	Male	6.36389E+17
9	8951"	http://services.odata.org/V4/TripPinService/People('marshallgaray')	marshallgaray	Marshall	Garay	List	List	Male	6.36389E+17
10	8951"	http://services.odata.org/V4/TripPinService/People('ryantheriault')	ryantheriault	Ryan	Theriault	List	List	Male	6.36389E+17
11	8951"	http://services.odata.org/V4/TripPinService/People('elainestewart')	elainestewart	Elaine	Stewart	List	List	Female	6.36389E+17
12	8951"	http://services.odata.org/V4/TripPinService/People('salliesampson')	salliesampson	Sallie	Sampson	List	List	Female	6.36389E+17
13	8951"	http://services.odata.org/V4/TripPinService/People('jonirosales')	jonirosales	Joni	Rosales	List	List	Female	6.36389E+17
14	8951"	http://services.odata.org/V4/TripPinService/People('georginabarlow')	georginabarlow	Georgina	Barlow	List	List	Female	6.36389E+17
15	8951"	http://services.odata.org/V4/TripPinService/People('angelhuffman')	angelhuffman	Angel	Huffman	List	List	Female	6.36389E+17
16	8951"	http://services.odata.org/V4/TripPinService/People('laurelosborn')	laurelosborn	Laurel	Osborn	List	List	Female	6.36389E+17
17	8951"	http://services.odata.org/V4/TripPinService/People('sandyosborn')	sandyosborn	Sandy	Osborn	List	List	Female	6.36389E+17
18	8951"	http://services.odata.org/V4/TripPinService/People('ursulabright')	ursulabright	Ursula	Bright	List	List	Female	6.36389E+17
19	8951"	http://services.odata.org/V4/TripPinService/People('genevievereeves')	genevievereeves	Genevieve	Reeves	List	List	Female	6.36389E+17
20	8951"	http://services.odata.org/V4/TripPinService/People('kristakemp')	kristakemp	Krista	Kemp	List	List	Female	6.36389E+17

Running the same query with the TripPin connector from this lesson, you'll now see that the type information is displayed correctly.

= Source{[Name="People"]}[Data]

	ABC 123 UserName	ABC 123 FirstName	ABC 123 LastName	Emails	AddressInfo	ABC 123 Gender	ABC 123 Concurrency
1	russellwhyte	Russell	Whyte	List	List	Male	6.36389E+17
2	scottketchum	Scott	Ketchum	List	List	Male	6.36389E+17
3	ronaldmundy	Ronald	Mundy	List	List	Male	6.36389E+17
4	javieralfred	Javier	Alfred	List	List	Male	6.36389E+17
5	willieashmore	Willie	Ashmore	List	List	Male	6.36389E+17
6	vincentcalabrese	Vincent	Calabrese	List	List	Male	6.36389E+17
7	clydeguess	Clyde	Guess	List	List	Male	6.36389E+17
8	keithpinckney	Keith	Pinckney	List	List	Male	6.36389E+17
9	marshallgaray	Marshall	Garay	List	List	Male	6.36389E+17
10	ryantheriault	Ryan	Theriault	List	List	Male	6.36389E+17
11	elainestewart	Elaine	Stewart	List	List	Female	6.36389E+17
12	salliesampson	Sallie	Sampson	List	List	Female	6.36389E+17
13	jonirosales	Joni	Rosales	List	List	Female	6.36389E+17
14	georginabarlow	Georgina	Barlow	List	List	Female	6.36389E+17
15	angelhuffman	Angel	Huffman	List	List	Female	6.36389E+17
16	laurelosborn	Laurel	Osborn	List	List	Female	6.36389E+17
17	sandyosborn	Sandy	Osborn	List	List	Female	6.36389E+17
18	ursulabright	Ursula	Bright	List	List	Female	6.36389E+17
19	genevievereeves	Genevieve	Reeves	List	List	Female	6.36389E+17
20	kristakemp	Krista	Kemp	List	List	Female	6.36389E+17

TripPin Part 7 - Advanced Schema with M Types

7 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Enforce a table schema using [M Types](#)
- Set types for nested records and lists
- Refactor code for reuse and unit testing

In the previous lesson you defined your table schemas using a simple "Schema Table" system. This schema table approach works for many REST APIs/Data Connectors, but services that return complete or deeply nested data sets might benefit from the approach in this tutorial, which leverages the [M type system](#).

This lesson will guide you through the following steps:

1. Adding unit tests
2. Defining custom M types
3. Enforcing a schema using types
4. Refactoring common code into separate files

Adding Unit Tests

Before you start making use of the advanced schema logic, you'll add a set of unit tests to your connector to reduce the chance of inadvertently breaking something. Unit testing works like this:

1. Copy the common code from the [UnitTest sample](#) into your `TripPin.query.pq` file
2. Add a section declaration to the top of your `TripPin.query.pq` file
3. Create a *shared* record (called `TripPin.UnitTesting`)
4. Define a `Fact` for each test
5. Call `Facts.Summarize()` to run all of the tests
6. Reference the previous call as the shared value to ensure that it gets evaluated when the project is run in Visual Studio

```

section TripPinUnitTests;

shared TripPin.UnitTesting =
[
    // Put any common variables here if you only want them to be evaluated once
    RootTable = TripPin.Contents(),
    Airlines = RootTable{[Name="Airlines"]}[Data],
    Airports = RootTable{[Name="Airports"]}[Data],
    People = RootTable{[Name="People"]}[Data],

    // Fact(<Name of the Test>, <Expected Value>, <Actual Value>)
    // <Expected Value> and <Actual Value> can be a literal or let statement
    facts =
    {
        Fact("Check that we have three entries in our nav table", 3, Table.RowCount(RootTable)),
        Fact("We have Airline data?", true, not Table.IsEmpty(Airlines)),
        Fact("We have People data?", true, not Table.IsEmpty(People)),
        Fact("We have Airport data?", true, not Table.IsEmpty(Airports)),
        Fact("Airlines only has 2 columns", 2, List.Count(Table.ColumnNames(Airlines))),
        Fact("Airline table has the right fields",
            {"AirlineCode", "Name"},
            Record.FieldNames(Type.RecordFields(Type.TableRow(Value.Type(Airlines)))))
    }
},
report = Facts.Summarize(facts)
][report];

```

Clicking run on the project will evaluate all of the Facts, and give you a report output that looks like this:

Query Result		
Result	Notes	Details
Success	All 6 Passed !!! ✓	100% success rate
Success ✓	Check that we have three entries in our nav table	(3 = 3)
Success ✓	We have Airline data?	(true = true)
Success ✓	We have People data?	(true = true)
Success ✓	We have Airport data?	(true = true)
Success ✓	Airlines only has 2 columns	(2 = 2)
Success ✓	Airline table has the right fields	({"AirlineCode", "Name"} = {"AirlineCode", "Name"})

Using some principles from [test-driven development](#), you'll now add a test that currently fails, but will soon be reimplemented and fixed (by the end of this tutorial). Specifically, you'll add a test that checks one of the nested records (Emails) you get back in the People entity.

```
Fact("Emails is properly typed", type text, Type.ListItem(Value.Type(People{0}[Emails])))
```

If you run the code again, you should now see that you have a failing test.

M Query Output

Output Log Errors Credentials

Query Result

Result	Notes	Details
6 Passed	6 Passed 1 Failed	85% success rate
Success ✓	Check that we have three entries in our nav table	(3 = 3)
Success ✓	We have Airline data?	(true = true)
Success ✓	We have People data?	(true = true)
Success ✓	We have Airport data?	(true = true)
Success ✓	Airlines only has 2 columns	(2 = 2)
Success ✓	Airline table has the right fields	{"AirlineCode", "Name"} = {"AirlineCode", "Name"})
Failure ⚠	Emails is properly typed	(type text <> type any)

Now you just need to implement the functionality to make this work.

Defining Custom M Types

The schema enforcement approach in the [previous lesson](#) used "schema tables" defined as Name/Type pairs. It works well when working with flattened/relational data, but didn't support setting types on nested records/lists, or allow you to reuse type definitions across tables/entities.

In the TripPin case, the data in the People and Airports entities contain structured columns, and even share a type (`Location`) for representing address information. Rather than defining Name/Type pairs in a schema table, you'll define each of these entities using custom M type declarations.

Here is a quick refresher about types in the M language from the [Language Specification](#):

A **type value** is a value that **classifies** other values. A value that is classified by a type is said to **conform** to that type. The M type system consists of the following kinds of types:

- Primitive types, which classify primitive values (`binary`, `date`, `datetime`, `datetimezone`, `duration`, `list`, `logical`, `null`, `number`, `record`, `text`, `time`, `type`) and also include a number of abstract types (`function`, `table`, `any`, and `none`)
- Record types, which classify record values based on field names and value types
- List types, which classify lists using a single item base type
- Function types, which classify function values based on the types of their parameters and return values
- Table types, which classify table values based on column names, column types, and keys
- Nullable types, which classifies the value null in addition to all the values classified by a base type
- Type types, which classify values that are types

Using the raw JSON output you get (and/or looking up the definitions in the service's `$metadata`), you can define the following record types to represent OData complex types:

```

LocationType = type [
    Address = text,
    City = CityType,
    Loc = LocType
];

CityType = type [
    CountryRegion = text,
    Name = text,
    Region = text
];

LocType = type [
    #"type" = text,
    coordinates = {number},
    crs = CrsType
];

CrsType = type [
    #"type" = text,
    properties = record
];

```

Note how the `LocationType` references the `CityType` and `LocType` to represent its structured columns.

For the top level entities (that you want represented as Tables), you define *table types*:

```

AirlinesType = type table [
    AirlineCode = text,
    Name = text
];

AirportsType = type table [
    Name = text,
    IataCode = text,
    Location = LocationType
];

PeopleType = type table [
    UserName = text,
    FirstName = text,
    LastName = text,
    Emails = {text},
    AddressInfo = {nullable LocationType},
    Gender = nullable text,
    Concurrency = Int64.Type
];

```

You then update your `SchemaTable` variable (which you use as a "lookup table" for entity to type mappings) to use these new type definitions:

```

SchemaTable = #table({"Entity", "Type"}, {
    {"Airlines", AirlinesType },
    {"Airports", AirportsType },
    {"People", PeopleType}
});

```

Enforcing a Schema Using Types

You'll rely on a common function (`Table.ChangeType`) to enforce a schema on your data, much like you used `SchemaTransformTable` in the [previous lesson](#). Unlike `SchemaTransformTable`, `Table.ChangeType` takes in an actual M

table type as an argument, and will apply your schema *recursively* for all nested types. It's signature looks like this:

```
Table.ChangeType = (table, tableType as type) as nullable table => ...
```

The full code listing for the `Table.ChangeType` function can be found in the [Table.ChangeType.pqm](#) file.

NOTE

For flexibility, the function can be used on tables, as well as lists of records (which is how tables would be represented in a JSON document).

You then need to update the connector code to change the `schema` parameter from a `table` to a `type`, and add a call to `Table.ChangeType` in `GetEntity`.

```
GetEntity = (url as text, entity as text) as table =>
    let
        fullUrl = Uri.Combine(url, entity),
        schema = GetSchemaForEntity(entity),
        result = TripPin.Feed(fullUrl, schema),
        appliedSchema = Table.ChangeType(result, schema)
    in
        appliedSchema;
```

`GetPage` is updated to use the list of fields from the schema (to know the names of what to expand when you get the results), but leaves the actual schema enforcement to `GetEntity`.

```
GetPage = (url as text, optional schema as type) as table =>
    let
        response = Web.Contents(url, [ Headers = DefaultRequestHeaders ]),
        body = Json.Document(response),
        nextLink = GetNextLink(body),

        // If we have no schema, use Table.FromRecords() instead
        // (and hope that our results all have the same fields).
        // If we have a schema, expand the record using its field names
        data =
            if (schema <> null) then
                Table.FromRecords(body[value])
            else
                let
                    // convert the list of records into a table (single column of records)
                    asTable = Table.FromList(body[value], Splitter.SplitByNothing(), {"Column1"}),
                    fields = Record.FieldNames(Type.RecordFields(Type.TableRow(schema))),
                    expanded = Table.ExpandRecordColumn(asTable, fields)
                in
                    expanded
    in
        data meta [NextLink = nextLink];
```

Confirming that nested types are being set

The definition for your `PeopleType` now sets the `Emails` field to a list of text (`{text}`). If you're applying the types correctly, the call to `Type.ListItem` in your unit test should now be returning `type text` rather than `type any`.

Running your unit tests again show that they are now all passing.

Query Result		
Result	Notes	Details
Success	All 7 Passed !!! ✓	100% success rate
Success ✓	Check that we have three entries in our nav table	(3 = 3)
Success ✓	We have Airline data?	(true = true)
Success ✓	We have People data?	(true = true)
Success ✓	We have Airport data?	(true = true)
Success ✓	Airlines only has 2 columns	(2 = 2)
Success ✓	Airline table has the right fields	{"AirlineCode", "Name"} = {"AirlineCode", "Name"})
Success ✓	Emails is properly typed	(type text = type text)

Refactoring Common Code into Separate Files

NOTE

The M engine will have improved support for referencing external modules/common code in the future, but this approach should carry you through until then.

At this point, your extension almost has as much "common" code as TripPin connector code. In the future these **common functions** will either be part of the built-in standard function library, or you'll be able to reference them from another extension. For now, you refactor your code in the following way:

1. Move the reusable functions to separate files (.pqm).
2. Set the **Build Action** property on the file to **Compile** to make sure it gets included in your extension file during the build.
3. Define a function to load the code using [Expression.Evaluate](#).
4. Load each of the common functions you want to use.

The code to do this is included in the snippet below:

```
Extension.LoadFunction = (name as text) =>
    let
        binary = Extension.Contents(name),
        asText = Text.FromBinary(binary)
    in
        Expression.Evaluate(asText, #shared);

Table.ChangeType = Extension.LoadFunction("Table.ChangeType.pqm");
Table.GenerateByPage = Extension.LoadFunction("Table.GenerateByPage.pqm");
Table.ToNavigationTable = Extension.LoadFunction("Table.ToNavigationTable.pqm");
```

Conclusion

This tutorial made a number of improvements to the way you enforce a schema on the data you get from a REST API. The connector is currently hard coding its schema information, which has a performance benefit at runtime, but is unable to adapt to changes in the service's metadata overtime. Future tutorials will move to a purely dynamic approach that will infer the schema from the service's \$metadata document.

In addition to the schema changes, this tutorial added Unit Tests for your code, and refactored the common helper functions into separate files to improve overall readability.

TripPin Part 8 - Adding Diagnostics

7 minutes to read • [Edit Online](#)

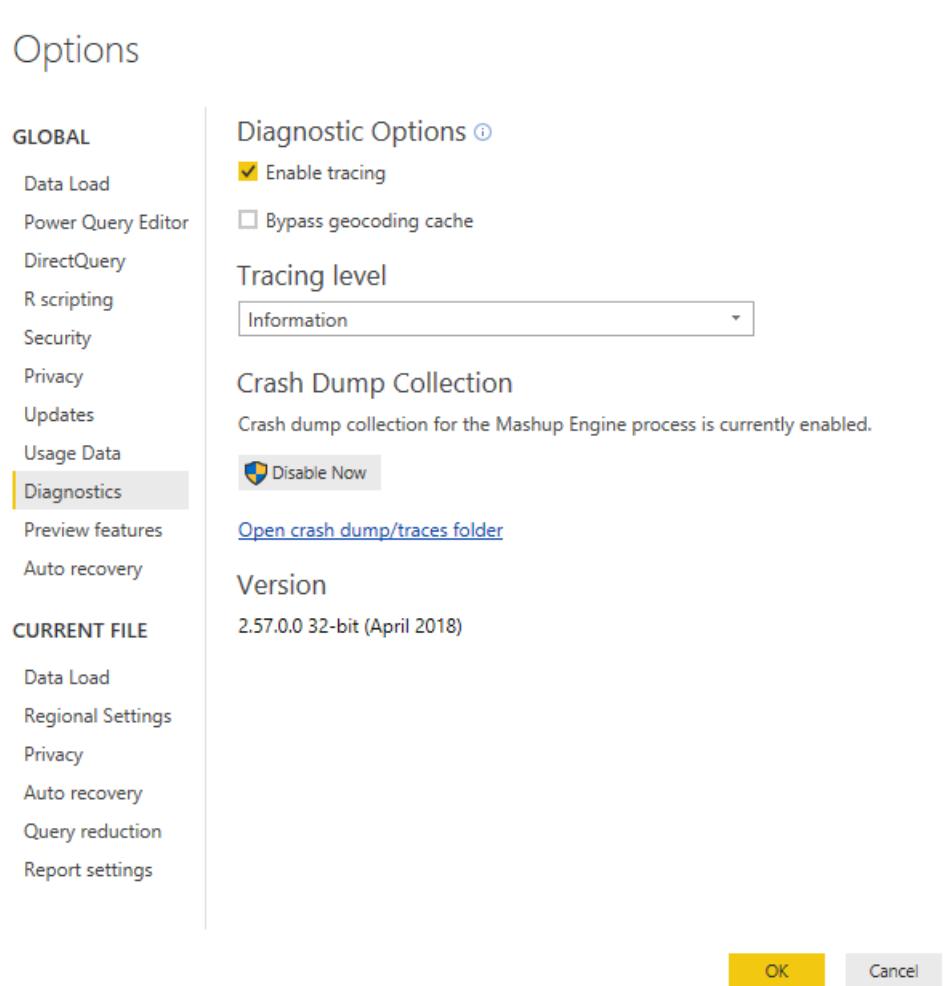
This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Learn about the `Diagnostics.Trace` function
- Use the Diagnostics helper functions to add trace information to help debug your connector

Enabling diagnostics

Power Query users can enable trace logging by selecting the checkbox under **Options | Diagnostics**.

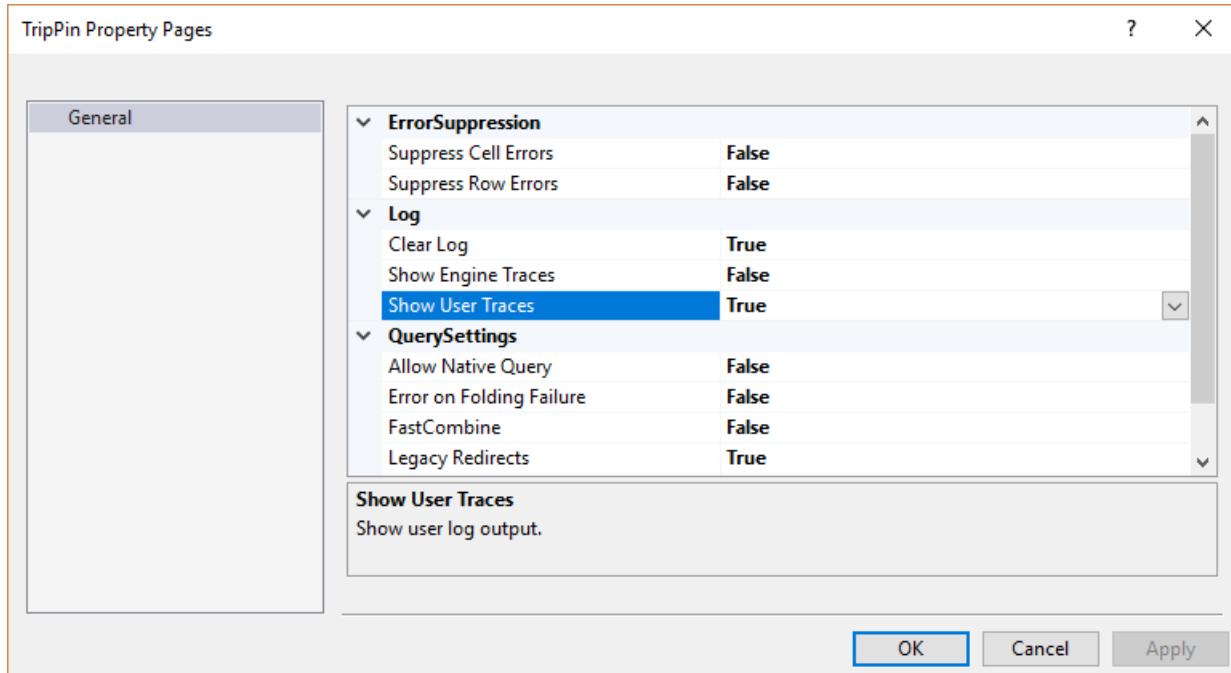


Once enabled, any subsequent queries will cause the M engine to emit trace information to log files located in a fixed user directory.

When running M queries from within the Power Query SDK, tracing is enabled at the project level. On the project properties page, there are three settings related to tracing:

- **Clear Log**—when this is set to `true`, the log will be reset/cleared when you run your queries. We recommend you keep this set to `true`.

- **Show Engine Traces**—this setting controls the output of built-in traces from the M engine. These traces are generally only useful to members of the Power Query team, so you'll typically want to keep this set to `false`.
- **Show User Traces**—this setting controls trace information output by your connector. You'll want to set this to `true`.



Once enabled, you'll start seeing log entries in the M Query Output window, under the Log tab.

Diagnostics.Trace

The [Diagnostics.Trace](#) function is used to write messages into the M engine's trace log.

```
Diagnostics.Trace = (traceLevel as number, message as text, value as any, optional delayed as nullable logical as any) => ...
```

IMPORTANT

M is a functional language with lazy evaluation. When using `Diagnostics.Trace`, keep in mind that the function will only be called if the expression it's a part of is actually evaluated. Examples of this can be found later in this tutorial.

The `traceLevel` parameter can be one of the following values (in descending order):

- `TraceLevel.Critical`
- `TraceLevel.Error`
- `TraceLevel.Warning`
- `TraceLevel.Information`
- `TraceLevel.Verbose`

When tracing is enabled, the user can select the maximum level of messages they would like to see. All trace messages of this level and under will be output to the log. For example, if the user selects the "Warning" level, trace messages of `TraceLevel.Warning`, `TraceLevel.Error`, and `TraceLevel.Critical` would appear in the logs.

The `message` parameter is the actual text that will be output to the trace file. Note that the text will not contain the `value` parameter unless you explicitly include it in the text.

The `value` parameter is what the function will return. When the `delayed` parameter is set to `true`, `value` will be

a zero parameter function that returns the actual value you're evaluating. When `delayed` is set to `false`, `value` will be the actual value. An example of how this works can be [found below](#).

Using `Diagnostics.Trace` in the TripPin connector

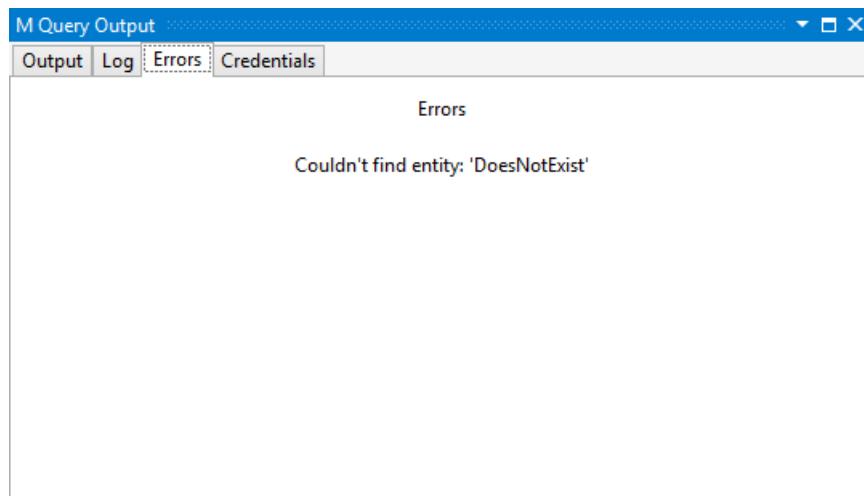
For a practical example of using `Diagnostics.Trace` and the impact of the `delayed` parameter, update the TripPin connector's `GetSchemaForEntity` function to wrap the `error` exception:

```
GetSchemaForEntity = (entity as text) as type =>
    try
        SchemaTable{[Entity=entity]}[Type]
    otherwise
        let
            message = Text.Format("Couldn't find entity: '#{0}'", {entity})
        in
            Diagnostics.Trace(TraceLevel.Error, message, () => error message, true);
```

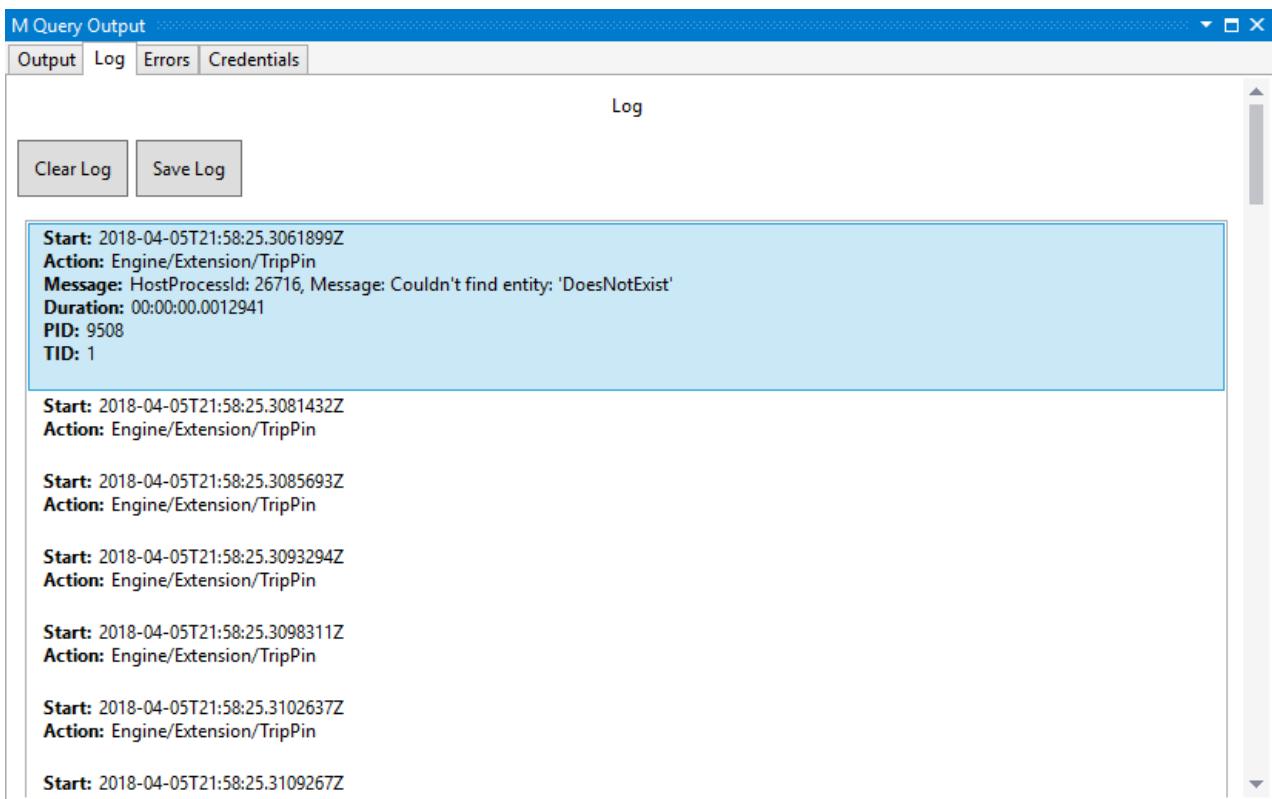
You can force an error during evaluation (for test purposes!) by passing an invalid entity name to the `GetEntity` function. Here you change the `withData` line in the `TripPinNavTable` function, replacing `[Name]` with `"DoesNotExist"`.

```
TripPinNavTable = (url as text) as table =>
    let
        // Use our schema table as the source of top level items in the navigation tree
        entities = Table.SelectColumns(SchemaTable, {"Entity"}),
        rename = Table.RenameColumns(entities, {[{"Entity", "Name"}]}),
        // Add Data as a calculated column
        withData = Table.AddColumn(rename, "Data", each GetEntity(url, "DoesNotExist"), type table),
        // Add ItemKind and ItemName as fixed text values
        withItemKind = Table.AddColumn(withData, "ItemKind", each "Table", type text),
        withItemName = Table.AddColumn(withItemKind, "ItemName", each "Table", type text),
        // Indicate that the node should not be expandable
        withIsLeaf = Table.AddColumn(withItemName, "IsLeaf", each true, type logical),
        // Generate the nav table
        navTable = Table.ToNavigationTable(withIsLeaf, {"Name"}, "Name", "Data", "ItemKind", "ItemName",
        "IsLeaf")
    in
        navTable;
```

Enable [tracing](#) for your project, and run your test queries. On the `Errors` tab you should see the text of the error you raised:



Also, on the `Log` tab, you should see the same message. Note that if you use different values for the `message` and `value` parameters, these would be different.



Also note that the `Action` field of the log message contains the name (Data Source Kind) of your extension (in this case, `Engine/Extension/TripPin`). This makes it easier to find the messages related to your extension when there are multiple queries involved and/or system (mashup engine) tracing is enabled.

Delayed evaluation

As an example of how the `delayed` parameter works, you'll make some modifications and run the queries again.

First, set the `delayed` value to `false`, but leave the `value` parameter as-is:

```
Diagnostics.Trace(TraceLevel.Error, message, () => error message, false);
```

When you run the query, you'll receive an error that "We cannot convert a value of type Function to type Type", and not the actual error you raised. This is because the call is now returning a `function` value, rather than the value itself.

Next, remove the function from the `value` parameter:

```
Diagnostics.Trace(TraceLevel.Error, message, error message, false);
```

When you run the query, you'll receive the correct error, but if you check the `Log` tab, there will be no messages. This is because the `error` ends up being raised/evaluated *during* the call to `Diagnostics.Trace`, so the message is never actually output.

Now that you understand the impact of the `delayed` parameter, be sure to reset your connector back to a working state before proceeding.

Diagnostic helper functions in `Diagnostics.pqm`

The `Diagnostics.pqm` file included in this project contains a number of helper functions that make tracing easier. As shown in the [previous tutorial](#), you can include this file in your project (remembering to set the Build Action to `Compile`), and then load it in your connector file. The bottom of your connector file should now look something

like the code snippet below. Feel free to explore the various functions this module provides, but in this sample, you'll only be using the `Diagnostics.LogValue` and `Diagnostics.LogFailure` functions.

```
// Diagnostics module contains multiple functions. We can take the ones we need.  
Diagnostics = Extension.LoadFunction("Diagnostics.pqm");  
Diagnostics.LogValue = Diagnostics[LogValue];  
Diagnostics.LogFailure = Diagnostics[LogFailure];
```

Diagnostics.LogValue

The `Diagnostics.LogValue` function is a lot like `Diagnostics.Trace`, and can be used to output the value of what you're evaluating.

```
Diagnostics.LogValue = (prefix as text, value as any) as any => ...
```

The `prefix` parameter is prepended to the log message. You'd use this to figure out which call output the message. The `value` parameter is what the function will return, and will also be written to the trace as a text representation of the M value. For example, if `value` is equal to a `table` with columns A and B, the log will contain the equivalent `#table` representation: `#table({{"A", "B"}, {"row1 A", "row1 B"}, {"row2 A", "row2 B"}})`

NOTE

Serializing M values to text can be an expensive operation. Be aware of the potential size of the values you are outputting to the trace.

NOTE

Most Power Query environments will truncate trace messages to a maximum length.

As an example, you'll update the `TripPin.Feed` function to trace the `url` and `schema` arguments passed into the function.

```
TripPin.Feed = (url as text, optional schema as type) as table =>  
    let  
        _url = Diagnostics.LogValue("Accessing url", url),  
        _schema = Diagnostics.LogValue("Schema type", schema),  
        //result = GetAllPagesByNextLink(url, schema)  
        result = GetAllPagesByNextLink(_url, _schema)  
    in  
        result;
```

Note that you have to use the new `_url` and `_schema` values in the call to `GetAllPagesByNextLink`. If you used the original function parameters, the `Diagnostics.LogValue` calls would never actually be evaluated, resulting in no messages written to the trace. *Functional programming is fun!*

When you run your queries, you should now see new messages in the log.

Accessing url:

M Query Output

Output Log Errors Credentials

Log

Clear Log Save Log

```
Start: 2018-04-05T21:01:25.1781124Z
Action: Engine/Extension/TripPin
Message: HostProcessId: 26716, Message: Accessing url: "http://services.odata.org/v4/TripPinService/Airlines"
Duration: 00:00:00.0000240
PID: 7796
TID: 1

Start: 2018-04-05T21:01:25.4088308Z
Action: Engine/Extension/TripPin

Start: 2018-04-05T21:01:25.5058279Z
Action: Engine/Extension/TripPin
```

Schema type:

M Query Output

Output Log Errors Credentials

Log

Clear Log Save Log

```
Start: 2018-04-06T15:48:31.1976744Z
Action: Engine/Extension/TripPin

Start: 2018-04-06T15:48:31.2190450Z
Action: Engine/Extension/TripPin
Message: HostProcessId: 26716, Message: Schema type: type table [AirlineCode = text, Name = text]
Duration: 00:00:00.0000225
PID: 7796
TID: 1

Start: 2018-04-06T15:48:31.4969434Z
Action: Engine/Extension/TripPin

Start: 2018-04-06T15:48:31.5014955Z
Action: Engine/Extension/TripPin

Start: 2018-04-06T15:48:31.5920417Z
Action: Engine/Extension/TripPin

Start: 2018-04-06T15:48:31.6089830Z
Action: Engine/Extension/TripPin
```

Note that you see the serialized version of the `schema` parameter `type`, rather than what you'd get when you do a simple `Text.FromValue` on a type value (which results in "type").

Diagnostics.LogFailure

The `Diagnostics.LogFailure` function can be used to wrap function calls, and will only write to the trace if the function call fails (that is, returns an `error`).

```
Diagnostics.LogFailure = (text as text, function as function) as any => ...
```

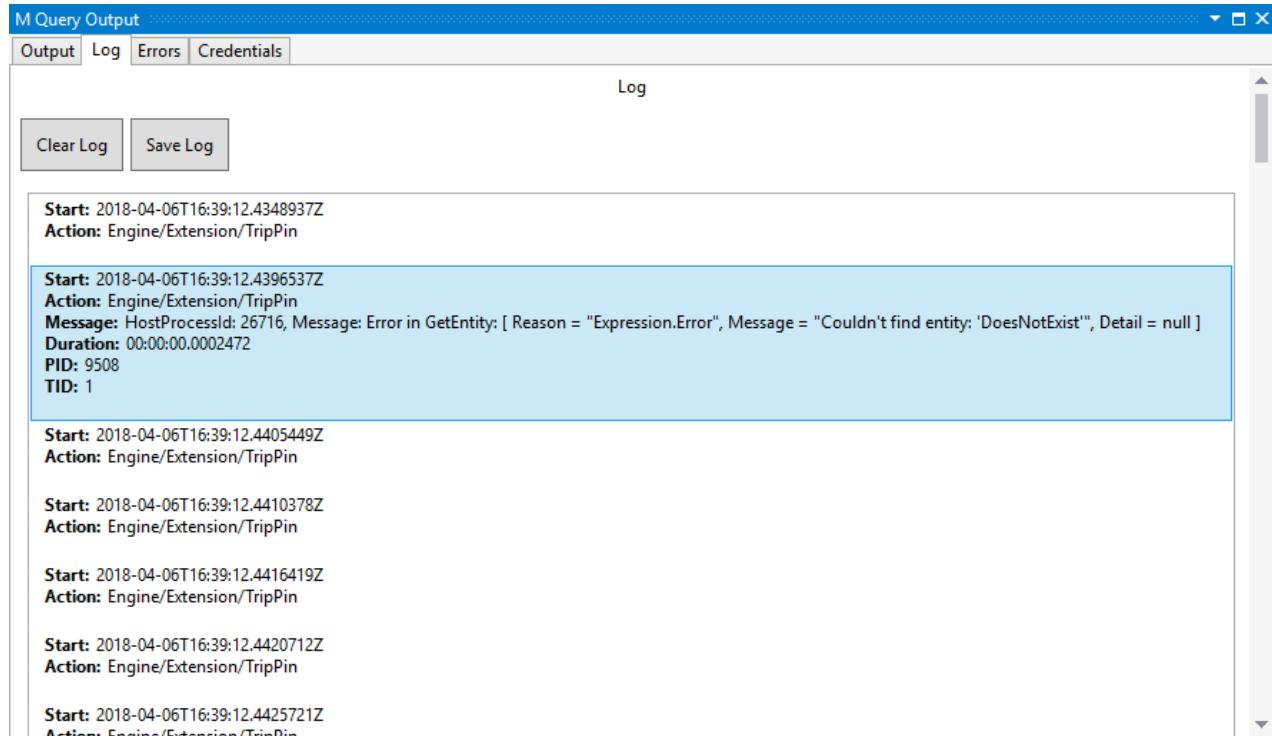
Internally, `Diagnostics.LogFailure` adds a `try` operator to the `function` call. If the call fails, the `text` value is

written to the trace before returning the original `error`. If the `function` call succeeds, the result is returned without writing anything to the trace. Since M errors don't contain a full stack trace (that is, you typically only see the message of the error), this can be useful when you want to pinpoint where the error was actually raised.

As a (poor) example, modify the `withData` line of the `TripPinNavTable` function to force an error once again:

```
withData = Table.AddColumn(rename, "Data", each Diagnostics.LogError("Error in GetEntity", () => GetEntity(url, "DoesNotExist")), type table),
```

In the trace, you can find the resulting error message containing your `text`, and the original error information.



Be sure to reset your function to a working state before proceeding with the next tutorial.

Conclusion

This brief (but important!) lesson showed you how to make use of the diagnostic helper functions to log to the Power Query trace files. When used properly, these functions are extremely useful in debugging issues within your connector.

NOTE

As a connector developer, it is your responsibility to ensure that you do not log sensitive or personally identifiable information (PII) as part of your diagnostic logging. You must also be careful to not output too much trace information, as it can have a negative performance impact.

TripPin Part 9 - TestConnection

4 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Add a TestConnection handler
- Configure the On-Premises Data Gateway (Personal mode)
- Test scheduled refresh through the Power BI service

Custom Connector support was added to the April 2018 release of the [Personal On-Premises Gateway](#). This new (preview) functionality allows for Scheduled Refresh of reports that make use of your custom connector.

This tutorial will cover the process of enabling your connector for refresh, and provide a quick walkthrough of the steps to configure the gateway. Specifically you will:

1. Add a TestConnection handler to your connector
2. Install the On-Premises Data Gateway in Personal mode
3. Enable Custom Connector support in the Gateway
4. Publish a workbook that uses your connector to PowerBI.com
5. Configure scheduled refresh to test your connector

See the [Handling Gateway Support](#) for more information on the TestConnection handler.

Background

There are three prerequisites for configuring a data source for scheduled refresh using PowerBI.com:

- **The data source is supported:** This means that the target gateway environment is aware of all of the functions contained within the query you want to refresh.
- **Credentials are provided:** To present the right credential entry dialog, Power BI needs to know the support authentication mechanism for a given data source.
- **The credentials are valid:** After the user provides credentials, they are validated by calling the data source's `TestConnection` handler.

The first two items are handled by registering your connector with the gateway. When the user attempts to configure scheduled refresh in PowerBI.com, the query information is sent to your personal gateway to determine if any data sources that aren't recognized by the Power BI service (that is, custom ones that you created) are available there. The third item is handled by invoking the TestConnection handler defined for your data source.

Adding a TestConnection Handler

The TestConnection handler is added to the Data Source Kind declaration record (the same place you declare its supported authentication type(s)). The handler is a `function` with a single parameter of type `any`, which returns a `list`. The first value in the list is the function that will be called to actually test the connection. This is generally the same as your main data source function. In some cases you may need to expose a separate `shared` function to provide an efficient connection test, however, this should generally be avoided.

Since the TripPin data source function has no required arguments, the implementation for TestConnection is fairly simple:

```
// Data Source Kind description
TripPin = [
    // TestConnection is required to enable the connector through the Gateway
    TestConnection = (dataSourcePath) => { "TripPin.Contents" },
    Authentication = [
        Anonymous = []
    ],
    Label = "TripPin Part 9 - TestConnection"
];
```

Future versions of the Power Query SDK will provide a way to validate the TestConnection handler from Visual Studio. Currently, the only mechanism that uses TestConnection is the On-premises Data Gateway.

Enabling Custom Connectors in the Personal Gateway

Reminder: Custom Connector support is currently in Preview, and requires the April 2018 or later version of the gateway, installed in Personal Mode.

Download and install the [On-Premises Data Gateway](#). When you run the installer, select the Personal Mode.

After installation is complete, launch the gateway and sign into Power BI. The sign-in process will automatically register your gateway with the Power BI services. Once signed in, perform the following steps:

1. Select the **Connectors** tab.
2. Select the switch to enable support for **Custom data connectors**.
3. Select the directory you want to load custom connectors from. This will usually be the same directory that you'd use for Power BI Desktop, but the value is configurable.
4. The page should now list all extension files in your target directory.

? X



On-premises data gateway (personal mode)

Status

Service Settings

Diagnostics

Network

Connectors

Custom data connectors



[Learn more](#)

Name

TripPin

Load custom data connectors from folder:

C:\Users\ [REDACTED] \Documents\Power BI Desktop\Custom Conn

...

Close

See the [online documentation](#) for more information about the gateway.

Testing Scheduled Refresh

Open Power BI Desktop and create a report that imports data using the TripPin connector.

Navigator

Display Options ▾

▲ TripPin Advanced Schema [3]

Airlines

Airports

People

UserName	FirstName	LastName	Emails	AddressInfo
russellwhyte	Russell	Whyte	List	List
scottketchum	Scott	Ketchum	List	List
ronaldmundy	Ronald	Mundy	List	List
javieralfred	Javier	Alfred	List	List
willieashmore	Willie	Ashmore	List	List
vincentcalabrese	Vincent	Calabrese	List	List
clydegueess	Clyde	Guess	List	List
keithpinckney	Keith	Pinckney	List	List
marshallgaray	Marshall	Garay	List	List
ryantheriault	Ryan	Theriault	List	List
elainestewart	Elaine	Stewart	List	List
salliesampson	Sallie	Sampson	List	List
jonirosales	Joni	Rosales	List	List
georginabarlow	Georgina	Barlow	List	List
angelhuffman	Angel	Huffman	List	List
laurelosborn	Laurel	Osborn	List	List
sandyosborn	Sandy	Osborn	List	List
ursulabright	Ursula	Bright	List	List
genevievereeves	Genevieve	Reeves	List	List
kristakemp	Krista	Kemp	List	List

◀ ▶

Load Edit Cancel

Add one or more visuals to your report page (optional), and then publish the report to PowerBI.com.

After publishing, go to PowerBI.com and find the dataset for the report you just published. Select the ellipses, and then select **Schedule Refresh**. Expand the **Gateway connection** and **Data source credentials** sections.

Settings for TripPin

[Refresh history](#)

▲ Gateway connection

To use a data gateway, make sure the computer is online and the data source is added in [Manage Gateways](#).

Use your data gateway (personal mode)

([Online](#), running on) [Delete Gateway](#)

Use an on-prem data gateway

[Apply](#)

[Discard](#)

▲ Data source credentials

Your data source can't be refreshed because the credentials are invalid. Please update your credentials and try again.

TripPin [Edit credentials](#)

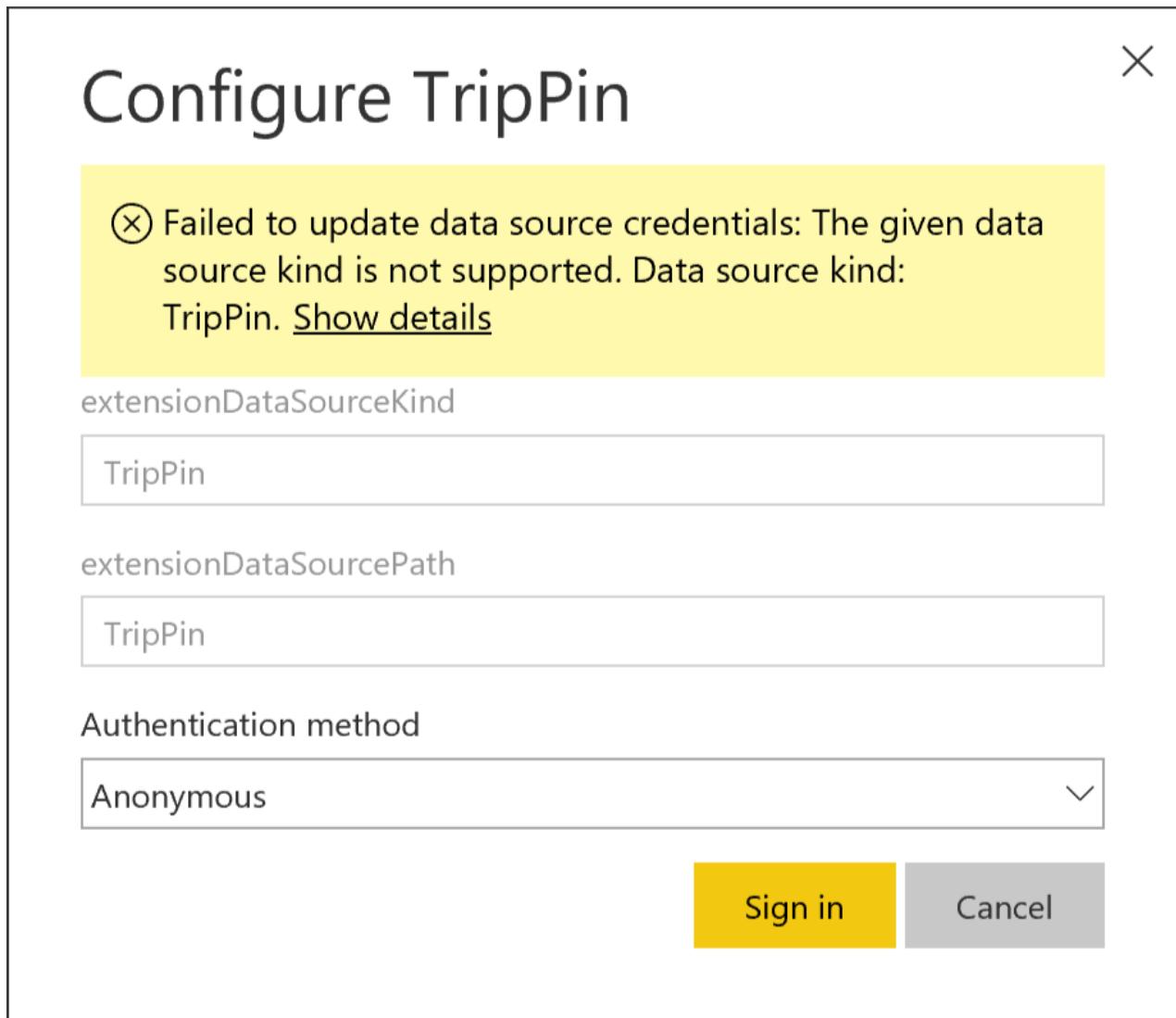
NOTE

If the dataset configuration page says that the report contains unknown data sources, your gateway/custom connector may not be configured properly. Go to the personal gateway configuration UI and make sure that there are no errors next to the TripPin connector. You may need to restart the gateway (on the **Service Settings** tab) to pick up the latest configuration.

Select the **Edit credentials** link to bring up the authentication dialog, and then select sign-in.

NOTE

If you receive an error similar to the one below ("Failed to update data source credentials"), you most likely have an issue with your TestConnection handler.



After a successful call to TestConnection, the credentials will be accepted. You can now schedule refresh, or select the dataset ellipse and then select **Refresh Now**. You can select the **Refresh history** link to view the status of the refresh (which generally takes a few minutes to get kicked off).

[Alerts](#)[Subscriptions](#)**Data source updated**

Your updates to the data source have been applied.

Settings for TripPin

[Refresh history](#)

► Gateway connection

To use a data gateway, make sure the computer is online and the data source is added in [Manage Gateways](#).

Use your data gateway (personal mode)

([Online](#), running on [REDACTED]) [Delete Gateway](#)

Use an on-prem data gateway

[Apply](#)[Discard](#)

► Data source credentials

TripPin

[Edit credentials](#)[► Parameters](#)[► Scheduled refresh](#)[► Q&A and Cortana](#)[► Featured Q&A questions](#)

Conclusion

Congratulations! You now have a production ready custom connector that supports automated refresh through the Power BI service.

TripPin Part 10 - Query Folding (part 1)

16 minutes to read • [Edit Online](#)

This multi-part tutorial covers the creation of a new data source extension for Power Query. The tutorial is meant to be done sequentially—each lesson builds on the connector created in previous lessons, incrementally adding new capabilities to your connector.

In this lesson, you will:

- Learn the basics of query folding
- Learn about the `Table.View` function
- Replicate OData query folding handlers for:
 - `$top`
 - `$skip`
 - `$count`
 - `$select`
 - `$orderby`

One of the powerful features of the M language is its ability to push transformation work to underlying data source(s). This capability is referred to as *Query Folding* (other tools/technologies also refer to similar function as Predicate Pushdown, or Query Delegation). When creating a custom connector that uses an M function with built-in query folding capabilities, such as `OData.Feed` or `OdBC.DataSource`, your connector will automatically inherit this capability for free.

This tutorial will replicate the built-in query folding behavior for OData by implementing function handlers for the `Table.View` function. This part of the tutorial will implement some of the *easier* handlers to implement (that is, ones that don't require expression parsing and state tracking). Future tutorials will implement more advanced query folding functionality.

To understand more about the query capabilities that an OData service might offer, see [OData v4 URL Conventions](#).

NOTE

As stated above, the `OData.Feed` function will automatically provide query folding capabilities. Since the TripPin series is treating the OData service as a regular REST API, using `Web.Contents` rather than `OData.Feed`, you'll need to implement the query folding handlers yourself. For real world usage, we recommend that you use `OData.Feed` whenever possible.

See the [Table.View documentation](#) for more information about query folding in M.

Using Table.View

The `Table.View` function allows a custom connector to override default transformation handlers for your data source. An implementation of `Table.View` will provide a function for one or more of the supported handlers. If a handler is unimplemented, or returns an `error` during evaluation, the M engine will fall back to its default handler.

When a custom connector uses a function that doesn't support implicit query folding, such as `Web.Contents`, default transformation handlers will always be performed locally. If the REST API you are connecting to supports query parameters as part of the query, `Table.View` will allow you to add optimizations that allow transformation

work to be pushed to the service.

The `Table.View` function has the following signature:

```
Table.View(table as nullable table, handlers as record) as table
```

Your implementation will wrap your main data source function. There are two required handlers for `Table.View`:

- `GetType` —returns the expected `table type` of the query result
- `GetRows` —returns the actual `table` result of your data source function

The simplest implementation would be similar to the following:

```
TripPin.SuperSimpleView = (url as text, entity as text) as table =>
    Table.View(null, [
        GetType = () => Value.Type(GetRows()),
        GetRows = () => GetEntity(url, entity)
    ]);
```

Update the `TripPinNavTable` function to call `TripPin.SuperSimpleView` rather than `GetEntity`:

```
withData = Table.AddColumn(rename, "Data", each TripPin.SuperSimpleView(url, [Name]), type table),
```

If you re-run the unit tests, you'll see that the behavior of your function hasn't changed. In this case your `Table.View` implementation is simply passing through the call to `GetEntity`. Since you haven't implemented any transformation handlers (yet), the original `url` parameter remains untouched.

Initial Implementation of `Table.View`

The above implementation of `Table.View` is simple, but not very useful. The following implementation will be used as your baseline—it doesn't implement any folding functionality, but has the scaffolding you'll need to do it.

```

TripPin.View = (baseUrl as text, entity as text) as table =>
    let
        // Implementation of Table.View handlers.
        //
        // We wrap the record with Diagnostics.WrapHandlers() to get some automatic
        // tracing if a handler returns an error.
        //
        View = (state as record) => Table.View(null, Diagnostics.WrapHandlers([
            // Returns the table type returned by GetRows()
            GetType = () => CalculateSchema(state),

            // Called last - retrieves the data from the calculated URL
            GetRows = () =>
                let
                    finalSchema = CalculateSchema(state),
                    finalUrl = CalculateUrl(state),

                    result = TripPin.Feed(finalUrl, finalSchema),
                    appliedType = Table.ChangeType(result, finalSchema)
                in
                    appliedType,

                //
                // Helper functions
                //
                // Retrieves the cached schema. If this is the first call
                // to CalculateSchema, the table type is calculated based on
                // the entity name that was passed into the function.
                CalculateSchema = (state) as type =>
                    if (state[Schema]? = null) then
                        GetSchemaForEntity(entity)
                    else
                        state[Schema],

                    // Calculates the final URL based on the current state.
                    CalculateUrl = (state) as text =>
                        let
                            urlWithEntity = Uri.Combine(state[Url], state[Entity])
                        in
                            urlWithEntity
                    ))
    in
        View([Url = baseUrl, Entity = entity]);

```

If you look at the call to `Table.View`, you'll see an additional wrapper function around the `handlers` record—`Diagnostics.WrapHandlers`. This helper function is found in the `Diagnostics` module (that was introduced in a previous tutorial), and provides you with a useful way to automatically trace any errors raised by individual handlers.

The `GetType` and `GetRows` functions have been updated to make use of two new helper functions—`CalculateSchema` and `CaculateUrl`. Right now the implementations of those functions are fairly straightforward—you'll notice they contain parts of what was previously done by the `GetEntity` function.

Finally, you'll notice that you're defining an internal function (`View`) that accepts a `state` parameter. As you implement more handlers, they will recursively call the internal `View` function, updating and passing along `state` as they go.

Update the `TripPinNavTable` function once again, replacing the call to `TripPin.SuperSimpleView` with a call to the new `TripPin.View` function, and re-run the unit tests. You won't see any new functionality yet, but you now have a solid baseline for testing.

Implementing Query Folding

Since the M engine will automatically fall back to local processing when a query can't be folded, you must take some additional steps to validate that your `Table.View` handlers are working correctly.

The manual way to validate folding behavior is to watch the URL requests your unit tests make using a tool like Fiddler. Alternatively, the diagnostic logging you added to `TripPin.Feed` will emit the full URL being run, which *should* include the OData query string parameters your handlers will add.

An automated way to validate query folding is to force your unit test execution to fail if a query doesn't fully fold. You can do this by opening the project properties, and setting **Error on Folding Failure** to **True**. With this setting enabled, any query that requires local processing results in the following error:

We couldn't fold the expression to the source. Please try a simpler expression.

You can test this out by adding a new `Fact` to your unit test file that contains one or more table transformations.

```
// Query folding tests
Fact("Fold $top 1 on Airlines",
    #table( type table [AirlineCode = text, Name = text] , {"AA", "American Airlines"} ),
    Table.FirstN(Airlines, 1)
)
```

NOTE

The **Error on Folding Failure** setting is an "all or nothing" approach. If you want to test queries that aren't designed to fold as part of your unit tests, you'll need to add some conditional logic to enable/disable tests accordingly.

The remaining sections of this tutorial will each add a new `Table.View` handler. You'll be taking a [Test Driven Development \(TDD\)](#) approach, where you first add failing unit tests, and then implement the M code to resolve them.

Each handler section below will describe the functionality provided by the handler, the OData equivalent query syntax, the unit tests, and the implementation. Using the scaffolding code described above, each handler implementation requires two changes:

- Adding the handler to `Table.View` that will update the `state` record.
- Modifying `calculateUrl` to retrieve the values from the `state` and add to the url and/or query string parameters.

Handling `Table.FirstN` with `OnTake`

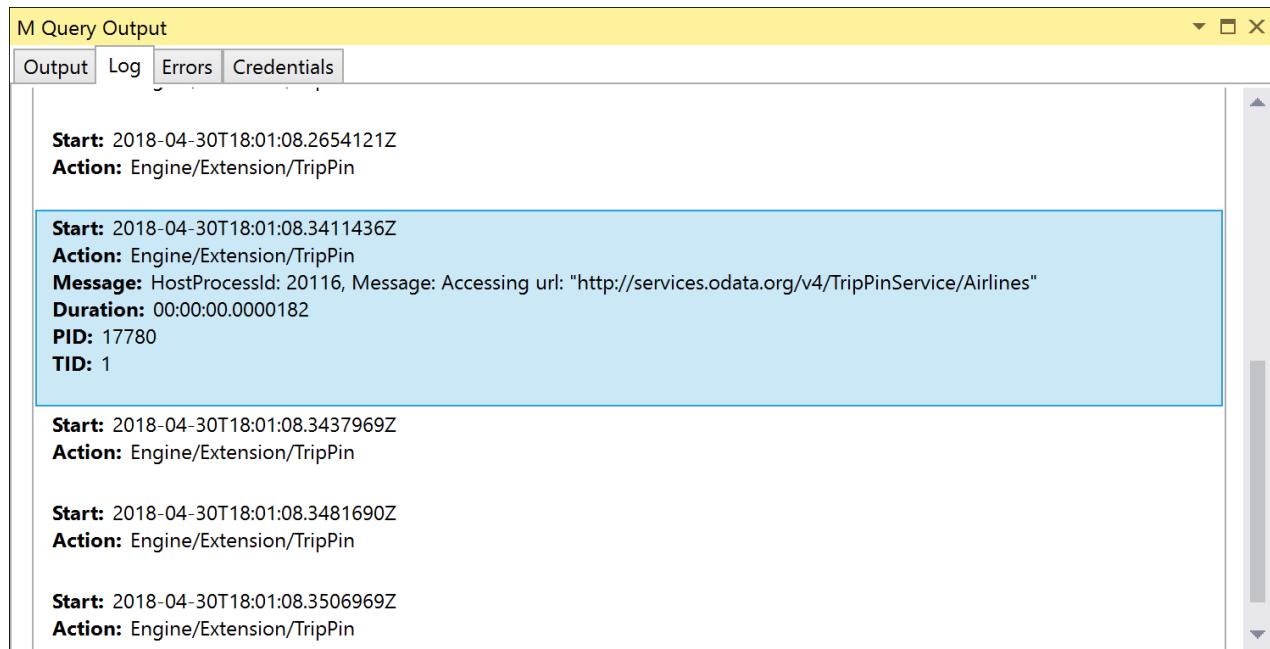
The [OnTake handler](#) receives a `count` parameter, which is the maximum number of rows to take. In OData terms, you can translate this to the `$top` query parameter.

You'll use the following unit tests:

```
// Query folding tests
Fact("Fold $top 1 on Airlines",
    #table( type table [AirlineCode = text, Name = text] , {"AA", "American Airlines"} ),
    Table.FirstN(Airlines, 1)
),
Fact("Fold $top 0 on Airports",
    #table( type table [Name = text, IataCode = text, Location = record] , {} ),
    Table.FirstN(Airports, 0)
),
```

These tests both use `Table.FirstN` to filter to the result set to the first X number of rows. If you run these tests with **Error on Folding Failure** set to `False` (the default), the tests should succeed, but if you run Fiddler (or check

the trace logs), you'll see that the request you send doesn't contain any OData query parameters.



The screenshot shows the 'M Query Output' window with the 'Log' tab selected. It displays several log entries from an Engine/Extension/TripPin action:

- Start:** 2018-04-30T18:01:08.2654121Z
- Action:** Engine/Extension/TripPin
- Start:** 2018-04-30T18:01:08.3411436Z
- Action:** Engine/Extension/TripPin
- Message:** HostProcessId: 20116, Message: Accessing url: "http://services.odata.org/v4/TripPinService/Airlines"
- Duration:** 00:00:00.0000182
- PID:** 17780
- TID:** 1
- Start:** 2018-04-30T18:01:08.3437969Z
- Action:** Engine/Extension/TripPin
- Start:** 2018-04-30T18:01:08.3481690Z
- Action:** Engine/Extension/TripPin
- Start:** 2018-04-30T18:01:08.3506969Z
- Action:** Engine/Extension/TripPin

If you set **Error on Folding Failure** to `True`, they will fail with the "Please try a simpler expression." error. To fix this, you'll define your first Table.View handler for `OnTake`.

The OnTake handler looks like this:

```
OnTake = (count as number) =>
  let
    // Add a record with Top defined to our state
    newState = state & [ Top = count ]
  in
    @View(newState),
```

The `CalculateUrl` function is updated to extract the `Top` value from the `state` record, and set the right parameter in the query string.

```
// Calculates the final URL based on the current state.
CalculateUrl = (state) as text =>
  let
    urlWithEntity = Uri.Combine(state[Url], state[Entity]),

    // Uri.BuildQueryString requires that all field values
    // are text literals.
    defaultQueryString = [],

    // Check for Top defined in our state
    qsWithTop =
      if (state[Top]? <> null) then
        // add a $top field to the query string record
        defaultQueryString & [ #"$top" = Number.ToText(state[Top]) ]
      else
        defaultQueryString,

    encodedQueryString = Uri.BuildQueryString(qsWithTop),
    finalUrl = urlWithEntity & "?" & encodedQueryString
  in
    finalUrl
```

Rerunning the unit tests, you can see that the URL you are accessing now contains the `$top` parameter. (Note that due to URL encoding, `$top` appears as `%24top`, but the OData service is smart enough to convert it)

automatically).

The screenshot shows the 'M Query Output' window with a yellow header bar. Below the header are tabs: 'Output' (which is selected), 'Log', 'Errors', and 'Credentials'. The main area contains several log entries:

- Action:** engine/extension/trippin
- Start:** 2018-04-30T19:22:43.1213089Z
Action: Engine/Extension/TripPin
- Start:** 2018-04-30T19:22:43.1297467Z
Action: Engine/Extension/TripPin
- Start:** 2018-04-30T19:22:43.1679684Z
Action: Engine/Extension/TripPin
Message: HostProcessId: 20116, Message: Accessing url: "http://services.odata.org/v4/TripPinService/People?%24top=2"
Duration: 00:00:00.0000459
PID: 24328
TID: 1
- Start:** 2018-04-30T19:22:43.1782248Z
Action: Engine/Extension/TripPin
- Start:** 2018-04-30T19:22:43.2195863Z
Action: Engine/Extension/TripPin

Handling Table.Skip with OnSkip

The [OnSkip handler](#) is a lot like OnTake. It receives a `count` parameter, which is the number of rows to skip from the result set. This translates nicely to the OData `$skip` query parameter.

Unit tests:

```
// OnSkip
Fact("Fold $skip 14 on Airlines",
    #table( type table [AirlineCode = text, Name = text] , [{"EK", "Emirates"}] ),
    Table.Skip(Airlines, 14)
),
Fact("Fold $skip 0 and $top 1",
    #table( type table [AirlineCode = text, Name = text] , [{"AA", "American Airlines"}] ),
    Table.FirstN(Table.Skip(Airlines, 0), 1)
),
```

Implementation:

```
// OnSkip - handles the Table.Skip transform.
// The count value should be >= 0.
OnSkip = (count as number) =>
    let
        newState = state & [ Skip = count ]
    in
        @View(newState),
```

Matching updates to `CalculateUrl`:

```
qsWithSkip =
    if (state[Skip]? <> null) then
        qsWithTop & [ "#$skip" = Number.ToString(state[Skip]) ]
    else
        qsWithTop,
```

Handling Table.SelectColumns with OnSelectColumns

The [OnSelectColumns](#) handler is called when the user selects or removes columns from the result set. The handler

receives a `list` of `text` values, representing the column(s) to be selected. In OData terms, this operation will map to the `$select` query option. The advantage of folding column selection becomes apparent when you are dealing with tables with many columns. The `$select` operator will remove unselected columns from the result set, resulting in more efficient queries.

Unit tests:

```
// OnSelectColumns
Fact("Fold $select single column",
    #table( type table [AirlineCode = text] , {"AA"} ),
    Table.FirstN(Table.SelectColumns(Airlines, {"AirlineCode"}), 1)
),
Fact("Fold $select multiple column",
    #table( type table [UserName = text, FirstName = text, LastName = text], {"russellwhyte", "Russell", "Whyte"} ),
    Table.FirstN(Table.SelectColumns(People, {"UserName", "FirstName", "LastName"}), 1)
),
Fact("Fold $select with ignore column",
    #table( type table [AirlineCode = text] , {"AA"} ),
    Table.FirstN(Table.SelectColumns(Airlines, {"AirlineCode", "DoesNotExist"}, MissingField.Ignore), 1)
),
```

The first two tests select different numbers of columns with `Table.SelectColumns`, and include a `Table.FirstN` call to simplify the test case.

NOTE

If the test were to simply return the column names (using `Table.ColumnNames`) and not any data, the request to the OData service will never actually be sent. This is because the call to `GetType` will return the schema, which contains all of the information the M engine needs to calculate the result.

The third test uses the `MissingField.Ignore` option, which tells the M engine to ignore any selected columns that don't exist in the result set. The `OnSelectColumns` handler does not need to worry about this option—the M engine will handle it automatically (that is, missing columns won't be included in the `columns` list).

NOTE

The other option for `Table.SelectColumns`, `MissingField.UseNull`, requires a connector to implement the `OnAddColumn` handler. This will be done in a subsequent lesson.

The implementation for `OnSelectColumns` does two things:

- Adds the list of selected columns to the `state`.
- Re-calculates the `Schema` value so you can set the right table type.

```

OnSelectColumns = (columns as list) =>
let
    // get the current schema
    currentSchema = CalculateSchema(state),
    // get the columns from the current schema (which is an M Type value)
    rowRecordType = Type.RecordFields(Type.TableRow(currentSchema)),
    existingColumns = Record.FieldNames(rowRecordType),
    // calculate the new schema
    columnsToRemove = List.Difference(existingColumns, columns),
    updatedColumns = Record.RemoveFields(rowRecordType, columnsToRemove),
    newSchema = type table (Type.ForRecord(updatedColumns, false))
in
@View(state &
[
    SelectColumns = columns,
    Schema = newSchema
]),

```

`CalculateUrl` is updated to retrieve the list of columns from the state, and combine them (with a separator) for the `$select` parameter.

```

// Check for explicitly selected columns
qsWithSelect =
if (state[SelectColumns]? <> null) then
    qsWithSkip & [ "#$select" = Text.Combine(state[SelectColumns], ",") ]
else
    qsWithSkip,

```

Handling Table.Sort with OnSort

The `OnSort` handler receives a `list` of `record` values. Each record contains a `Name` field, indicating the name of the column, and an `Order` field which is equal to `Order.Ascending` or `Order.Descending`. In OData terms, this operation will map to the `$orderby` query option. The `$orderby` syntax has the column name followed by `asc` or `desc` to indicate ascending or descending order. When sorting on multiple columns, the values are separated with a comma. Note that if the `columns` parameter contains more than one item, it is important to maintain the order in which they appear.

Unit tests:

```

// OnSort
Fact("Fold $orderby single column",
    #table( type table [AirlineCode = text, Name = text], [{"TK", "Turkish Airlines"}]),
    Table.FirstN(Table.Sort(Airlines, {"AirlineCode", Order.Descending}), 1)
),
Fact("Fold $orderby multiple column",
    #table( type table [UserName = text], [{"javieralfred"}]),
    Table.SelectColumns(Table.FirstN(Table.Sort(People, {"LastName", Order.Ascending}, {"UserName", Order.Descending})), 1), {"UserName"})
)

```

Implementation:

```

// OnSort - receives a list of records containing two fields:
//   [Name] - the name of the column to sort on
//   [Order] - equal to Order.Ascending or Order.Descending
// If there are multiple records, the sort order must be maintained.
//
// OData allows you to sort on columns that do not appear in the result
// set, so we do not have to validate that the sorted columns are in our
// existing schema.
OnSort = (order as list) =>
    let
        // This will convert the list of records to a list of text,
        // where each entry is "<columnName> <asc|desc>"
        sorting = List.Transform(order, (o) =>
            let
                column = o[Name],
                order = o[Order],
                orderText = if (order = Order.Ascending) then "asc" else "desc"
            in
                column & " " & orderText
        ),
        orderBy = Text.Combine(sorting, ", ")
    in
        @View(state & [ OrderBy = orderBy ]),

```

Updates to `CalculateUrl`:

```

qsWithOrderBy =
    if (state[OrderBy]? <> null) then
        qsWithSelect & [ #"$orderby" = state[OrderBy] ]
    else
        qsWithSelect,

```

Handling `Table.RowCount` with `GetRowCount`

Unlike the other query handlers you've implemented, the `GetRowCount` handler will return a single value—the number of rows expected in the result set. In an M query, this would typically be the result of the `Table.RowCount` transform. You have a few different options on how to handle this as part of an OData query.

- The [\\$count query parameter](#), which returns the count as a separate field in the result set.
- The [/\\$count path segment](#), which will return **only** the total count, as a scalar value.

The downside to the query parameter approach is that you still need to send the entire query to the OData service. Since the count comes back inline as part of the result set, you'll have to process the first page of data from the result set. While this is still more efficient than reading the entire result set and counting the rows, it's probably still more work than you want to do.

The advantage of the path segment approach is that you'll only receive a single scalar value in the result. This makes the entire operation a lot more efficient. However, as described in the OData specification, the `/$count` path segment will return an error if you include other query parameters, such as `$top` or `$skip`, which limits its usefulness.

In this tutorial, you'll implement the `GetRowCount` handler using the path segment approach. To avoid the errors you'd get if other query parameters are included, you'll check for other state values, and return an "unimplemented error" (`...`) if you find any. Returning any error from a `Table.View` handler tells the M engine that the operation cannot be folded, and it should fallback to the default handler instead (which in this case would be counting the total number of rows).

First, add a simple unit test:

```
// GetRowCount
Fact("Fold $count", 15, Table.RowCount(Airlines)),
```

Since the `/$count` path segment returns a single value (in plain/text format) rather than a JSON result set, you'll also have to add a new internal function (`TripPin.Scalar`) for making the request and handling the result.

```
// Similar to TripPin.Feed, but is expecting back a scalar value.
// This function returns the value from the service as plain text.
TripPin.Scalar = (url as text) as text =>
    let
        _url = Diagnostics.LogValue("TripPin.Scalar url", url),
        headers = DefaultRequestHeaders & [
            #"Accept" = "text/plain"
        ],
        response = Web.Contents(_url, [ Headers = headers ]),
        toText = Text.FromBinary(response)
    in
        toText;
```

The implementation will then use this function (if no other query parameters are found in the `state`):

```
GetRowCount = () as number =>
    if (Record.FieldCount(Record.RemoveFields(state, {"Url", "Entity", "Schema"}, MissingField.Ignore)) > 0)
    then
        ...
    else
        let
            newState = state & [ RowCountOnly = true ],
            finalUrl = CalculateUrl(newState),
            value = TripPin.Scalar(finalUrl),
            converted = Number.FromText(value)
        in
            converted,
```

The `CalculateUrl` function is updated to append `/$count` to the URL if the `RowCountOnly` field is set in the `state`.

```
// Check for $count. If all we want is a row count,
// then we add /$count to the path value (following the entity name).
urlWithRowCount =
    if (state[RowCountOnly]? = true) then
        urlWithEntity & "/$count"
    else
        urlWithEntity,
```

The new `Table.RowCount` unit test should now pass.

To test the fallback case, you'll add another test that forces the error. First, add a helper method that checks the result of a `try` operation for a folding error.

```
// Returns true if there is a folding error, or the original record (for logging purposes) if not.
Test.IsFoldingError = (tryResult as record) =>
    if ( tryResult[HasError]? = true and tryResult[Error][Message] = "We couldn't fold the expression to the
data source. Please try a simpler expression." ) then
        true
    else
        tryResult;
```

Then add a test that uses both `Table.RowCount` and `Table.FirstN` to force the error.

```
// test will fail if "Fail on Folding Error" is set to false
Fact("Fold $count + $top *error*", true, Test.IsFoldingError(try Table.RowCount(Table.FirstN(Airlines, 3)))),
```

An important note here is that this test will now return an error if **Error on Folding Error** is set to `false`, because the `Table.RowCount` operation will fall back to the local (default) handler. Running the tests with **Error on Folding Error** set to `true` will cause `Table.RowCount` to fail, and allows the test to succeed.

Conclusion

Implementing `Table.View` for your connector adds a significant amount of complexity to your code. Since the M engine can process all transformations locally, adding `Table.View` handlers does not enable new scenarios for your users, but will result in more efficient processing (and potentially, happier users). One of the main advantages of the `Table.View` handlers being optional is that it allows you to incrementally add new functionality without impacting backwards compatibility for your connector.

For most connectors, an important (and basic) handler to implement is `OnTake` (which translates to `$top` in OData), as it limits the amount of rows returned. The Power Query experience will always perform an `OnTake` of `1000` rows when displaying previews in the navigator and query editor, so your users might see significant performance improvements when working with larger data sets.

In subsequent tutorials, we'll look at the more advanced query handlers (such as `OnSelectRows`), which require translating M expressions.

Github Connector Sample

7 minutes to read • [Edit Online](#)

The Github M extension shows how to add support for an OAuth 2.0 protocol authentication flow. You can learn more about the specifics of Github's authentication flow on the [Github Developer site](#).

Before you get started creating an M extension, you need to register a new app on Github, and replace the `client_id` and `client_secret` files with the appropriate values for your app.

Note about compatibility issues in Visual Studio: *The Power Query SDK uses an Internet Explorer based control to popup OAuth dialogs. Github has deprecated its support for the version of IE used by this control, which will prevent you from completing the permission grant for your app if run from within Visual Studio. An alternative is to load the extension with Power BI Desktop and complete the first OAuth flow there. After your application has been granted access to your account, subsequent logins will work fine from Visual Studio.*

OAuth and Power BI

OAuth is a form of credentials delegation. By logging in to Github and authorizing the "application" you create for Github, the user is allowing your "application" to login on their behalf to retrieve data into Power BI. The "application" must be granted rights to retrieve data (get an `access_token`) and to refresh the data on a schedule (get and use a `refresh_token`). Your "application" in this context is your Data Connector used to run queries within Power BI. Power BI stores and manages the `access_token` and `refresh_token` on your behalf.

NOTE

To allow Power BI to obtain and use the `access_token`, you must specify the redirect url as <https://oauth.powerbi.com/views/oauthredirect.html>.

When you specify this URL and Github successfully authenticates and grants permissions, Github will redirect to PowerBI's `oauthredirect` endpoint so that Power BI can retrieve the `access_token` and `refresh_token`.

How to register a Github app

Your Power BI extension needs to login to Github. To enable this, you register a new OAuth application with Github at <https://Github.com/settings/applications/new>.

1. `Application name` : Enter a name for the application for your M extension.
2. `Authorization callback URL` : Enter <https://oauth.powerbi.com/views/oauthredirect.html>.
3. `Scope` : In Github, set scope to `user, repo`.

NOTE

A registered OAuth application is assigned a unique Client ID and Client Secret. The Client Secret should not be shared. You get the Client ID and Client Secret from the Github application page. Update the files in your Data Connector project with the Client ID (`client_id` file) and Client Secret (`client_secret` file).

How to implement Github OAuth

This sample will walk you through the following steps:

1. Create a Data Source Kind definition that declares it supports OAuth.
2. Provide details so the M engine can start the OAuth flow (`StartLogin`).
3. Convert the code received from Github into an access_token (`FinishLogin` and `TokenMethod`).
4. Define functions that access the Github API (`GithubSample.Contents`).

Step 1 - Create a Data Source definition

A Data Connector starts with a `record` that describes the extension, including its unique name (which is the name of the record), supported authentication type(s), and a friendly display name (label) for the data source. When supporting OAuth, the definition contains the functions that implement the OAuth contract—in this case, `StartLogin` and `FinishLogin`.

```
//  
// Data Source definition  
//  
GithubSample = [  
    Authentication = [  
        OAuth = [  
            StartLogin = StartLogin,  
            FinishLogin = FinishLogin  
        ]  
    ],  
    Label = Extension.LoadString("DataSourceLabel")  
];
```

Step 2 - Provide details so the M engine can start the OAuth flow

The Github OAuth flow starts when you direct users to the <https://Github.com/login/oauth/authorize> page. For the user to login, you need to specify a number of query parameters:

NAME	TYPE	DESCRIPTION
client_id	string	Required. The client ID you received from Github when you registered.
redirect_uri	string	The URL in your app where users will be sent after authorization. See details below about redirect urls. For M extensions, the <code>redirect_uri</code> must be "https://oauth.powerbi.com/views/oauthredirect.html".
scope	string	A comma separated list of scopes. If not provided, scope defaults to an empty list of scopes for users that don't have a valid token for the app. For users who do already have a valid token for the app, the user won't be shown the OAuth authorization page with the list of scopes. Instead, this step of the flow will automatically complete with the same scopes that were used last time the user completed the flow.
state	string	An un-guessable random string. It's used to protect against cross-site request forgery attacks.

The following code snippet describes how to implement a `StartLogin` function to start the login flow. A

`StartLogin` function takes a `resourceUrl`, `state`, and `display` value. In the function, create an `AuthorizeUrl` that concatenates the Github authorize URL with the following parameters:

- `client_id`: You get the client ID after you register your extension with Github from the Github application page.
- `scope`: Set scope to "`user, repo`". This sets the authorization scope (that is, what your app wants to access) for the user.
- `state`: An internal value that the M engine passes in.
- `redirect_uri`: Set to <https://oauth.powerbi.com/views/oauthredirect.html>.

```
StartLogin = (resourceUrl, state, display) =>
    let
        AuthorizeUrl = "https://Github.com/login/oauth/authorize?" & Uri.BuildQueryString([
            client_id = client_id,
            scope = "user, repo",
            state = state,
            redirect_uri = redirect_uri])
    in
    [
        LoginUri = AuthorizeUrl,
        CallbackUri = redirect_uri,
        WindowHeight = windowHeight,
        WindowWidth = windowWidth,
        Context = null
    ];

```

If this is the first time the user is logging in with your app (identified by its `client_id` value), they'll see a page that asks them to grant access to your app. Subsequent login attempts will simply ask for their credentials.

Step 3 - Convert the code received from Github into an access_token

If the user completes the authentication flow, Github redirects back to the Power BI redirect URL with a temporary code in a `code` parameter, as well as the state you provided in the previous step in a `state` parameter. Your `FinishLogin` function will extract the code from the `callbackUri` parameter, and then exchange it for an access token (using the `TokenMethod` function).

```
FinishLogin = (context, callbackUri, state) =>
    let
        Parts = Uri.Parts(callbackUri)[Query]
    in
        TokenMethod(Parts[code]);
```

To get a Github access token, you pass the temporary code from the Github Authorize Response. In the `TokenMethod` function, you formulate a POST request to Github's `access_token` endpoint (https://github.com/login/oauth/access_token). The following parameters are required for the Github endpoint:

NAME	TYPE	DESCRIPTION
client_id	string	Required . The client ID you received from Github when you registered.
client_secret	string	Required . The client secret you received from Github when you registered.
code	string	Required . The code you received in <code>FinishLogin</code> .

NAME	TYPE	DESCRIPTION
redirect_uri	string	The URL in your app where users will be sent after authorization. See details below about redirect URLs.

Here are the details used parameters for the [Web.Contents](#) call.

ARGUMENT	DESCRIPTION	VALUE
url	The URL for the Web site.	https://Github.com/login/oauth/access_token
options	A record to control the behavior of this function.	Not used in this case
Query	Programmatically add query parameters to the URL.	<pre>Content = Text.ToBinary(Uri.BuildQueryString([client_id = client_id, client_secret = client_secret, code = code, redirect_uri = redirect_uri]))</pre> <p>Where</p> <ul style="list-style-type: none"> • <code>client_id</code> : Client ID from Github application page. • <code>client_secret</code> : Client secret from Github application page. • <code>code</code> : Code in Github authorization response. • <code>redirect_uri</code> : The URL in your app where users will be sent after authorization.
Headers	A record with additional headers for the HTTP request.	<pre>Headers= [#Content-type" = "application/x-www-form-urlencoded", #Accept" = "application/json"]</pre>

This code snippet describes how to implement a `TokenMethod` function to exchange an auth code for an access token.

```
TokenMethod = (code) =>
    let
        Response = Web.Contents("https://Github.com/login/oauth/access_token", [
            Content = Text.ToBinary(Uri.BuildQueryString([
                client_id = client_id,
                client_secret = client_secret,
                code = code,
                redirect_uri = redirect_uri])),
            Headers=[#"Content-type" = "application/x-www-form-urlencoded",#"Accept" = "application/json"]),
            Parts = Json.Document(Response)
        in
            Parts;
```

The JSON response from the service will contain an `access_token` field. The `TokenMethod` method converts the JSON response into an M record using [Json.Document](#), and returns it to the engine.

Sample response:

```
{  
    "access_token": "e72e16c7e42f292c6912e7710c838347ae178b4a",  
    "scope": "user,repo",  
    "token_type": "bearer"  
}
```

Step 4 - Define functions that access the Github API

The following code snippet exports two functions (`GithubSample.Contents` and `GithubSample.PagedTable`) by marking them as `shared`, and associates them with the `GithubSample` Data Source Kind.

```
[DataSource.Kind="GithubSample", Publish="GithubSample.UI"]  
shared GithubSample.Contents = Value.ReplaceType(Github.Contents, type function (url as Uri.Type) as any);  
  
[DataSource.Kind="GithubSample"]  
shared GithubSample.PagedTable = Value.ReplaceType(Github.PagedTable, type function (url as Uri.Type) as nullable table);
```

The `GithubSample.Contents` function is also published to the UI (allowing it to appear in the **Get Data** dialog). The `Value.ReplaceType` function is used to set the function parameter to the `Uri.Type` ascribed type.

By associating these functions with the `GithubSample` data source kind, they'll automatically use the credentials that the user provided. Any M library functions that have been enabled for extensibility (such as `Web.Contents`) will automatically inherit these credentials as well.

For more details on how credential and authentication works, see [Handling Authentication](#).

Sample URL

This connector is able to retrieve formatted data from any of the Github v3 REST API endpoints. For example, the query to pull all commits to the Data Connectors repo would look like this:

```
GithubSample.Contents("https://api.github.com/repos/microsoft/dataconnectors/commits")
```

MyGraph Connector Sample

12 minutes to read • [Edit Online](#)

In this sample you'll create a basic data source connector for [Microsoft Graph](#). It is written as a walk-through that you can follow step by step.

To access Graph, you'll first need to register your own Azure Active Directory (AAD) client application. If you don't have an application ID already, you can create one through the [Getting Started with Microsoft Graph](#) site. Select the **Universal Windows** option, and then the **Let's go** button. Follow the steps and receive an App ID. As described in the steps below, use `https://oauth.powerbi.com/views/oauthredirect.html` as your redirect URI when registering your app. Use the Client ID value to replace the existing value in the `client_id` file in the code sample.

Writing an OAuth v2 Flow with Power BI Desktop

There are three parts to implementing your OAuth Flow:

- Create the URL for the Authorization endpoint
- Post a code to the Token endpoint and extract the auth and refresh tokens
- Trade a refresh token for a new auth token

You'll use Power Query (through Power BI Desktop) to write the M code for the Graph OAuth flow.

First, review details about how the OAuth v2 flow works for Graph:

- [Microsoft Graph App authentication using Azure AD](#)
- [Authentication Code Grant Flow](#)
- [Permission scopes](#)

You'll also want to download and install [Fiddler](#) to help trace the raw HTTP requests you make while developing the extension.

To get started, create a new blank query in Power BI Desktop and bring up the advanced query editor.

Define the following variables that will be used in your OAuth flow:

```
let
    client_id = "<your app id>",
    redirect_uri = "https://oauth.powerbi.com/views/oauthredirect.html",
    token_uri = "https://login.microsoftonline.com/common/oauth2/v2.0/token",
    authorize_uri = "https://login.microsoftonline.com/common/oauth2/v2.0/authorize",
    logout_uri = "https://login.microsoftonline.com/logout.srf"
in
    logout_uri
```

Set the `client_id` with the app id you received when you registered your Graph application.

Graph has an extensive list of permission scopes that your application can request. For this sample, the app will request all scopes that do not require admin consent. You'll define two more variables—a list of the scopes you want, and the prefix string that graph uses. You'll also add a couple of helper functions to convert the scope list into the expected format.

```

// The "offline_access" scope is required to receive a refresh token value. It is added
// separately from the Graph scopes. Please see
// https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-permissions-and-consent#offline_access
//
// For more information on available Graph scopes, please see
// https://developer.microsoft.com/en-us/graph/docs/authorization/permission_scopes
scope_prefix = "https://graph.microsoft.com/",
scopes = {
    "User.ReadWrite",
    "Contacts.Read",
    "User.ReadBasic.All",
    "Calendars.ReadWrite",
    "Mail.ReadWrite",
    "Mail.Send",
    "Contacts.ReadWrite",
    "Files.ReadWrite",
    "Tasks.ReadWrite",
    "People.Read",
    "Notes.ReadWrite.All",
    "Sites.Read.All"
},
Value.IfNull = (a, b) => if a <> null then a else b,
GetScopeString = (scopes as list, optional scopePrefix as text) as text =>
let
    prefix = Value.IfNull(scopePrefix, ""),
    addPrefix = List.Transform(scopes, each prefix & _),
    asText = Text.Combine(addPrefix, " ")
in
    asText,

```

The `GetScopeString` function will end up generating a scope string that looks like this:

```
https://graph.microsoft.com/User.ReadWrite https://graph.microsoft.com/Contacts.Read
https://graph.microsoft.com/User.ReadBasic.All ...
```

You'll need to set [several query string parameters](#) as part of the authorization URL. You can use the `Uri.BuildQueryString` function to properly encode the parameter names and values. Construct the URL by concatenating the `authorize_uri` variable and query string parameters.

The full code sample is below.

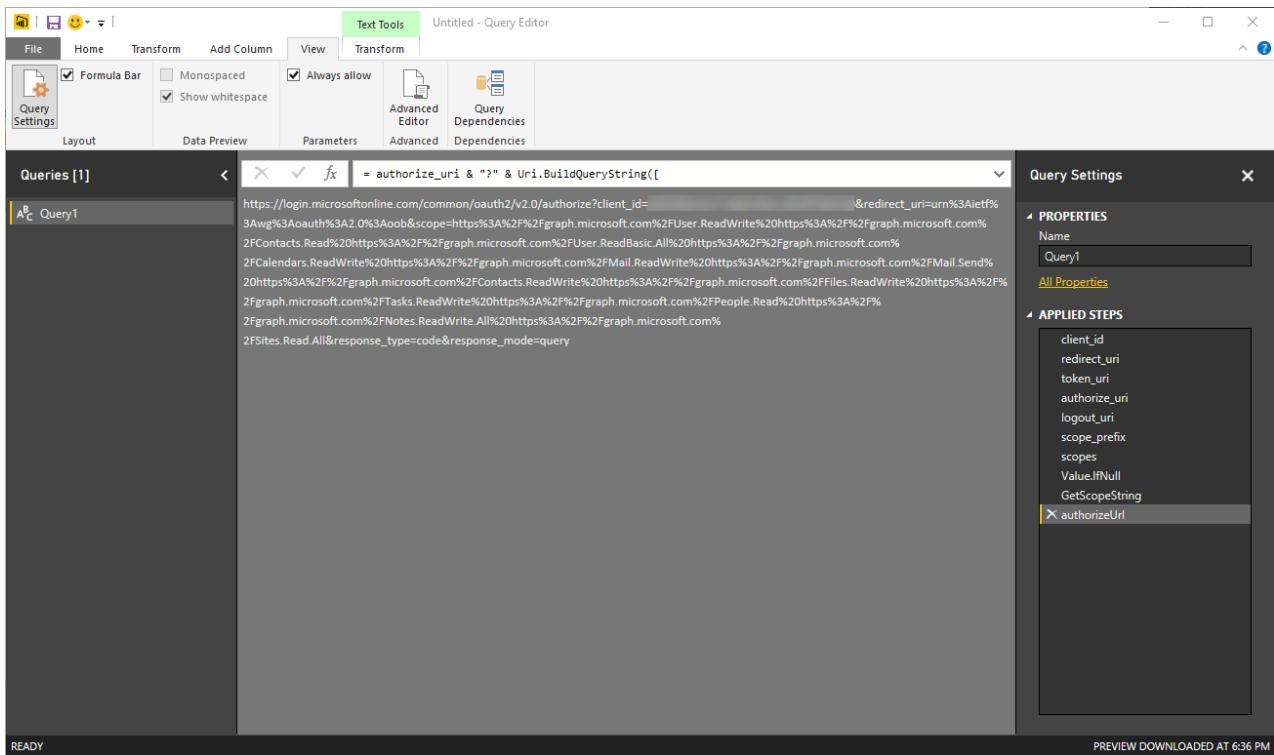
```

let
    client_id = "<your app id>",
    redirect_uri = "urn:ietf:wg:oauth:2.0:oob",
    token_uri = "https://login.microsoftonline.com/common/oauth2/v2.0/token",
    authorize_uri = "https://login.microsoftonline.com/common/oauth2/v2.0/authorize",
    logout_uri = "https://login.microsoftonline.com/logout.srf",

    scope_prefix = "https://graph.microsoft.com/",
    scopes = {
        "User.ReadWrite",
        "Contacts.Read",
        "User.ReadBasic.All",
        "Calendars.ReadWrite",
        "Mail.ReadWrite",
        "Mail.Send",
        "Contacts.ReadWrite",
        "Files.ReadWrite",
        "Tasks.ReadWrite",
        "People.Read",
        "Notes.ReadWrite.All",
        "Sites.Read.All"
    },
    Value.IfNull = (a, b) => if a <> null then a else b,
    GetScopeString = (scopes as list, optional scopePrefix as text) as text =>
        let
            prefix = Value.IfNull(scopePrefix, ""),
            addPrefix = List.Transform(scopes, each prefix & _),
            asText = Text.Combine(addPrefix, " ")
        in
            asText,
    authorizeUrl = authorize_uri & "?" & Uri.BuildQueryString([
        client_id = client_id,
        redirect_uri = redirect_uri,
        scope = GetScopeString(scopes, scope_prefix),
        response_type = "code",
        response_mode = "query",
        login = "login"
    ])
in
    authorizeUrl

```

Close the Advanced Query Editor to see the generated authorization URL.



Launch [Fiddler](#), and copy and paste the URL into the browser of your choice.

NOTE

You'll need to configure Fiddler to decrypt HTTPS traffic and skip decryption for the following hosts:

```
msft.sts.microsoft.com .
```

Entering the URL should bring up the standard Azure Active Directory login page. Complete the auth flow using your regular credentials, and then look at the fiddler trace. You'll be interested in the lines with a status of 302 and a host value of `login.microsoftonline.com`.

#	Overall_Elapsed	Result	Protocol	Host	URL
12	0:00:00.233	200	HTTPS	login.microsoftonline.com	/common/oauth2/v2.0/authorize?client_id=d2f23...
43	0:00:00.346	302	HTTPS	login.microsoftonline.com	/common/reprocess?sessionId=2c8b5939-45b3-4...
65	0:00:00.387	302	HTTPS	login.microsoftonline.com	/login.srf

Select the request to `/login.srf` and view the Headers of the Response. Under Transport, you'll find a location header with the `redirect_uri` value you specified in your code, and a query string containing a very long code value. Extract the code value only, and paste it into your M query as a new variable. Note, the header value will also contain a `&session_state=xxxx` query string value at the end—remove this part from the code. Also, be sure to include double quotes around the value after you paste it into the advanced query editor.

To exchange the code for an auth token, you'll need to create a POST request to the token endpoint. You can do this using the `Web.Contents` call, and use the `Uri.BuildQueryString` function to format your input parameters. The code will look like this:

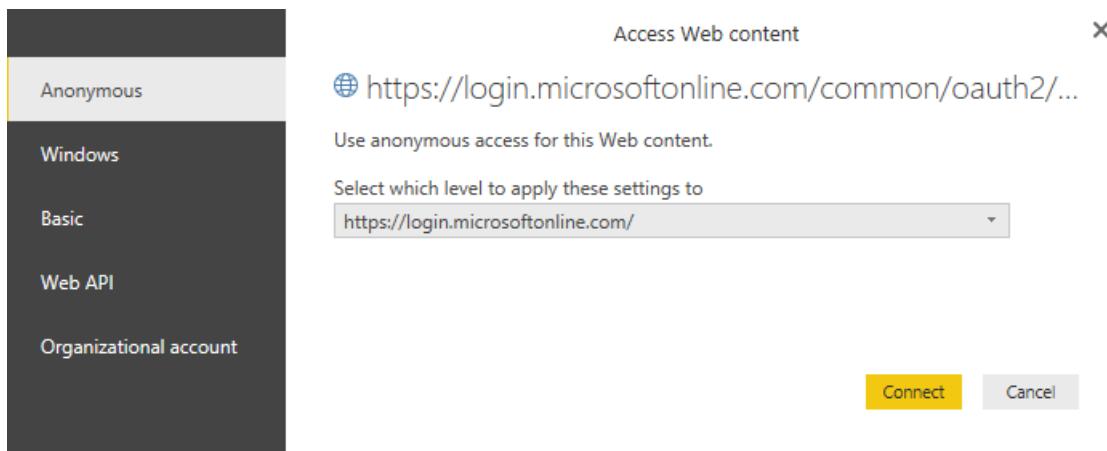
```

tokenResponse = Web.Contents(token_uri, [
    Content = Text.ToBinary(Uri.BuildQueryString([
        client_id = client_id,
        code = code,
        scope = GetScopeString(scopes, scope_prefix),
        grant_type = "authorization_code",
        redirect_uri = redirect_uri])),
    Headers = [
        #"Content-type" = "application/x-www-form-urlencoded",
        #"Accept" = "application/json"
    ]
]),

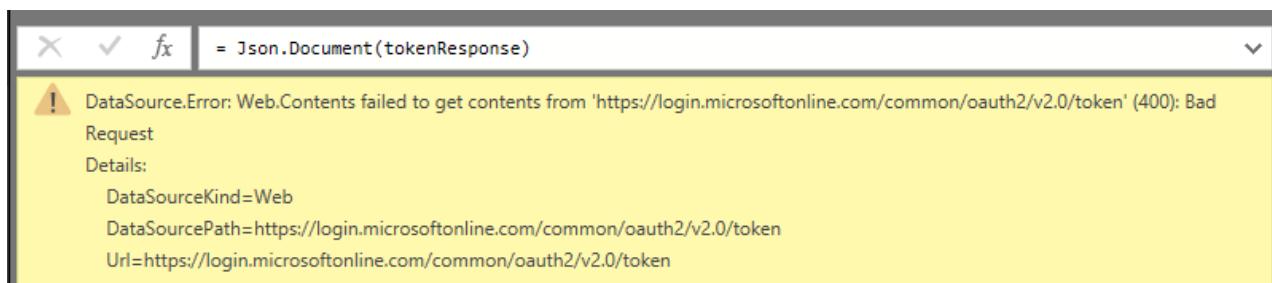
jsonResponse = Json.Document(tokenResponse)

```

When you return the `jsonResponse`, Power Query will likely prompt you for credentials. Choose **Anonymous**, and select **OK**.



The authentication code returned by AAD has a short timeout—probably shorter than the time it took you to do the previous steps. If your code has expired, you'll see a response like this:



If you check the fiddler trace, you'll see a more detailed error message in the JSON body of the response related to timeout. Later in this sample you'll update your code so that end users will be able to see the detailed error messages instead of the generic 400 Bad Request error. Try the authentication process again (you'll likely want your browser to be In Private mode to avoid any stored auth information). Capture the `Location` header, and update your M query with the new code value. Run the query again, and you should see a parsed record containing an `access_token` value.

<code>token_type</code>	Bearer
<code>scope</code>	https://graph.microsoft.com/calendars.readwrite https://graph.microsoft.com/contacts.read https://graph.microsoft.com/
<code>expires_in</code>	3600
<code>ext_expires_in</code>	0
<code>access_token</code>	

You now have the raw code you'll need to implement your OAuth flow. As an optional step, you can improve the

error handling of your OAuth code by using the `ManualStatusHandling` option to `Web.Contents`. This will let you process the body of an error response (which is a JSON document with `[error]` and `[error_description]` fields), rather than displaying a `DataSource.Error` to the user. The updated code looks like this:

```
tokenResponse = Web.Contents(token_uri, [
    Content = Text.ToBinary(Uri.BuildQueryString([
        client_id = client_id,
        code = code,
        scope = GetScopeString(scopes, scope_prefix),
        grant_type = "authorization_code",
        redirect_uri = redirect_uri])),
    Headers = [
        #"Content-type" = "application/x-www-form-urlencoded",
        #"Accept" = "application/json"
    ],
    ManualStatusHandling = {400}
]),
body = Json.Document(tokenResponse),
result = if (Record.HasFields(body, {"error", "error_description"})) then
    error Error.Record(body[error], body[error_description], body)
else
    body
```

Run your query again and you'll receive an error (because your code was already exchanged for an auth token). This time you should see the full detailed error message from the service, rather than a generic 400 status code.



Creating Your Graph Connector

Take a copy of the code contained within this sample, and open the `MyGraph.mproj` project file in Visual Studio. Update the `client_id` file with the AAD `client_id` you received when you registered your own app. You'll likely notice that the code is very similar to the OAuth sample code above, with some key differences that will be described below. There are also slight formatting differences due to the code being within a section document (rather than query expression).

Another difference is the `MyGraph.Feed` function. This will be the data source function you'll expose to the engine. You'll be adding your logic to access and read Graph data in here. You've associated the function with the `MyGraph` Data Source Kind, and exposed it in the UI using the `MyGraph.UI` record (`[DataSource.Kind="MyGraph", Publish="MyGraph.UI"]`).

Since your data source function has no required arguments, it acts as a `Singleton` data source credential type. This means that a user will have a single credential for the data source, and that the credential isn't dependent on any of the parameters supplied to the function.

In the following sample, you declare that `OAuth` is one of your supported credential types and provide function names for the OAuth interface functions.

```

[DataSource.Kind="MyGraph", Publish="MyGraph.UI"]
MyGraph.Feed = () =>
    let
        source = OData.Feed("https://graph.microsoft.com/v1.0/me/", null, [ ODataVersion = 4, MoreColumns =
true ])
    in
        source;

//
// Data Source definition
//
MyGraph = [
    Authentication = [
        OAuth = [
            StartLogin=StartLogin,
            FinishLogin=FinishLogin,
            Refresh=Refresh,
            Logout=Logout
        ]
    ],
    Label = "My Graph Connector"
];

//
// UI Export definition
//
MyGraph.UI = [
    Beta = true,
    ButtonText = { "MyGraph.Feed", "Connect to Graph" },
    SourceImage = MyGraph.Icons,
    SourceTypeImage = MyGraph.Icons
];

```

Implementing the OAuth Interface

The code that you wrote to test out the Graph OAuth flow in Power BI Desktop won't work in the connector as-is, but at least you've proven that it works (which is generally the trickiest part). Now you'll reformat the code to fill in the four functions expected by the M Engine's OAuth interface:

- StartLogin
- FinishLogin
- Refresh
- Logout

StartLogin

The first function you'll implement is `StartLogin`. This function will create the Authorization URL that will be sent to users to initiate their OAuth flow. The function signature must look like this:

```
StartLogin = (resourceUrl, state, display) as record
```

This function is expected to return a record with all the fields that Power BI will need to initiate an OAuth flow.

Since your data source function has no required parameters, you won't be making use of the `resourceUrl` value. If your data source function required a user supplied URL or sub-domain name, then this is where it would be passed to you. The `state` parameter includes a blob of state information that you're expected to include in the URL. You won't need to use the `display` value at all. The body of the function will look a lot like the `authorizeUrl` variable you created earlier in this sample—the main difference will be the inclusion of the `state` parameter (which is used to prevent [replay attacks](#)).

```

StartLogin = (resourceUrl, state, display) =>
  let
    authorizeUrl = authorize_uri & "?" & Uri.BuildQueryString([
      client_id = client_id,
      redirect_uri = redirect_uri,
      state = state,
      scope = "offline_access " & GetScopeString(scopes, scope_prefix),
      response_type = "code",
      response_mode = "query",
      login = "login"
    ])
  in
  [
    LoginUri = authorizeUrl,
    CallbackUri = redirect_uri,
    WindowHeight = 720,
    WindowWidth = 1024,
    Context = null
  ];

```

FinishLogin

The `FinishLogin` function will be called once the user has completed their OAuth flow. Its signature looks like this:

```
FinishLogin = (context, callbackUri, state) as record
```

The `context` parameter will contain any value set in the `Context` field of the record returned by `startLogin`. Typically, this will be a tenant ID or other identifier that was extracted from the original resource URL. The `callbackUri` parameter contains the redirect value in the `Location` header, which you'll parse to extract the code value. The third parameter (`state`) can be used to round-trip state information to the service—you won't need to use it for AAD.

Use the `Uri.Parts` function to break apart the `callbackUri` value. For the AAD auth flow, all you'll care about is the `code` parameter in the query string.

```

FinishLogin = (context, callbackUri, state) =>
  let
    parts = Uri.Parts(callbackUri)[Query],
    result = if (Record.HasFields(parts, {"error", "error_description"})) then
      error Error.Record(parts[error], parts[error_description], parts)
    else
      TokenMethod("authorization_code", "code", parts[code])
  in
  result;

```

If the response doesn't contain `error` fields, pass the `code` query string parameter from the `Location` header to the `TokenMethod` function.

The `TokenMethod` function converts the `code` to an `access_token`. It's not a direct part of the OAuth interface, but it provides all the heavy lifting for the `FinishLogin` and `Refresh` functions. Its implementation is essentially the `tokenResponse` logic you created earlier with one small addition—you'll use a `grantType` variable rather than hardcoding the value to "authorization_code".

```

TokenMethod = (grantType, tokenField, code) =>
let
    queryString = [
        client_id = client_id,
        scope = "offline_access " & GetScopeString(scopes, scope_prefix),
        grant_type = grantType,
        redirect_uri = redirect_uri
    ],
    queryWithCode = Record.AddField(queryString, tokenField, code),

    tokenResponse = Web.Contents(token_uri, [
        Content = Text.ToBinary(Uri.BuildQueryString(queryWithCode)),
        Headers = [
            #"Content-type" = "application/x-www-form-urlencoded",
            #"Accept" = "application/json"
        ],
        ManualStatusHandling = {400}
]),
    body = Json.Document(tokenResponse),
    result = if (Record.HasFields(body, {"error", "error_description"})) then
        error Error.Record(body[error], body[error_description], body)
    else
        body
in
result;

```

Refresh

This function is called when the `access_token` expires—Power Query will use the `refresh_token` to retrieve a new `access_token`. The implementation here is just a call to `TokenMethod`, passing in the refresh token value rather than the code.

```
Refresh = (resourceUrl, refresh_token) => TokenMethod("refresh_token", "refresh_token", refresh_token);
```

Logout

The last function you need to implement is Logout. The logout implementation for AAD is very simple—you just return a fixed URL.

```
Logout = (token) => logout_uri;
```

Testing the Data Source Function

`MyGraph.Feed` contains your actual data source function logic. Since Graph is an OData v4 service, you can leverage the built-in `OData.Feed` function to do all the hard work for you (including query folding and generating a navigation table!).

```
[DataSource.Kind="MyGraph", Publish="MyGraph.UI"]
MyGraph.Feed = () =>
let
    source = OData.Feed("https://graph.microsoft.com/v1.0/me/", null, [ ODataVersion = 4, MoreColumns =
true ])
in
source;
```

Once your function is updated, make sure there are no syntax errors in your code (look for red squiggles). Also be sure to update your `client_id` file with your own AAD app ID. If there are no errors, open the `MyGraph.query.m` file.

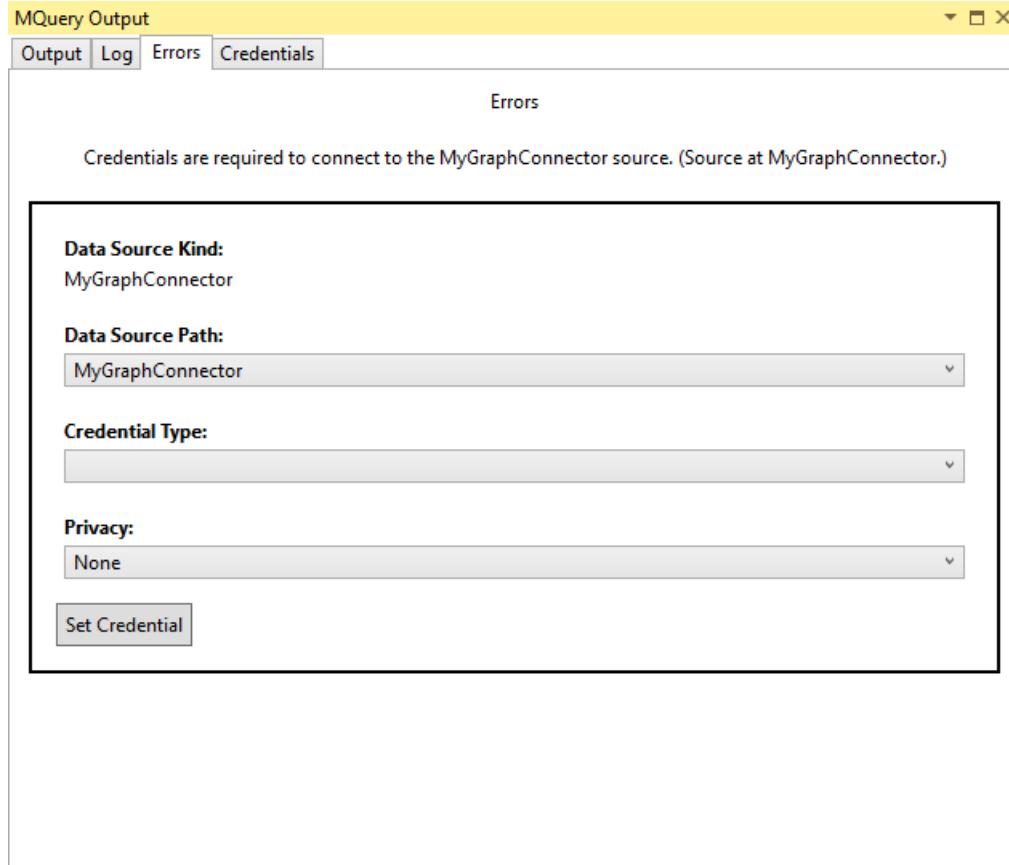
The `<project>.query.m` file lets you test out your extension. You (currently) don't get the same navigation table / query building experience you get in Power BI Desktop, but this does provide a quick way to test out your code.

A query to test your data source function would be:

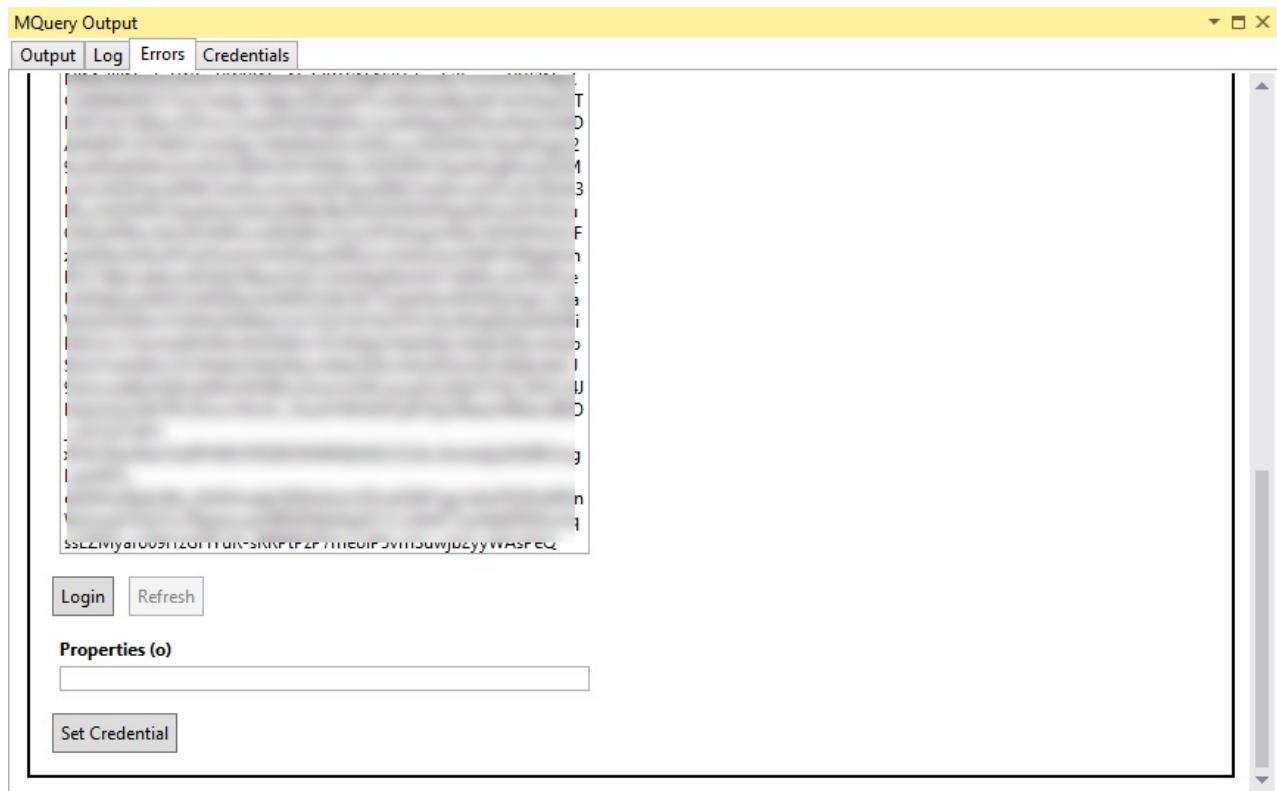
```
MyGraph.Feed()
```

Select the **Start (Debug)** button to execute the query.

Since this is the first time you are accessing your new data source, you'll receive a credential prompt.



Select OAuth2 from the Credential Type drop down, and then select **Login**. This will popup your OAuth flow. After completing your flow, you should see a large blob (your token). Select the **Set Credential** button at the bottom of the dialog, and close the MQuery Output window.



Run the query again. This time you should get a spinning progress dialog, and a query result window.

Query Result	
Name	Value
id	[Redacted]
accountEnabled	
assignedLicenses	[List]
assignedPlans	[List]
businessPhones	[List]
city	
companyName	
country	
department	
displayName	[Redacted]
givenName	[Redacted]
jobTitle	[Redacted]
mail	[Redacted]
mailNickname	
mobilePhone	[Redacted]
onPremisesImmutableId	
onPremisesLastSyncDateTime	
onPremisesSecurityIdentifier	
onPremisesSyncEnabled	
passwordPolicies	
passwordProfile	
officeLocation	[Redacted]
postalCode	

You can now build your project in Visual Studio to create a compiled extension file, and deploy it to your Custom Connectors directory. Your new data source should now appear in the **Get Data** dialog the next time you launch Power BI Desktop.

List of Samples

2 minutes to read • [Edit Online](#)

We maintain a list of samples on the DataConnectors repo on Github. Each of the links below links to a folder in the sample repository. Generally these folders include a readme, one or more .pq / .query.pq files, a project file for Visual Studio, and in some cases icons. To open these in Visual Studio, make sure you've set up the SDK properly, and run the .mpj file from the cloned or downloaded folder.

Functionality

SAMPLE	DESCRIPTION	LINK
Hello World	This very simple sample shows the basic structure of a connector.	Github Link
Hello World with Docs	Similar to the Hello World sample, this sample shows how to add documentation to a shared function.	Github Link
Navigation Tables	This sample provides two examples of how to create a navigation table for your data connector using the <code>Table.ToNavigationTable</code> function.	Github Link
Unit Testing	This sample shows how you can add simple unit testing to your <extension>.query.pq file.	Github Link
Relationships	This sample demonstrates the declaration of table relationships that will be detected by Power BI Desktop.	Github Link

OAuth

SAMPLE	DESCRIPTION	LINK
Github	This sample corresponds to the Github connector tutorial .	Github Link
MyGraph	This sample corresponds to the Microsoft Graph connector tutorial .	Github Link

ODBC

SAMPLE	DESCRIPTION	LINK
SQL	This connector sample serves as a template for ODBC connectors.	Github Link

SAMPLE	DESCRIPTION	LINK
Redshift	This connector sample uses the Redshift ODBC driver, and is based on the connector template.	Github Link
Hive LLAP	This connector sample uses the Hive ODBC driver, and is based on the connector template.	Github Link
Snowflake	This connector sample uses the Snowflake ODBC driver, and is based on the connector template.	Github Link
Impala	This connector sample uses the Cloudera Impala ODBC driver, and is based on the connector template.	Github Link
Direct Query for SQL	This sample creates an ODBC based custom connector that enables Direct Query for SQL Server.	Github Link

TripPin

SAMPLE	DESCRIPTION	LINK
Part 1	This sample corresponds to TripPin Tutorial Part 1 - OData .	Github Link
Part 2	This sample corresponds to TripPin Tutorial Part 2 - REST .	Github Link
Part 3	This sample corresponds to TripPin Tutorial Part 3 - Navigation Tables .	Github Link
Part 4	This sample corresponds to TripPin Tutorial Part 4 - Data Source Paths .	Github Link
Part 5	This sample corresponds to TripPin Tutorial Part 5 - Paging .	Github Link
Part 6	This sample corresponds to TripPin Tutorial Part 6 - Enforcing Schema .	Github Link
Part 7	This sample corresponds to TripPin Tutorial Part 7 - Advanced Schema with M Types .	Github Link
Part 8	This sample corresponds to TripPin Tutorial Part 8 - Adding Diagnostics .	Github Link
Part 9	This sample corresponds to TripPin Tutorial Part 9 - Test Connection .	Github Link

SAMPLE	DESCRIPTION	LINK
Part 10	This sample corresponds to TripPin Tutorial Part 10 - Query Folding Part 1.	Github Link

List of Samples

2 minutes to read • [Edit Online](#)

We maintain a list of samples on the DataConnectors repo on Github. Each of the links below links to a folder in the sample repository. Generally these folders include a readme, one or more .pq / .query.pq files, a project file for Visual Studio, and in some cases icons. To open these in Visual Studio, make sure you've set up the SDK properly, and run the .mpproj file from the cloned or downloaded folder.

Functionality

SAMPLE	DESCRIPTION	LINK
Hello World	This very simple sample shows the basic structure of a connector.	Github Link
Hello World with Docs	Similar to the Hello World sample, this sample shows how to add documentation to a shared function.	Github Link
Navigation Tables	This sample provides two examples of how to create a navigation table for your data connector using the <code>Table.ToNavigationTable</code> function.	Github Link
Unit Testing	This sample shows how you can add simple unit testing to your <extension>.query.pq file.	Github Link
Relationships	This sample demonstrates the declaration of table relationships that will be detected by Power BI Desktop.	Github Link

OAuth

SAMPLE	DESCRIPTION	LINK
Github	This sample corresponds to the Github connector tutorial .	Github Link
MyGraph	This sample corresponds to the Microsoft Graph connector tutorial .	Github Link

ODBC

SAMPLE	DESCRIPTION	LINK
SQL	This connector sample serves as a template for ODBC connectors.	Github Link

SAMPLE	DESCRIPTION	LINK
Redshift	This connector sample uses the Redshift ODBC driver, and is based on the connector template.	Github Link
Hive LLAP	This connector sample uses the Hive ODBC driver, and is based on the connector template.	Github Link
Snowflake	This connector sample uses the Snowflake ODBC driver, and is based on the connector template.	Github Link
Impala	This connector sample uses the Cloudera Impala ODBC driver, and is based on the connector template.	Github Link
Direct Query for SQL	This sample creates an ODBC based custom connector that enables Direct Query for SQL Server.	Github Link

TripPin

SAMPLE	DESCRIPTION	LINK
Part 1	This sample corresponds to TripPin Tutorial Part 1 - OData .	Github Link
Part 2	This sample corresponds to TripPin Tutorial Part 2 - REST .	Github Link
Part 3	This sample corresponds to TripPin Tutorial Part 3 - Navigation Tables .	Github Link
Part 4	This sample corresponds to TripPin Tutorial Part 4 - Data Source Paths .	Github Link
Part 5	This sample corresponds to TripPin Tutorial Part 5 - Paging .	Github Link
Part 6	This sample corresponds to TripPin Tutorial Part 6 - Enforcing Schema .	Github Link
Part 7	This sample corresponds to TripPin Tutorial Part 7 - Advanced Schema with M Types .	Github Link
Part 8	This sample corresponds to TripPin Tutorial Part 8 - Adding Diagnostics .	Github Link
Part 9	This sample corresponds to TripPin Tutorial Part 9 - Test Connection .	Github Link

SAMPLE	DESCRIPTION	LINK
Part 10	This sample corresponds to TripPin Tutorial Part 10 - Query Folding Part 1.	Github Link

List of Samples

2 minutes to read • [Edit Online](#)

We maintain a list of samples on the DataConnectors repo on Github. Each of the links below links to a folder in the sample repository. Generally these folders include a readme, one or more .pq / .query.pq files, a project file for Visual Studio, and in some cases icons. To open these in Visual Studio, make sure you've set up the SDK properly, and run the .mproj file from the cloned or downloaded folder.

Functionality

SAMPLE	DESCRIPTION	LINK
Hello World	This very simple sample shows the basic structure of a connector.	Github Link
Hello World with Docs	Similar to the Hello World sample, this sample shows how to add documentation to a shared function.	Github Link
Navigation Tables	This sample provides two examples of how to create a navigation table for your data connector using the <code>Table.ToNavigationTable</code> function.	Github Link
Unit Testing	This sample shows how you can add simple unit testing to your <extension>.query.pq file.	Github Link
Relationships	This sample demonstrates the declaration of table relationships that will be detected by Power BI Desktop.	Github Link

OAuth

SAMPLE	DESCRIPTION	LINK
Github	This sample corresponds to the Github connector tutorial .	Github Link
MyGraph	This sample corresponds to the Microsoft Graph connector tutorial .	Github Link

ODBC

SAMPLE	DESCRIPTION	LINK
SQL	This connector sample serves as a template for ODBC connectors.	Github Link

SAMPLE	DESCRIPTION	LINK
Redshift	This connector sample uses the Redshift ODBC driver, and is based on the connector template.	Github Link
Hive LLAP	This connector sample uses the Hive ODBC driver, and is based on the connector template.	Github Link
Snowflake	This connector sample uses the Snowflake ODBC driver, and is based on the connector template.	Github Link
Impala	This connector sample uses the Cloudera Impala ODBC driver, and is based on the connector template.	Github Link
Direct Query for SQL	This sample creates an ODBC based custom connector that enables Direct Query for SQL Server.	Github Link

TripPin

SAMPLE	DESCRIPTION	LINK
Part 1	This sample corresponds to TripPin Tutorial Part 1 - OData .	Github Link
Part 2	This sample corresponds to TripPin Tutorial Part 2 - REST .	Github Link
Part 3	This sample corresponds to TripPin Tutorial Part 3 - Navigation Tables .	Github Link
Part 4	This sample corresponds to TripPin Tutorial Part 4 - Data Source Paths .	Github Link
Part 5	This sample corresponds to TripPin Tutorial Part 5 - Paging .	Github Link
Part 6	This sample corresponds to TripPin Tutorial Part 6 - Enforcing Schema .	Github Link
Part 7	This sample corresponds to TripPin Tutorial Part 7 - Advanced Schema with M Types .	Github Link
Part 8	This sample corresponds to TripPin Tutorial Part 8 - Adding Diagnostics .	Github Link
Part 9	This sample corresponds to TripPin Tutorial Part 9 - Test Connection .	Github Link

SAMPLE	DESCRIPTION	LINK
Part 10	This sample corresponds to TripPin Tutorial Part 10 - Query Folding Part 1.	Github Link

Connector Certification

3 minutes to read • [Edit Online](#)

When developing a custom connector, or if you've been given one by another developer or vendor, you'll notice that they require you to lower the security settings in Power BI to use them. This is due to the fact that M is a versatile language that (as seen in [Handling Authentication](#)) has the capacity to interact with stored credentials. This means that we needed to give end users a way to only allow certified connectors to run.

The 'Connector Certification' program is a program in which Microsoft works with vendors to extend the data connectivity capabilities of Power BI.

Certified connectors are:

- Certified by Microsoft
- Distributed by Microsoft
- Maintained by the developer
- Supported by the developer

We work with vendors to try to make sure that they have support in maintenance, but customer issues with the connector itself will be directed to the developer.

We have a certain set of requirements for certification. We recognize that not every developer can meet these requirements, and as above we're hoping to introduce a feature set that will handle their needs in short order.

Cut-off dates for certification are:

- Notification of a submission on the 15th of the month, two months before the targeted Power BI desktop release. In other words for April Power BI, the cut-off is February 15th.
- Submission must be done by the 1st of the month before the targeted Power BI desktop release. For April Power BI, the cut-off for submission would be March 1st.
- Technical review must be passed by the 15th of the month before the targeted Power BI desktop release. For April Power BI, the cut-off for technical review would be March 15th.

Due to backlogs, delays, rearchitecture, and testing issues, we highly recommend a long lead time for your initial release, and to very carefully read our requirements below. If you feel like your connector is important to deliver to a few connectors with minimal overhead, we recommend self signing and providing it that way.

Certification Requirements

Before starting

- Developer must own the data source or have recorded permission from the owner of the data source to develop a connector for it.
- The connector must be for a public product.
- The developer must provide an estimate for usage. We suggest that developers of connectors for very boutique products use our Connector Signing capabilities and provide them directly to the customer.
- Developer must submit to the Connector Certification Portal. In the future this will be available via standard Microsoft ISV channels. Currently you are required to reach out to dataconnectors@microsoft.com first to connect with the team.

Artifacts

- PBIX file

- Report should contain one or more queries to test each item in their navigation table.
- If you don't have a set schema (as an example, databases), you should include a query for each 'type' of table you're concerned with.
- .mez file
 - The .mez file should follow style standards. For example, use Product.mez rather than Product_PowerBI_Connector.mez.
- Test account
 - The test account will be reused whenever we're troubleshooting or certifying updates, so if you have a persistent test account it would be best to find a way to share this.
- Link to external dependencies (ODBC drivers, for example)
- Documentation on how to use the connector if needed

Features and Style

- Connector MUST use Section document format
- Connector MUST have [Version adornment](#) on section
- Connector MUST provide function documentation metadata
- Connector MUST have [TestConnection handler](#)
- Connector MUST follow naming conventions (DataSourceKind.FunctionName)
- FunctionName should make sense for their domain - generally "Contents", "Tables", "Document", "Databases" ...
- Connector SHOULD have icons
- Connector SHOULD provide a navigation table
- Connector SHOULD place strings in resources.resx file

Security

- If using Extension.CurrentCredentials() ...
 - Is the usage required? If so, where do the credentials get sent to?
 - Are the requests guaranteed to be made through HTTPS?
 - You can use the [HTTPS enforcement helper function](#).
 - If the credentials are sent using Web.Contents() via GET ...
 - Can it be turned into a POST?
 - If GET is required, connector MUST use the CredentialQueryString record in the Web.Contents() options record to pass in sensitive credentials
- If [Diagnostics.* functions](#) are used ...
 - Validate what is being traced—it MUST NOT
 - Contain PII
 - Contain large amounts of data
 - If you implemented significant tracing in development, you should attach it to a variable that checks if tracing should be on or not, and you should turn it off before shipping.
- If Expression.Evaluate() is used ...
 - Validate where the expression is coming from / what it is (that is, can dynamically construct calls to Extension.CurrentCredentials(), and so on ...)
 - Expression should not be user provided / take user input
 - Expression should not be dynamic (i.e. retrieved from a web call)

Getting your connector certified

If you'd like to reach out about connector certification, please contact us at dataconnectors@microsoft.com.

Power Query Online Limits

2 minutes to read • [Edit Online](#)

Summary

Power Query Online is integrated into a variety of Microsoft products. Since these products target different scenarios, they may set different limits for Power Query Online usage.

Limits are enforced at the beginning of query evaluations. Once an evaluation is underway, only timeout limits are imposed.

Limit Types

Hourly Evaluation Count: The maximum number of evaluation requests a user can issue during any 60 minute period

Daily Evaluation Time: The net time a user can spend evaluating queries during any 24 hour period

Concurrent Evaluations: The maximum number of evaluations a user can have running at any given time

Authoring Limits

Authoring limits are the same across all products. During authoring, query evaluations return previews that may be subsets of the data. Data is not persisted.

Hourly Evaluation Count: 1000

Daily Evaluation Time: Currently unrestricted

Per Query Timeout: 10 minutes

Refresh Limits

During refresh (either scheduled or on-demand), query evaluations return complete results. Data is typically persisted in storage.

PRODUCT INTEGRATION	HOURLY EVALUATION COUNT (#)	DAILY EVALUATION TIME (HOURS)	CONCURRENT EVALUATIONS (#)
Microsoft Flow (SQL Connector—Transform Data Using Power Query)	500	2	5
Dataflows in PowerApps.com (Trial)	500	2	5
Dataflows in PowerApps.com (Production)	1000	8	10
Data Integration in PowerApps.com Admin Portal	1000	24	10
Dataflows in PowerBI.com	1000	100	10

Power Query query folding

4 minutes to read • [Edit Online](#)

This article targets data modelers developing models in Power Pivot or Power BI Desktop. It describes what Power Query query folding is, and why it is important in your data model designs. It also describes the data sources and transformations that can achieve query folding, and how to determine that your Power Query queries can be folded—whether fully or partially.

Query folding is the ability for a Power Query query to generate a single query statement to retrieve and transform source data. The Power Query mashup engine strives to achieve query folding whenever possible for reasons of efficiency.

Query folding is an important topic for data modeling for several reasons:

- **Import model tables:** Data refresh will take place efficiently for Import model tables (Power Pivot or Power BI Desktop), in terms of resource utilization and refresh duration.
- **DirectQuery and Dual storage mode tables:** Each DirectQuery and Dual storage mode table (Power BI only) must be based on a Power Query query that can be folded.
- **Incremental refresh:** Incremental data refresh (Power BI only) will be efficient, in terms of resource utilization and refresh duration. In fact, the Power BI **Incremental Refresh** configuration window will notify you of a warning should it determine that query folding for the table cannot be achieved. If it cannot be achieved, the objective of incremental refresh is defeated. The mashup engine would then be required to retrieve all source rows, and then apply filters to determine incremental changes.

Query folding may occur for an entire Power Query query, or for a subset of its steps. When query folding cannot be achieved—either partially or fully—the Power Query mashup engine must compensate by processing data transformations itself. This process can involve retrieving source query results, which for large datasets is very resource intensive and slow.

We recommend that you strive to achieve efficiency in your model designs by ensuring query folding occurs whenever possible.

Sources that support folding

Most data sources that have the concept of a query language support query folding. These data sources can include relational databases, OData feeds (including SharePoint lists), Exchange, and Active Directory. However, data sources like flat files, blobs, and web typically do not.

Transformations that can achieve folding

Relational data source transformations that can be query folded are those that can be written as a single SELECT statement. A SELECT statement can be constructed with appropriate WHERE, GROUP BY, and JOIN clauses. It can also contain column expressions (calculations) that use common built-in functions supported by SQL databases.

Generally, the following list describes transformations that can be query folded.

- Removing columns.
- Renaming columns (SELECT column aliases).
- Filtering rows, with static values or Power Query parameters (WHERE clause predicates).
- Grouping and summarizing (GROUP BY clause).

- Expanding record columns (source foreign key columns) to achieve a join of two source tables (JOIN clause).
- Non-fuzzy merging of foldable queries based on the same source (JOIN clause).
- Appending foldable queries based on the same source (UNION ALL operator).
- Adding custom columns with *simple logic* (SELECT column expressions). Simple logic implies uncomplicated operations, possibly including the use of M functions that have equivalent functions in the SQL data source, like mathematic or text manipulation functions. For example, the following expression returns the year component of the **OrderDate** column value (to return a numeric value).

```
Date.Year([OrderDate])
```

- Pivoting and unpivoting (PIVOT and UNPIVOT operators).

Transformations that prevent folding

Generally, the following list describes transformations that prevent query folding. This is not intended to be an exhaustive list.

- Merging queries based on different sources.
- Appending (union-ing) queries based on different sources.
- Adding custom columns with *complex logic*. Complex logic implies the use of M functions that have no equivalent functions in the data source. For example, the following expression formats the **OrderDate** column value (to return a text value).

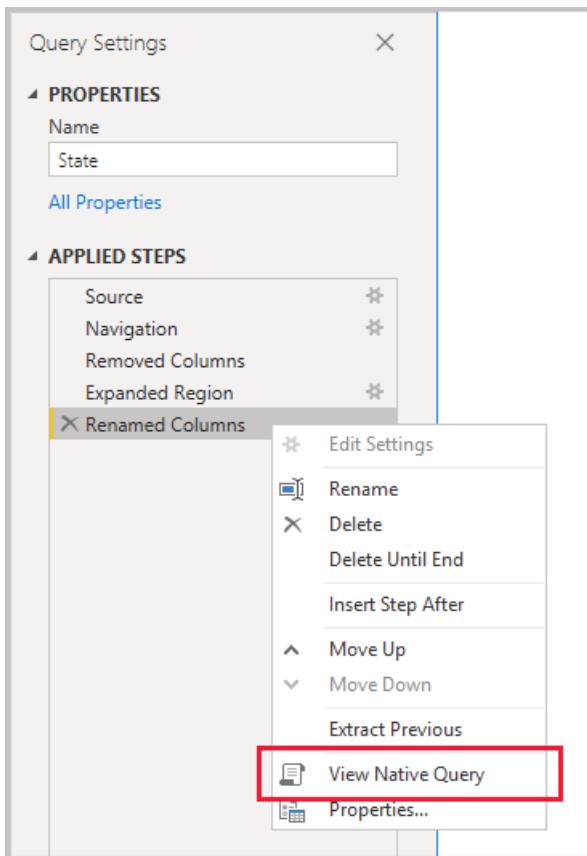
```
Date.ToString([OrderDate], "yyyy")
```

- Adding index columns.
- Changing a column data type.

Note that when a Power Query query encompasses multiple data sources, incompatibility of data source privacy levels can prevent query folding from taking place. For more information, see the [Power BI Desktop privacy levels](#) article.

Determine when a query can be folded

In the Power Query Editor window, it is possible to determine when a Power Query query can be folded. In the **Query Settings** pane, when you right-click the last applied step, if the **View Native Query** option is enabled (not greyed out), then the entire query can be folded.



NOTE

The **View Native Query** option is only available for certain relational DB/SQL generating connectors. It doesn't work for OData based connectors, for example, even though there is folding occurring on the backend. The Query Diagnostics feature is the best way to see what folding has occurred for non-SQL connectors (although the steps that fold aren't explicitly called out—you just see the resulting URL that was generated).

To view the folded query, you select the **View Native Query** option. You are then presented with the native query that Power Query will use to source data.

Native Query

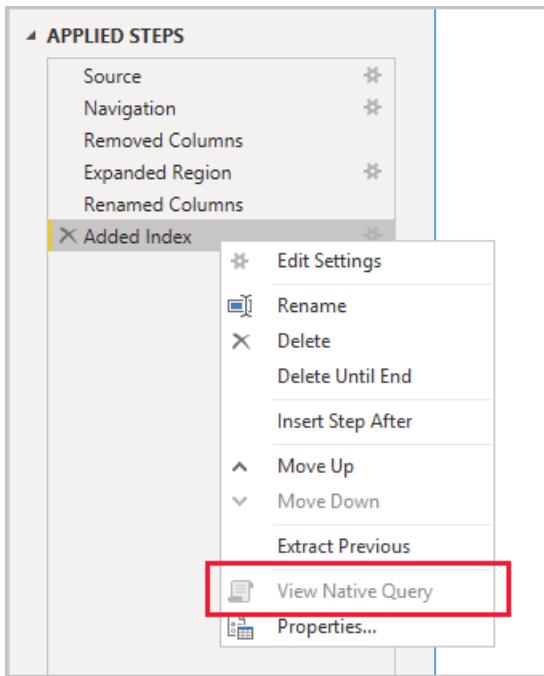
```

select [$Outer].[StateID] as [StateID],
[$Outer].[StateCode] as [State Code],
[$Outer].[StateName] as [State],
[$Outer].[TimeZone] as [Time Zone],
[$Inner].[RegionName] as [Region]
from
(
    select [__].[StateID] as [StateID],
    [__].[StateCode] as [StateCode],
    [__].[StateName] as [StateName],
    [__].[RegionID] as [RegionID2],
    [__].[TimeZone] as [TimeZone]
    from [dbo].[State] as [__]
) as [$Outer]
left outer join [dbo].[Region] as [$Inner] on ([$Outer].[RegionID2] = [$Inner].[RegionID])

```

OK

If the **View Native Query** option is not enabled (greyed out), this is evidence that all query steps cannot be folded. However, it could mean that a subset of steps can still be folded. Working backwards from the last step, you can check each step to see if the **View Native Query** option is enabled. If this is the case, then you have learned where, in the sequence of steps, that query folding could no longer be achieved.



Next steps

For more information about Query Folding and related topics, check out the following resources:

- [Best practice guidance for query folding](#)
- [Use composite models in Power BI Desktop](#)
- [Incremental refresh in Power BI Premium](#)
- [Using Table.View to Implement Query Folding](#)

Combine Files in Power Query

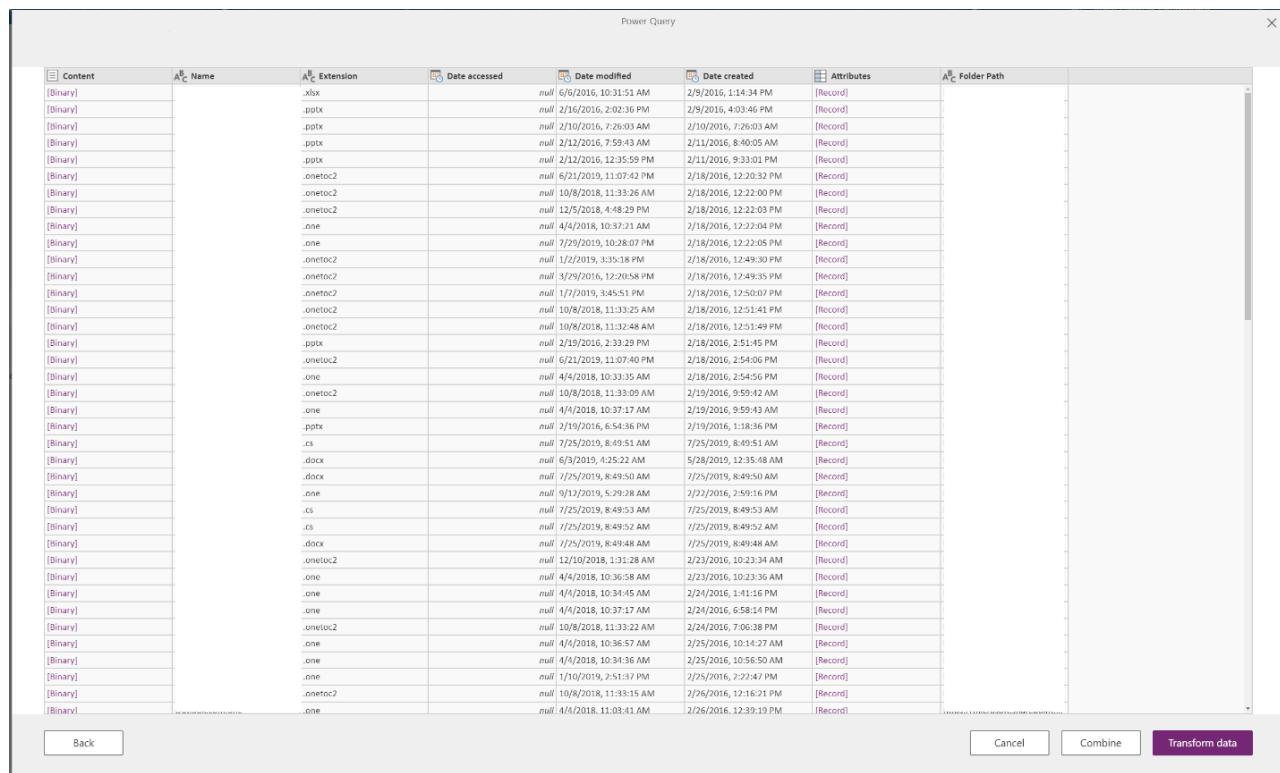
2 minutes to read • [Edit Online](#)

Applies to: Power Query Desktop, Power Query Online

One powerful capability to import data using Power Query is to combine multiple files from a folder, which have the same schema, into a single logical table. To start the process of combining files from the same folder, select one of the folder-based connectors (for example, Folder or SharePoint Folder) within Power Query's **Get Data** dialog.

Combine files step-by-step experience

Upon specifying the local folder or SharePoint site, you'll be taken into the **Navigator** dialog (in Power Query Desktop) or the **Choose data** screen (in Power Query Online), where you can opt into the "Combine" option.



The screenshot shows the Power Query Navigator dialog. It displays a table of file metadata with the following columns:

Content	Name	Extension	Date accessed	Date modified	Date created	Attributes	Folder Path
[Binary]	.xlsx	null	6/10/2016, 10:31:51 AM	2/9/2016, 1:14:34 PM	[Record]		
[Binary]	.potx	null	2/16/2016, 2:02:36 PM	2/9/2016, 4:03:46 PM	[Record]		
[Binary]	.potx	null	2/10/2016, 7:26:03 AM	2/10/2016, 7:26:03 AM	[Record]		
[Binary]	.potx	null	2/12/2016, 7:59:43 AM	2/11/2016, 8:40:05 AM	[Record]		
[Binary]	.potx	null	2/12/2016, 12:35:59 PM	2/11/2016, 9:33:01 PM	[Record]		
[Binary]	.onetoc2	null	6/21/2019, 11:07:42 PM	2/18/2016, 12:20:52 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:35:26 AM	2/18/2016, 12:22:00 PM	[Record]		
[Binary]	.onetoc2	null	12/5/2018, 4:48:29 PM	2/18/2016, 12:22:03 PM	[Record]		
[Binary]	.one	null	4/4/2018, 10:37:21 AM	2/18/2016, 12:22:04 PM	[Record]		
[Binary]	.one	null	7/29/2019, 10:28:07 PM	2/18/2016, 12:22:05 PM	[Record]		
[Binary]	.onetoc2	null	1/2/2019, 3:35:18 PM	2/18/2016, 12:49:30 PM	[Record]		
[Binary]	.onetoc2	null	3/29/2016, 12:20:58 PM	2/18/2016, 12:49:35 PM	[Record]		
[Binary]	.onetoc2	null	1/7/2019, 3:45:51 PM	2/18/2016, 12:50:00 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:33:25 AM	2/18/2016, 12:51:41 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:32:48 AM	2/18/2016, 12:51:49 PM	[Record]		
[Binary]	.potx	null	2/19/2016, 2:33:29 PM	2/18/2016, 2:51:45 PM	[Record]		
[Binary]	.onetoc2	null	6/2/2019, 11:07:40 PM	2/18/2016, 2:54:06 PM	[Record]		
[Binary]	.one	null	4/4/2018, 10:33:35 AM	2/18/2016, 2:54:56 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:33:09 AM	2/19/2016, 6:59:42 AM	[Record]		
[Binary]	.one	null	4/4/2018, 10:37:17 AM	2/19/2016, 6:59:43 AM	[Record]		
[Binary]	.potx	null	2/19/2016, 6:54:36 PM	2/19/2016, 1:18:36 PM	[Record]		
[Binary]	.cs	null	7/25/2019, 8:49:51 AM	7/25/2019, 8:49:51 AM	[Record]		
[Binary]	.docx	null	6/9/2019, 4:25:22 AM	5/28/2019, 12:35:48 AM	[Record]		
[Binary]	.docx	null	7/25/2019, 8:49:50 AM	1/25/2019, 8:49:50 AM	[Record]		
[Binary]	.one	null	9/13/2019, 5:29:38 AM	2/22/2016, 2:59:16 PM	[Record]		
[Binary]	.cs	null	7/25/2019, 8:49:53 AM	7/25/2019, 8:49:53 AM	[Record]		
[Binary]	.cs	null	7/25/2019, 8:49:52 AM	7/25/2019, 8:49:52 AM	[Record]		
[Binary]	.docx	null	7/25/2019, 8:49:48 AM	1/25/2019, 8:49:48 AM	[Record]		
[Binary]	.onetoc2	null	12/10/2018, 1:31:28 AM	2/23/2016, 10:23:34 AM	[Record]		
[Binary]	.one	null	4/4/2018, 10:36:58 AM	2/23/2016, 10:23:36 AM	[Record]		
[Binary]	.one	null	4/4/2018, 10:34:45 AM	2/24/2016, 1:41:16 PM	[Record]		
[Binary]	.one	null	4/4/2018, 10:37:17 AM	2/24/2016, 6:58:14 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:33:22 AM	2/24/2016, 7:06:38 PM	[Record]		
[Binary]	.one	null	4/4/2018, 10:36:57 AM	2/25/2016, 10:14:27 AM	[Record]		
[Binary]	.one	null	4/4/2018, 10:34:36 AM	2/25/2016, 10:56:50 AM	[Record]		
[Binary]	.one	null	1/10/2019, 2:51:37 PM	2/25/2016, 2:22:47 PM	[Record]		
[Binary]	.onetoc2	null	10/8/2018, 11:33:15 AM	2/26/2016, 12:16:21 PM	[Record]		
[Binary]	.one	null	4/4/2018, 11:03:41 AM	2/26/2016, 12:39:19 PM	[Record]		

At the bottom of the dialog, there are three buttons: **Back**, **Cancel**, **Combine**, and **Transform data**.

Alternatively, you can select **Transform Data** to access the Power Query Editor where the list of files can be subset (for example, using filters within the Folder Path or any of the other columns in this table) and then *Combine Files* by selecting the column containing the binaries (for example, Content) and selecting the **Combine Files** button in the Power Query Desktop ribbon (**Home** tab) or the Power Query Online toolbar (under **Combine** menu).

Power Query

Edit queries

Get data Refresh Options Manage columns Transform column Transform table Reduce rows Add column Go to column Combine tables

Query

Content [+] Name Extension Date accessed Date modified Date created Merge queries as new Append queries Append queries as new Combine files

	[Binary]	2019-01.xlsx	.xlsx	null	10/15/2019, 11:07:22 AM	10/15/2019,	ps://microsoft.sha
1	[Binary]	2019-02.xlsx	.xlsx	null	10/15/2019, 11:07:22 AM	10/15/2019,	ps://microsoft.sha
2	[Binary]	2019-03.xlsx	.xlsx	null	10/15/2019, 11:07:23 AM	10/15/2019,	ps://microsoft.sha
3	[Binary]	2019-04.xlsx	.xlsx	null	10/15/2019, 11:07:23 AM	10/15/2019,	ps://microsoft.sha
4	[Binary]	2019-05.xlsx	.xlsx	null	10/15/2019, 11:07:23 AM	10/15/2019,	ps://microsoft.sha
5	[Binary]	2019-06.xlsx	.xlsx	null	10/15/2019, 11:07:23 AM	10/15/2019,	ps://microsoft.sha
6	[Binary]	2019-07.xlsx	.xlsx	null	10/15/2019, 11:07:24 AM	10/15/2019,	ps://microsoft.sha
7	[Binary]	2019-08.xlsx	.xlsx	null	10/15/2019, 11:07:24 AM	10/15/2019, 11:07:24 ...	[Record] https://microsoft.sha
8	[Binary]	2019-09.xlsx	.xlsx	null	10/15/2019, 11:07:25 AM	10/15/2019, 11:07:25 ...	[Record] https://microsoft.sha
9	[Binary]	2019-09.xlsx	.xlsx	null	10/15/2019, 11:07:25 AM	10/15/2019, 11:07:25 ...	[Record] https://microsoft.sha

Name: Query
Source: Filtered rows
Applied steps: Folder Path

Next

The combine files experience behaves as follows:

- Power Query analyzes each input file, and determines the correct file connector to use in order to open the file contents, such as text, Excel workbook, or JSON file.
- Within the **Combine Files** dialog, Power Query lets you select a specific object from the first file, for example, an Excel worksheet, to extract.
 - You can also pick a different example file instead of the first file, by using the **Example File** dropdown menu.
 - Optionally, you can exclude files that result in errors from the final output.

Combine files

Select the object to be extracted from each file. [Learn more](#)

Example file:

First file ▾

Search

Display options ▾



Parameter [3]

Orders

Sheet1

Sheet2

Orders

ABC 123 OrderID	ABC 123 CustomerID	ABC 123 EmployeeID
10,400	EASTC	
10,401	RATTC	
10,402	ERNSH	
10,403	ERNSH	
10,404	MAGAA	
10,405	LINOD	
10,406	QUEEN	
10,407	OTTIK	
10,408	FOLIG	
10,409	OCEAN	
10,410	BOTTM	
10,411	BOTTM	
10,412	WARTH	
10,413	LAMAI	
10,414	FAMIA	
10,415	HUNGC	
10,416	WARTH	
10,417	SIMOB	
10,418	QUICK	

Skip files with errors

OK

Cancel

Power Query then automatically performs the following actions:

1. Creates an exemplar query that performs all the required extraction steps in a single file. It uses the file that was selected as the sample file in the **Combine Files** dialog.
2. Creates a function query that parameterizes the file/binary input to the exemplar query. The exemplar query and the function query are linked, so that changes to the exemplar query are reflected in the function query.
3. Applies the function query to the original query with input binaries (for example, the Folder query) so it applies the function query for binary inputs on each row, then expands the resulting data extraction as top-level columns.
4. Note that in addition to the above queries, a new "Sample query" group is created with two helper queries that reference the sample file used.

Power Query

Edit queries

Get data Refresh Options Manage columns Transform table Reduce rows Add column Go to column Combine tables

Transform file from... Sample query (2) Transform file from... Transform file from... Query

Table.TransformColumnTypes(#"Expanded table column", {{"Source.Name", type text}, {"OrderID", Int64.Type}, {"CustomerID", type text}, {"EmployeeID", type text}, {"OrderDate", type date}, {"ShipVia", type number}, {"Freight", type number}, {"ShipName", type text}, {"ShipAddress", type text}}

Source Name: Query

Applied steps

- Source
- Filtered rows
- Filtered hidden files
- Invoke custom function
- Renamed columns
- Removed other colu...
- Expanded table column
- Changed column type

OrderID	CustomerID	EmployeeID	OrderDate	ShipVia	Freight	ShipName	ShipAddress
10400	EASTC		1/1/2019	3	83.93	Eastern Connection	35 King George
10401	RATTI		1/1/2019	1	12.51	Rattlesnake Canyon Grocery	2817 Milton Dr.
10402	ERNSH		8/1/2019	2	67.89	Ernst Handel	Kirchgasse 6
10403	ERNSH		4/3/2019	3	73.70	Ernst Handel	Kirchgasse 6
10404	MAGAA		2/3/2019	1	155.97	Magazzini Alimentari Riuniti	Via Ludovico il Moro 22
10405	LINOD		1/6/2019	1	34.82	LINO-Delicatessen	Ave. 5 de Mayo Portamar
10406	QUEEN		7/17/2019	1	108.04	Queen Cozinha	Alameda dos Canários, 801
10407	OTTIK		2/17/2019	2	91.48	Ottiles Käseladen	Mehrheimerstr. 369
10408	FOUG		8/18/2019	1	11.26	Folies gourmandes	184, chaussée de Journa
10409	OCEAN		3/9/2019	1	29.83	Océano Atlântico Ltda.	Ing. Gustavo Moncada 8585 Piso 20-A
10410	BOTTM		3/10/2019	3	2.47	Bottom-Dollar Markets	23 Tsawassen Blvd.
10411	BOITM		9/1/2019	3	23.65	Bottom-Dollar Markets	23 Tsawassen Blvd.
10412	WARTH		8/13/2019	2	3.77	Wartian Herkku	Torikatu 38
10413	LAMAI		3/14/2019	2	95.66	La maison d'Asie	1 rue Alsace-Lorraine
10414	FAMILIA		2/14/2019	3	21.48	Familia Arquibaldo	Rua Orós, 92
10415	HUNGIC		3/15/2019	1	1	Hungry Coyote Import Store	City Center Plaza 516 Main St.
10416	WARTH		8/16/2019	3	22.72	Wartian Herkku	Torikatu 38
10417	SIMOB		4/1/2019	3	70.29	Simons bistro	Vinbaletet 34
10418	QUICK		4/1/2019	1	17.55	QUICK-Stop	Taucherstraße 10
10419	RICSU		4/20/2019	2	137.35	Richter Supermarkt	Sturenweg 5
10420	WELLU		3/21/2019	1	44.12	Wellington Importadora	Rua do Mercado, 12
10421	QUEDE		8/21/2019	1	99.23	Que Delicia	Rua da Panificadora, 12
10422	FRANS		2/22/2019	1	3.02	Franchi S.p.A.	Via Monte Bianco 34
10423	GOURL		6/1/2019	3	24.5	Gourmet Lanchonetes	Av Brasil, 442
10424	MEREPE		7/1/2019	2	370.61	Mère Pallarde	43 rue St. Laurent
10425	LAMAI		6/1/2019	2	7.93	La maison d'Asie	1 rue Alsace-Lorraine
10426	GALFO		4/7/2019	1	18.69	Galera do gastronomô	Rambá de Cataluña, 23
10427	PICCO		4/17/2019	2	31.29	Piccolo und mehr	Geistweg 14
10428	REGGIC		7/28/2019	1	11.09	Ruggiani Caseifici	Strada Provinciale 124
10429	HUNGIC		3/19/2019	2	56.63	Hungry Owl All-Night Grocers	8 Johnston Road
10430	ERNSH		4/1/2019	1	458.78	Ernst Handel	Kirchgasse 6
10431	BOTTM		4/1/2019	2	44.17	Bottom Dollar Markets	23 Tsawassen Blvd.
33	...						

Next

Using "Combine Files", you can easily combine all files within a given folder, as long as they have the same file type and structure (such as the same columns). In addition, you can easily apply additional transformation or extraction steps by modifying the automatically generated exemplar query, without having to worry about modifying or creating additional function query steps. Any changes to the exemplar query are automatically propagated to the linked function query.

Common Issues

2 minutes to read • [Edit Online](#)

Preserving Sort

In Power Query, you'll often want to sort your data before you perform some other operation. For example, if you wanted to sort a sales table by the Store ID and the sale amount, and then you wanted to perform a group, you might expect sort order to be preserved. However, due to how operation application works, sort order is not preserved through aggregations or joins.

If you sorted your table, applied an aggregation, and then you tried to apply a distinct to the original sort operation, you might be surprised to find out that you had lost both the first and the second sort. In other words, if you had two rows with sales for a single store, and you had them sorted in descending order so that the first row had a greater dollar value than the second, you might find that the second row was preserved when you ran a distinct on the Store ID.

There are ways to make this work via a smart combination of aggregations, but these aren't exposed by the user experience. Unfortunately, there are a sufficiently large number of possible transformations here that we can't give an example for all outcomes, but here is how you might address the problem above.

```
let
    Source = Sql.Database("Server", "AdventureWorks"),
    Sales_SalesPerson = Source{[Schema="Sales",Item="SalesPerson"]}[Data],
    #"Grouped Rows" = Table.Group(Sales_SalesPerson, {"TerritoryID"}, {"Rows", each _}),
    Custom1 = Table.TransformColumns(#"Grouped Rows", {"Rows", each Table.FirstN(Table.Sort(_, {"SalesYTD", Order.Descending}), 1)})}
in
    Custom1
```

The data you want is the entire record with the highest SalesYTD in each TerritoryID. If you only wanted the max, this would be a simple aggregation—but you want the entire input record. To get this, you need to group by TerritoryID and then sort inside each group, keeping the first record.

Data Type Inference

When you import a table, you may find that Power Query sometimes incorrectly detects a column's data type. One reason this can happen is that Power Query infers data types using only the first 200 rows of data. If the data in the first 200 rows is somehow different than the data after row 200, Power Query may detect an incorrect column type. This may or may not result in errors, which can make the incorrect type inference tricky to detect in some cases.

For example, imagine a column that contains integers in the first 200 rows (such as all zeroes), but contains decimal numbers after row 200. In this case, Power Query will infer the data type of the column to be Whole Number (Int64.Type). This will result in the decimal portions of any non-integer numbers being truncated.

Or imagine a column that contains textual date values in the first 200 rows, and other kinds of text values after row 200. In this case, Power Query will infer the data type of the column to be Date. This will result in the non-date text values being treated as type conversion errors.

Because type detection works on the first 200 rows, but Data Profiling can operate over the entire dataset, you can consider using the Data Profiling functionality to get an early indication in the Query Editor about Errors (from type detection or any number of other reasons) beyond the top N rows.

Excel

3 minutes to read • [Edit Online](#)

Summary

Release State: General Availability

Products: Power BI Desktop, Power BI Service (Gateway for on-premise or .xls files), Dataflows in PowerBI.com (Gateway for on-premise or .xls files), Dataflows in PowerApps.com (Gateway for on-premise or .xls files), Excel

Authentication Types Supported: No authentication

NOTE

Some capabilities may be present in one product but not others due to deployment schedules and host-specific capabilities.

Prerequisites

In order to connect to a legacy workbook (such as .xls or .xslb), the Access Database Engine 2010 OLEDB provider is required. To install this, go to the [download page](#) and install the relevant (32 or 64 bit) version. If you don't have it installed, when connecting to legacy workbooks you'll see the following error:

The 32-bit (or 64-bit) version of the Access Database Engine 2010 OLEDB provider may be required to read this type of file. To download the client software, visit the following site: <https://go.microsoft.com/fwlink/?LinkID=285987>.

Capabilities Supported

- Import

Connect to an Excel workbook

To connect to an Excel workbook, select 'Excel' from the product specific data connector list.

Troubleshooting

Legacy ACE Connector

Error resolution

Workbooks built in a legacy format (such as .xls and .xslb) are accessed through the Access Database Engine OLEDB provider, and Power Query will display values as returned by this provider. This may cause a lack of fidelity in certain cases compared to what you would see in an equivalent xlsx (OpenXML based) file.

Incorrect column typing returning nulls

When Ace loads a sheet, it looks at the first 8 rows to try to guess the data types to use. If the first 8 rows of data are not inclusive of the following data (for example, numeric only in the first 8 rows versus text in the following rows), ACE will apply an incorrect type to that column and return nulls for any data that does not match the type.

Excel Data Completeness

Under certain circumstances, users will run up against issues where Power Query fails to extract all the data from

an Excel Worksheet, or performance is severely degraded against a reasonably sized table. Both of these failures generally resolve to the same cause: improper cell range specification.

Incomplete Data Loading from Excel

Incomplete data loading from Excel is generally caused by a tool exporting an Excel document with an improper 'final cell' value. This will generally be fixed by opening and re-saving the document, but that doesn't explain the cause.

To see the source of the issue, you have to look at the underlying XML of the Worksheet.

1. Rename the xlsx file with a .zip extension
2. Navigate into xl\worksheets
3. Copy the xml file for the problematic sheet (for example, Sheet1.xml) out of the zip file to another location
4. Inspect the first few lines of the file a. If the file is small enough, simply open it in a text editor b. If the file is too large to be opened in a text editor, run the following from the DOS command-line: more Sheet1.xml
5. Look for a <dimension .../> tag

If your file has a dimension attribute that resembles <dimension ref="A1" />, we use it to find the starting row and column of the data on Sheet1, because it's only pointing at a single cell.

However, if your file has a dimension attribute that resembles <dimension ref="A1:AJ45000" />, we use it to find the starting row and column **as well as the ending row and column**. If this range does not contain all relevant rows or columns, those won't be loaded.

As mentioned, this can be resolved by re-saving the file in Excel or changing the tool to generate either a proper ending point or just the starting cell.

Sluggish Data Loading from Excel

One common cause of slow data loading in Excel is caused by a similar issue. Instead of not using all the data, the ending point indicates significantly more data than is actually there.

To fix this, you can refer to [Locate and reset the last cell on a worksheet](#) for detailed instructions.

XML

2 minutes to read • [Edit Online](#)

Summary

Release State: General Availability Products: Power BI Desktop, Power BI Service (Enterprise Gateway), Dataflows in PowerBI.com (Enterprise Gateway), Dataflows in PowerApps.com (Enterprise Gateway), Excel

Function Reference Documentation: [Xml.Tables](#), [Xml.Document](#)

Capabilities supported

- Import

Load from XML

Load from file

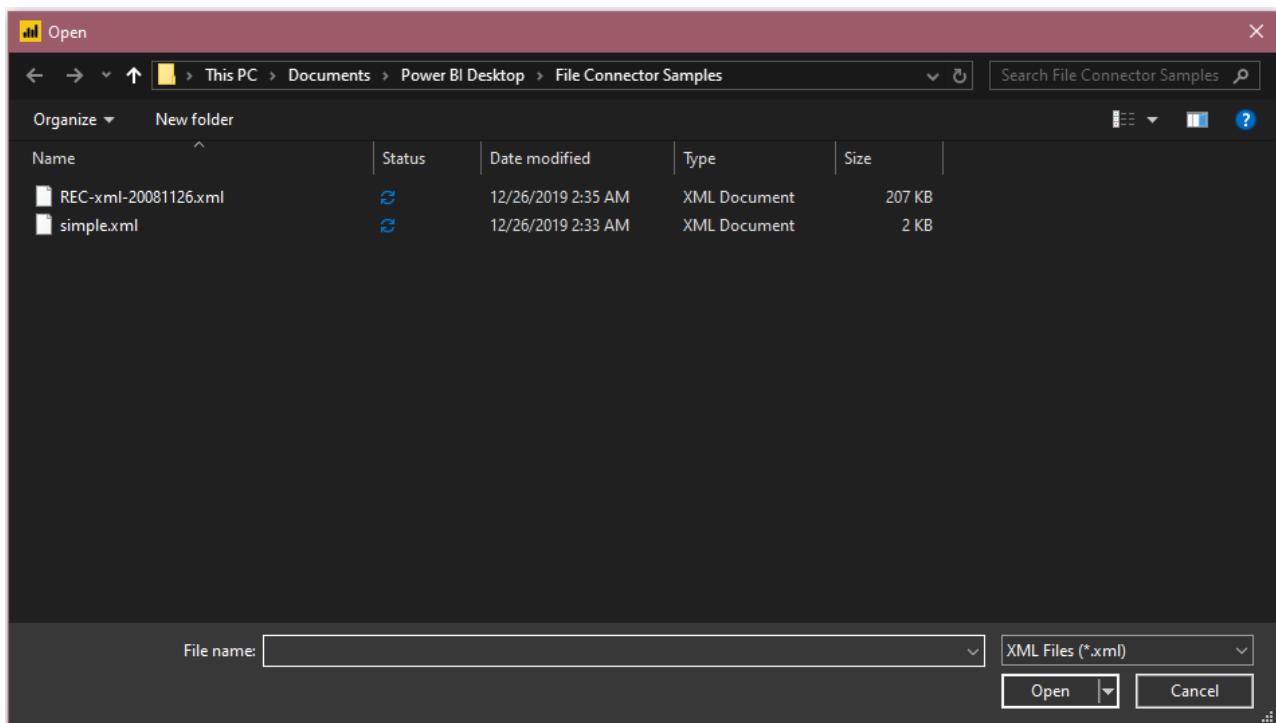
To load a local XML file, all you need to do is select the **XML** option in the connector selection. This will launch a local file browser and allow you to select your XML file.

The screenshot shows the Power BI Navigator window. On the left, there's a sidebar with a search bar, 'Display Options' dropdown, and a tree view showing a folder 'simple.xml [1]' containing a table 'food'. On the right, the 'food' table is displayed in a grid format with columns 'name', 'price', and 'description'. The table data includes:

name	price	description
Belgian Waffles	5.95	Two of our famous Belgian Waffles with plenty of ri
Strawberry Belgian Waffles	7.95	Light Belgian waffles covered with strawberries an
Berry-Berry Belgian Waffles	8.95	Light Belgian waffles covered with an assortment c
French Toast	4.5	Thick slices made from our homemade sourdough
Homestyle Breakfast	6.95	Two eggs, bacon or sausage, toast, and our ever-pi

At the bottom of the window are buttons for 'Load', 'Transform Data', and 'Cancel'.

You'll be presented with the table that the connector loads, which you can then Load or Transform.



Load from web

If you want to load an XML file from the web, instead of selecting the XML connector you can select the Web connector. Paste in the address of the desired file and you'll be prompted with an authentication selection, since you're accessing a website instead of a static file. If there's no authentication, you can just select **Anonymous**. As in the local case, you'll then be presented with the table that the connector loads by default, which you can Load or Transform.

Troubleshooting

Data Structure

Due to the fact that many XML documents have ragged or nested data, you may have to do extra data shaping to get it in the sort of form that will make it convenient to do analytics. This holds true whether you use the UI accessible Xml.Tables function, or the Xml.Document function. Depending on your needs, you may find you have to do more or less data shaping.

Text versus nodes

If your document contains a mixture of text and non-text sibling nodes, you may encounter issues.

For example if you have a node like this:

```
<abc>
    Hello <i>world</i>
</abc>
```

Xml.Tables will return the "world" portion but ignore "Hello". Only the element(s) are returned, not the text. However, Xml.Document will return "Hello <i>world</i>". The entire inner node is turned to text, and structure is not preserved.

PostgreSQL

2 minutes to read • [Edit Online](#)

Summary

Release State: General Availability

Products: Power BI Desktop, Power BI Service (Enterprise Gateway), Dataflows in PowerBI.com (Enterprise Gateway), Dataflows in PowerApps.com (Enterprise Gateway), Excel

Authentication Types Supported: Database (Username/Password)

Function Reference Documentation: [PostgreSQL.Database](#)

NOTE

Some capabilities may be present in one product but not others due to deployment schedules and host-specific capabilities.

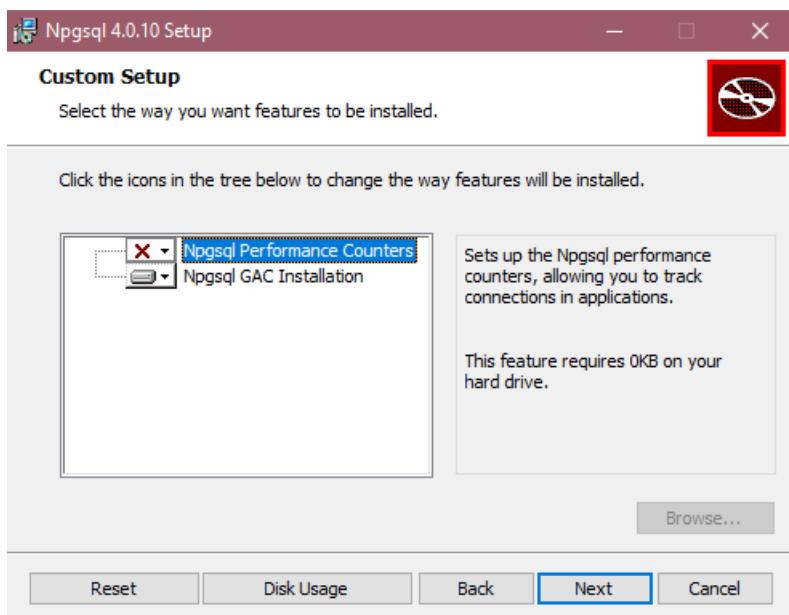
Prerequisites

As of the December Power BI Desktop release, Npgsql 4.0.10 shipped with Power BI Desktop and no additional installation is required. GAC Installation overrides the version provided with Power BI Desktop, which will be the default. Refreshing is supported both through the cloud in the Power BI Service as well as on premise through the Gateway. In the Power BI service, Npgsql 4.0.10 will be used, while on premise refresh will use the local installation of Npgsql, if available, and otherwise use Npgsql 4.0.10.

In order to connect to a PostgreSQL database with **Power BI Desktop**, the Npgsql provider must be installed on the computer running Power BI Desktop.

We recommend Npgsql 4.0.10. Npgsql 4.1 and up will not work due to .NET version incompatibilities.

To install the Npgsql provider, go to the [releases page](#) and download the relevant release. The provider architecture (32-bit or 64-bit) needs to match the architecture of the product where you intent to use the connector. When installing, make sure that you select Npgsql GAC Installation to ensure Npgsql itself is added to your machine.



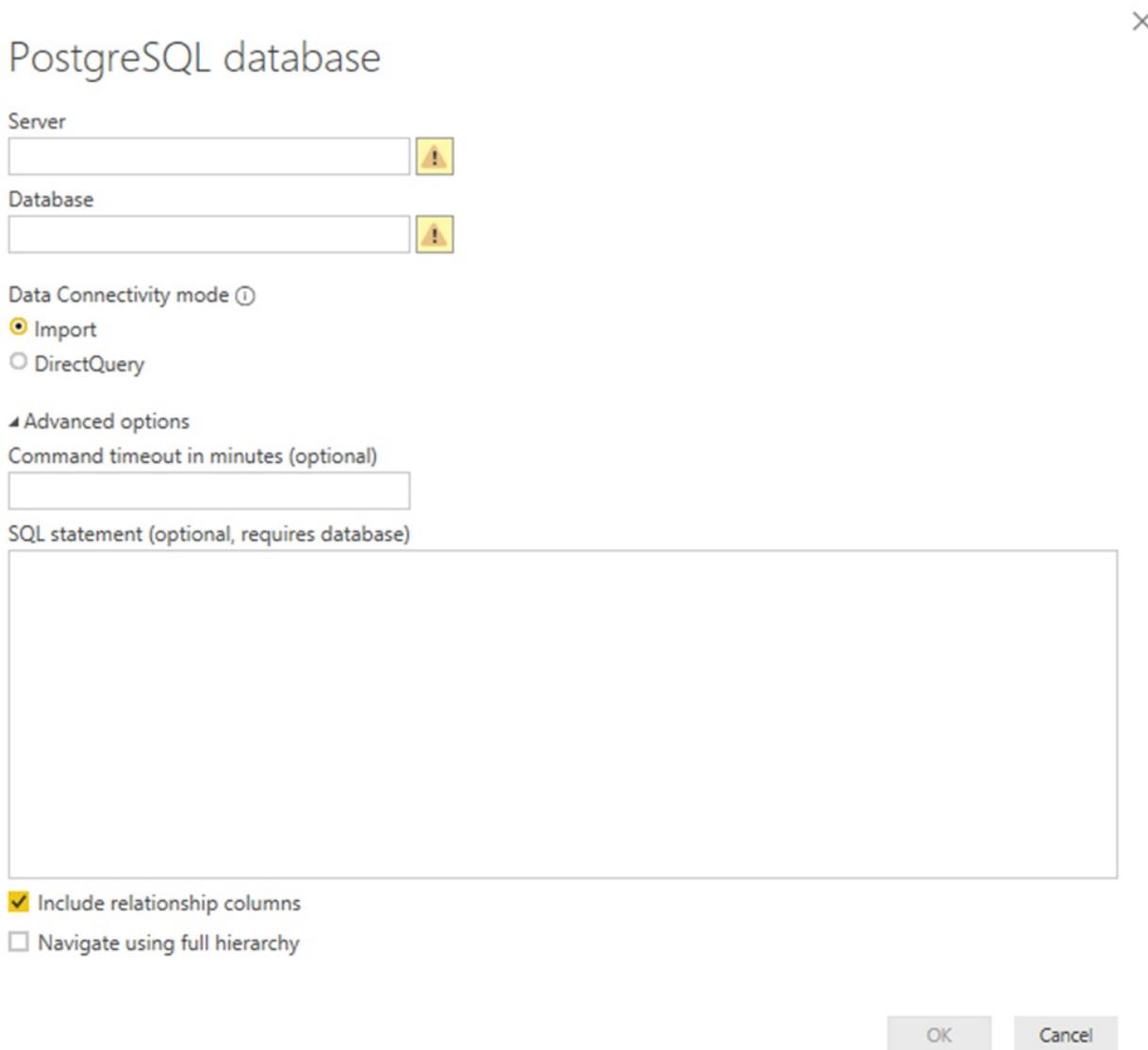
Capabilities Supported

- Import
- DirectQuery (Power BI only, learn more)
- Advanced options
 - Command timeout in minutes
 - Native SQL statement
 - Relationship columns
 - Navigate using full hierarchy

Connect to a PostgreSQL database

Once the matching Npgsql provider is installed, you can connect to a PostgreSQL database. To make the connection, take the following steps:

1. From the Power Query **Get Data** dialog (or **Data** tab in the Excel ribbon), select **Database > PostgreSQL**.



2. In the **PostgreSQL** dialog that appears, provide the name of the server and database. Optionally, you may provide a command timeout and a native query (SQL statement), as well as select whether or not you want to include relationship columns and navigate using full hierarchy. Once you're done, select **Connect**.
3. If the PostgreSQL database requires database user credentials, input those credentials in the dialogue when prompted.

Native Query Folding

To enable Native Query Folding, set the `EnableFolding` flag to `true` for `Value.NativeQuery()` in the advanced editor.

Sample: `Value.NativeQuery(target as any, query, null, [EnableFolding=true])`

Operations that are capable of folding will be applied on top of your native query according to normal Import or Direct Query logic. Native Query folding is not applicable with optional parameters present in `Value.NativeQuery()`.

Troubleshooting

Your native query may throw the following error:

We cannot fold on top of this native query. Please modify the native query or remove the 'EnableFolding' option.

A basic trouble shooting step is to check if the query in `Value.NativeQuery()` throws the same error with a `limit 1` clause around it:

```
select * from (query) _ limit 1
```

SQL Server

2 minutes to read • [Edit Online](#)

Summary

Release State: General Availability

Products: Power BI Desktop, Power BI Service (Enterprise Gateway), Dataflows in PowerBI.com (Enterprise Gateway), Dataflows in PowerApps.com (Enterprise Gateway), Excel, Flow

Authentication Types Supported: Database (Username/Password), Windows

[M Function Reference](#)

NOTE

Some capabilities may be present in one product but not others due to deployment schedules and host-specific capabilities.

Prerequisites

By default, Power BI installs an OLE DB driver for SQL Server. However, for optimal performance, we recommend that the customer installs the [SQL Server Native Client](#) before using the SQL Server connector. SQL Server Native Client 11.0 and SQL Server Native Client 10.0 are both supported in the latest version.

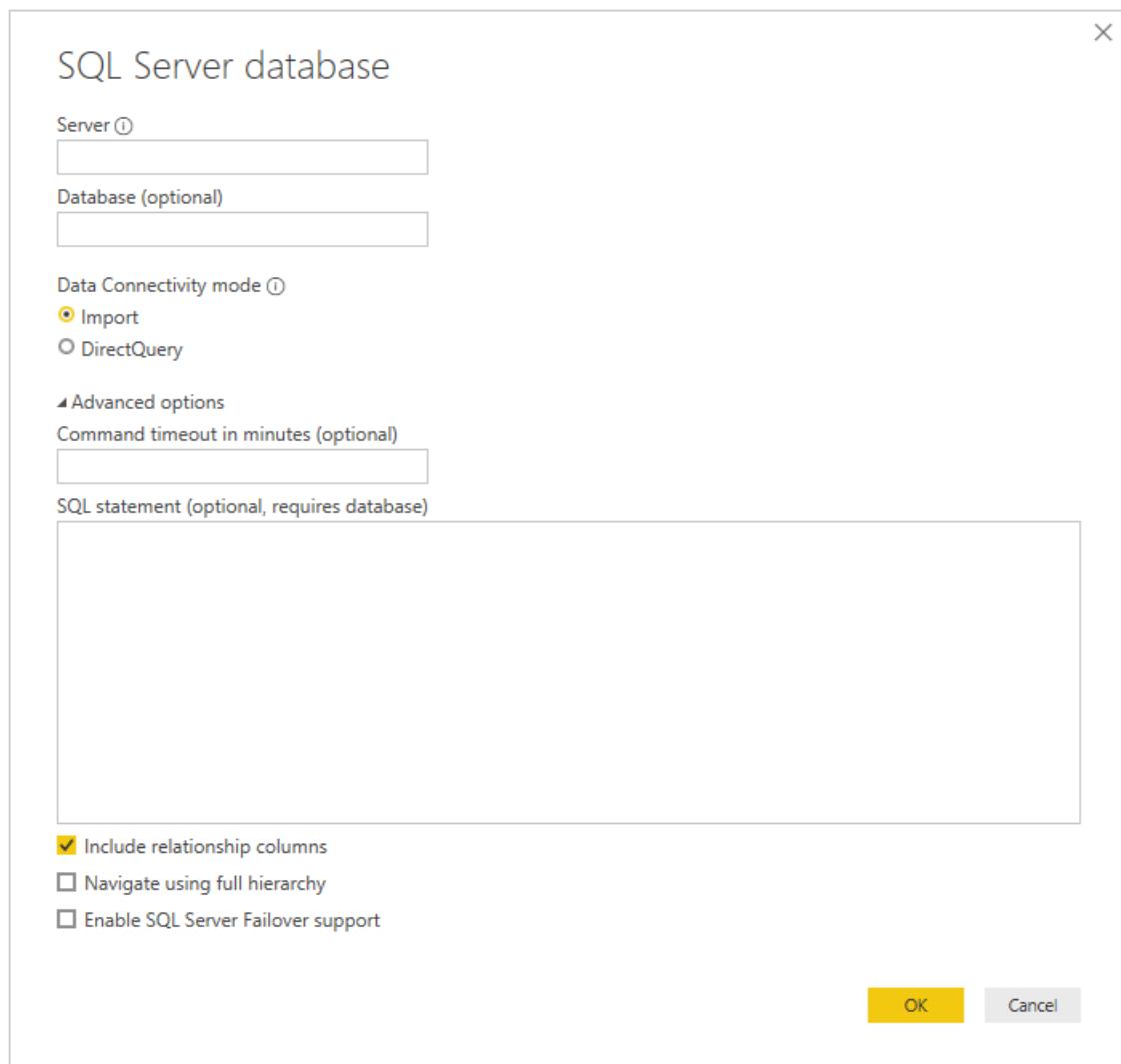
Capabilities Supported

- Import
- DirectQuery (Power BI only, learn more)
- Advanced options
 - Command timeout in minutes
 - Native SQL statement
 - Relationship columns
 - Navigate using full hierarchy
 - SQL Server failover support

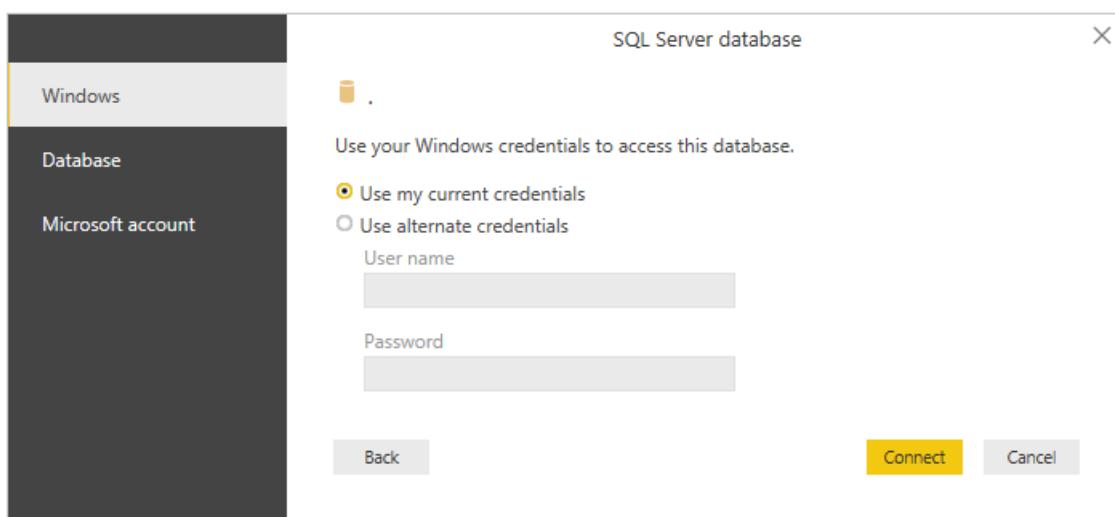
Connect to SQL Server database

To make the connection, take the following steps:

1. From the Power Query **Get Data** dialog (or **Data** tab in the Excel ribbon), select **Database > SQL Server database**.

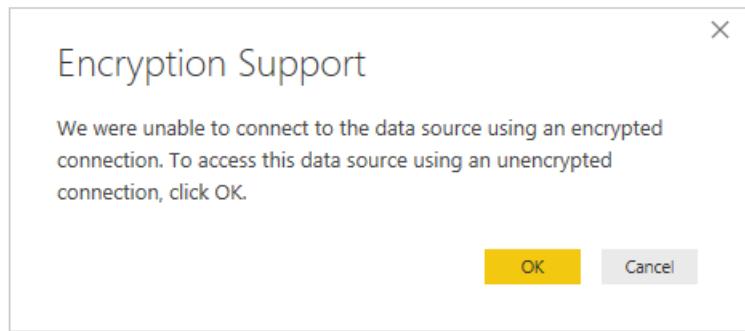


2. In the **SQL Server database** dialog that appears, provide the name of the server and database (optional). Optionally, you may provide a command timeout and a native query (SQL statement), as well as select whether or not you want to include relationship columns and navigate using full hierarchy. Once you're done, select **Connect**.
3. Select the authentication type and input those credentials in the dialogue when prompted.



NOTE

If the connection is not encrypted, you'll be prompted with the following dialog.



Select **OK** to connect to the database by using an unencrypted connection, or follow the [instructions](#) to setup encrypted connections to SQL Server.

Text/CSV

4 minutes to read • [Edit Online](#)

Summary

Release State: General Availability Products: Power BI Desktop, Power BI Service (Enterprise Gateway), Dataflows in PowerBI.com (Enterprise Gateway), Dataflows in PowerApps.com (Enterprise Gateway), Excel

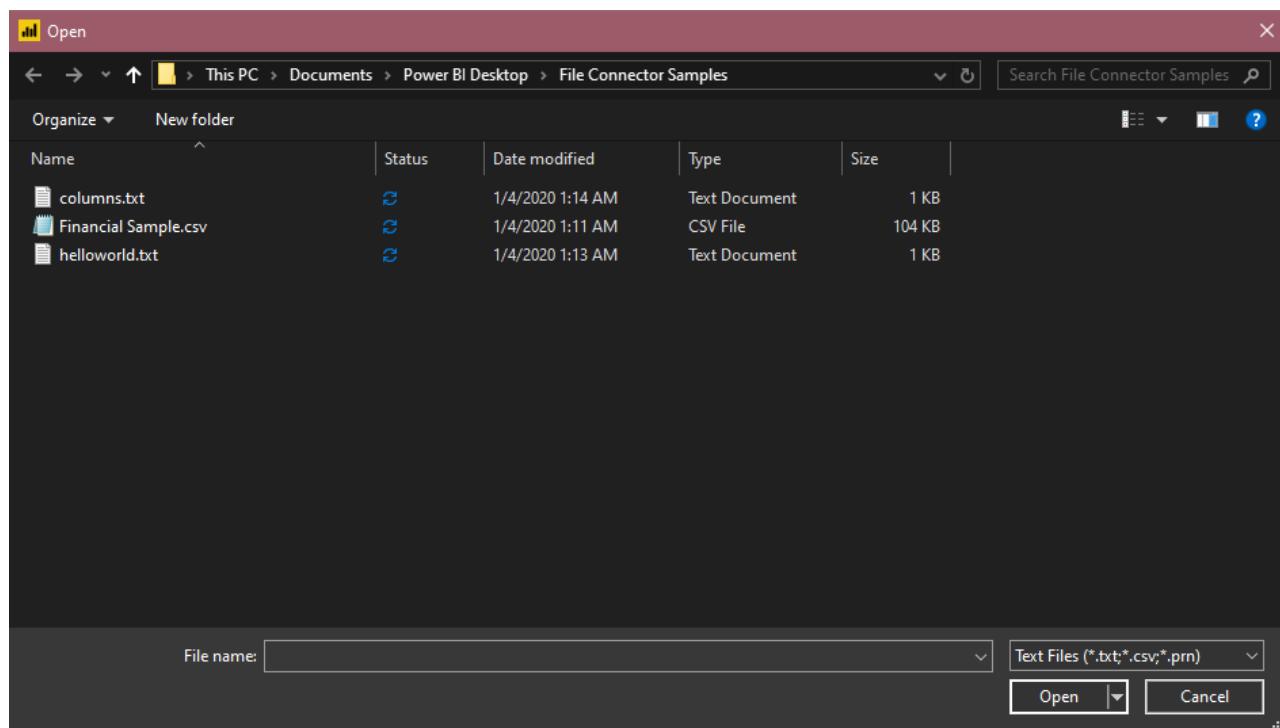
Function Reference Documentation: [File.Contents](#), [Lines.FromBinary](#), [Csv.Document](#)

Capabilities supported

- Import

Load from Text/CSV File

To load a local text or csv file, all you need to do is select the **Text/CSV** option in the connector selection. This will launch a local file browser and allow you to select your text file.



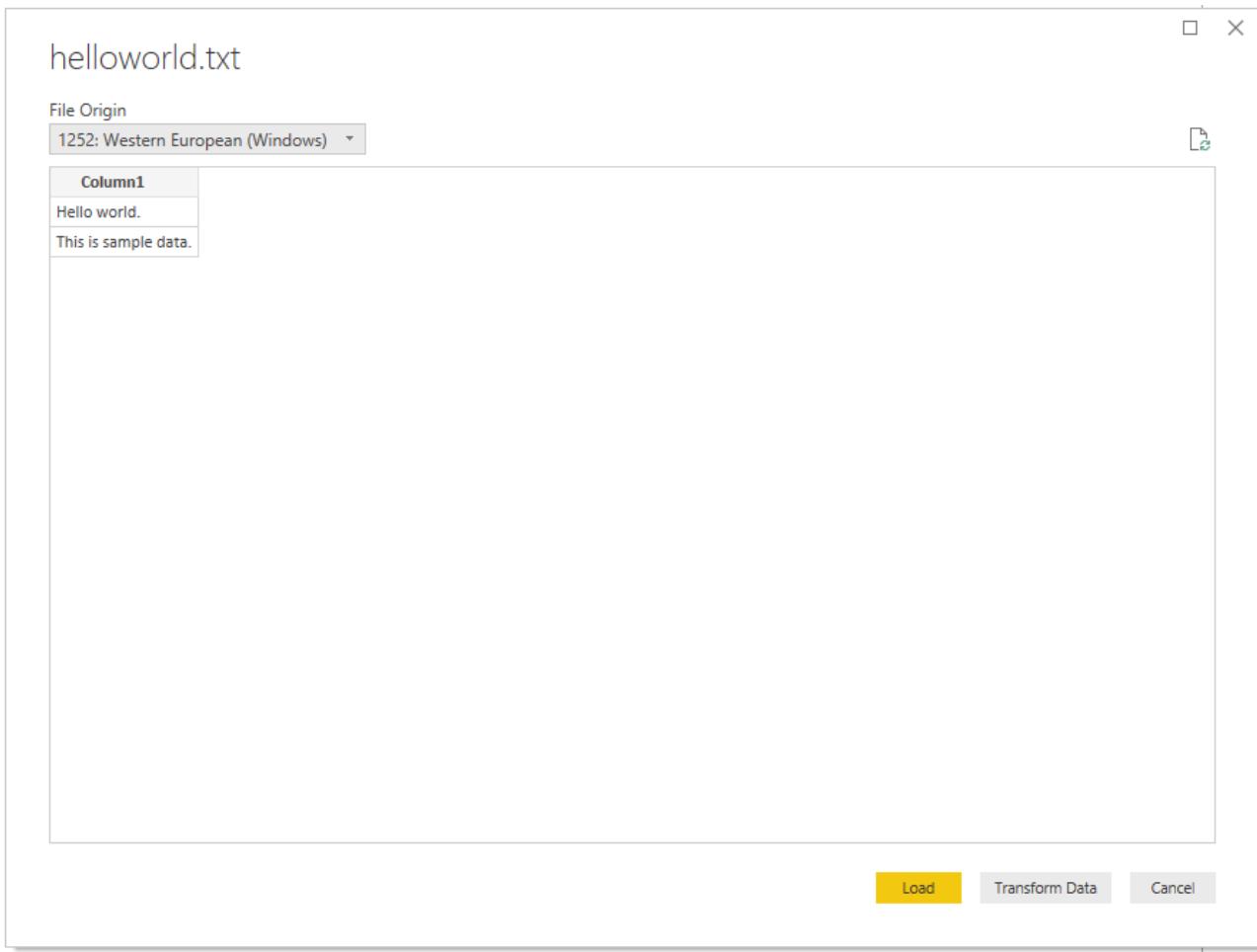
Power Query will treat CSVs as structured files with a comma as a delimiter—a special case of a text file. If you choose a text file, Power Query will automatically attempt to determine if it has delimiter separated values, and what that delimiter is. If it can infer this, it'll automatically treat it as a structured data source.

Unstructured Text

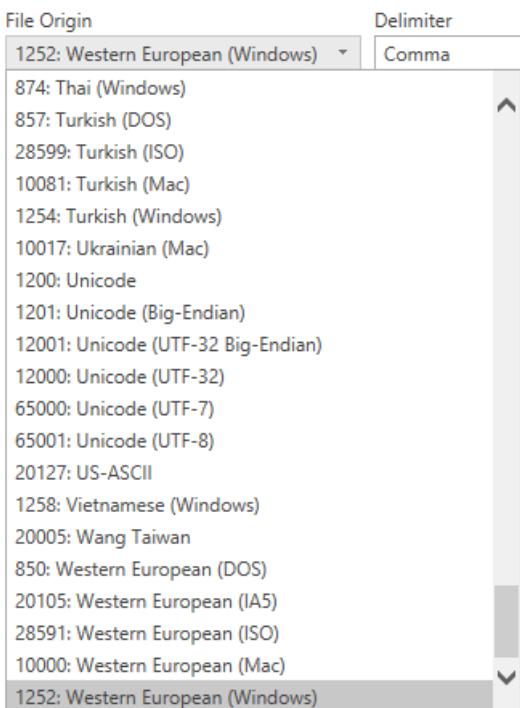
If your text file doesn't have structure you'll get a single column with a new row per line encoded in the source text. As a sample for unstructured text, you can consider a notepad file with the following contents:

```
Hello world.  
This is sample data.
```

When you load it, you're presented with a navigation screen that loads each of these lines into their own row.



There's only one thing you can configure on this dialog, which is the **File Origin** dropdown select. This lets you select [which character set](#) was used to generate the file. Currently, character set is not inferred, and UTF-8 will only be inferred if it starts with a [UTF-8 BOM](#).



CSV

You can find a sample CSV file [here](#).

In addition to file origin, CSV also supports specifying the delimiter, as well as how data type detection will be handled.

Financial Sample.csv

File Origin Delimiter Data Type Detection

1252: Western European (Windows) Comma Based on first 200 rows

Segment	Country	Product	Discount Band	Units Sold	Manufacturing Price	Sale Price	Gross Sales	Discounts
Government	United States of America	Paseo	Low	3450	10	350	1207500	48300
Government	France	Amarilla	None	2750	260	350	962500	0
Government	Germany	Velo	Low	2966	120	350	1038100	20762
Government	Germany	Amarilla	Low	2966	260	350	1038100	20762
Government	Germany	Velo	Low	2877	120	350	1006950	20139
Government	Germany	VTT	Low	2877	250	350	1006950	20139
Government	Canada	Carretera	Low	2852	3	350	998200	19964
Government	Canada	Paseo	Low	2852	10	350	998200	19964
Government	France	Amarilla	Medium	2876	260	350	1006600	70462
Government	France	Carretera	Low	2155	3	350	754250	7542.5
Government	France	Paseo	Low	2155	10	350	754250	7542.5
Government	France	Velo	Low	2177	120	350	761950	30478
Government	France	VTT	Low	2177	250	350	761950	30478
Government	Mexico	VTT	Low	1940	250	350	679000	13580
Government	Canada	Paseo	None	1725	10	350	603750	0
Government	United States of America	VTT	High	2807	250	350	982450	98245
Government	Germany	Amarilla	Low	1907	260	350	667450	26698
Government	France	Velo	Medium	2076	120	350	726600	43596
Government	France	Amarilla	Medium	2076	260	350	726600	43596
Government	Germany	Montana	Low	1797	5	350	628950	18868.5

Load Transform Data Cancel

Delimiters available include colon, comma, equals sign, semicolon, space, tab, a custom delimiter (which can be any string), and a fixed width (splitting up text by some standard number of characters).

Delimiter

- Comma
- Colon
- Comma
- Equals Sign
- Semicolon
- Space
- Tab
- Custom--
- Fixed Width--

The final dropdown allows you to select how you want to handle data type detection. It can be done based on the first 200 rows, on the entire data set, or you can choose to not do automatic data type detection and instead let all columns default to 'Text'. Warning: if you do it on the entire data set it may cause the initial load of the data in the editor to be slower.

Data Type Detection

- Based on first 200 rows
- Based on first 200 rows
- Based on entire dataset
- Do not detect data types

Since inference can be incorrect, it is worth double checking settings before loading.

Structured Text

When Power Query can detect structure to your text file, it'll treat it as a delimiter separated value file, and give you the same options available when opening a CSV—which is essentially just a file with an extension indicating the delimiter type.

For example, if you save the following below as a text file, it'll be read as having a tab delimiter rather than unstructured text.

The screenshot shows the 'Get Data' dialog in Power BI. The file 'columns.txt' is selected. In the 'File Origin' section, the encoding is set to '1252: Western European (Windows)'. The 'Delimiter' dropdown is set to 'Tab', which is highlighted in the dropdown menu. The 'Data Type Detection' dropdown is set to 'Based on first 200 rows'. The preview pane shows the following data:

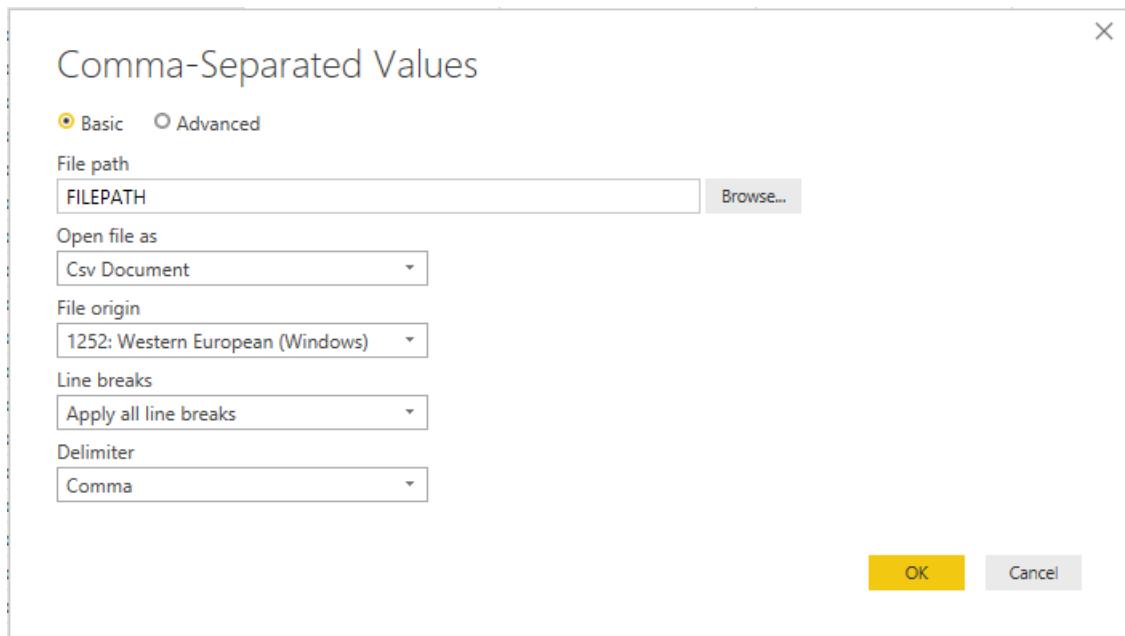
Column 1	Column 2	Column 3
This is a string.	1	ABC123
This is also a string.	2	DEF456

At the bottom right are the 'Load', 'Transform Data', and 'Cancel' buttons.

This can be used for any kind of other delimiter based file.

Editing Source

When editing the source step, you'll be presented with a slightly different dialog than when initially loading. Depending on what you are currently treating the file as (that is, text or csv) you'll be presented with a screen with a variety of dropdowns.



The **Line breaks** dropdown will allow you to select if you want to apply linebreaks that are inside quotes or not.



For example, if you edit the 'structured' sample provided above, you can add a line break.

Column 1	Column 2	Column 3
This is a string. 1 "ABC 123"		
This is also a string. 2 "DEF456"		

If **Line breaks** is set to **Ignore quoted line breaks**, it will load as if there was no line break (with an extra space).

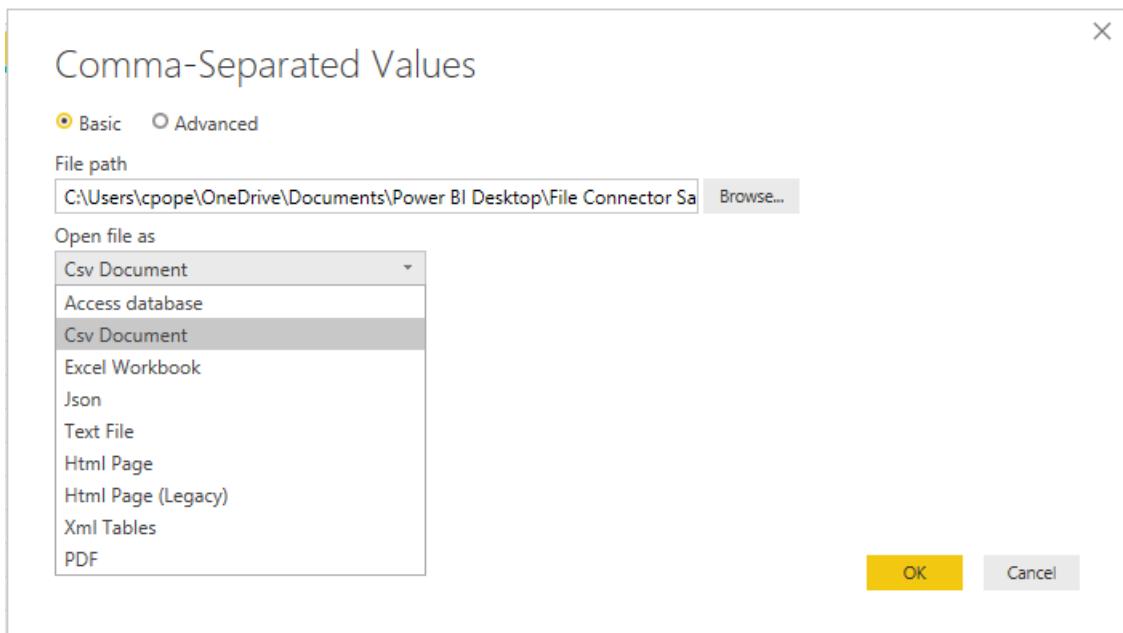
	A ^B _C Column1	A ^B _C Column2	A ^B _C Column3
1	Column 1	Column 2	Column 3
2	This is a string. 123"	1	ABC 123
3	This is also a string.	2	DEF456

If **Line breaks** is set to **Apply all line breaks**, it will load an extra row, with the content after the line breaks being the only content in that row (exact output may depend on structure of the file contents).

	A ^B _C Column1	A ^B _C Column2	A ^B _C Column3
1	Column 1	Column 2	Column 3
2	This is a string.	1	ABC
3	123"		
4	This is also a string.	2	DEF456

The **Open file as** dropdown will let you edit what you want to load the file as—important for troubleshooting.

Note that for structured files that aren't technically CSVs (such as a tab separated value file saved as a text file), you should still have **Open file as** set to CSV. This also determines which dropdowns are available in the rest of the dialog.



Troubleshooting

Loading Files from the Web

If you are requesting text/csv files from the web and also promoting headers, and you're retrieving enough files that you need to be concerned with potential throttling, you should consider wrapping your `Web.Contents` call with `Binary.Buffer()`. In this case, buffering the file before promoting headers will cause the file to only be requested once.

Unstructured text being interpreted as structured

In rare cases, a document that has similar comma numbers across paragraphs might be interpreted to be a CSV. If this happens, edit the **Source** step in the Query Editor, and select **Text** instead of **CSV** in the **Open File As** dropdown select.

ODBC

2 minutes to read • [Edit Online](#)

Summary

Release State: General Availability

Products: Power BI Desktop, Power BI Service (On-Premise Gateway), Dataflows in PowerBI.com (On-Premise Gateway), Dataflows in PowerApps.com (On-Premise Gateway), Excel, Flow

Authentication Types Supported: Database (Username/Password), Windows, Default or Custom

M Function Reference: [Odbc.DataSource](#), [Odbc.Query](#)

NOTE

Some capabilities may be present in one product but not others due to deployment schedules and host-specific capabilities.

Capabilities Supported

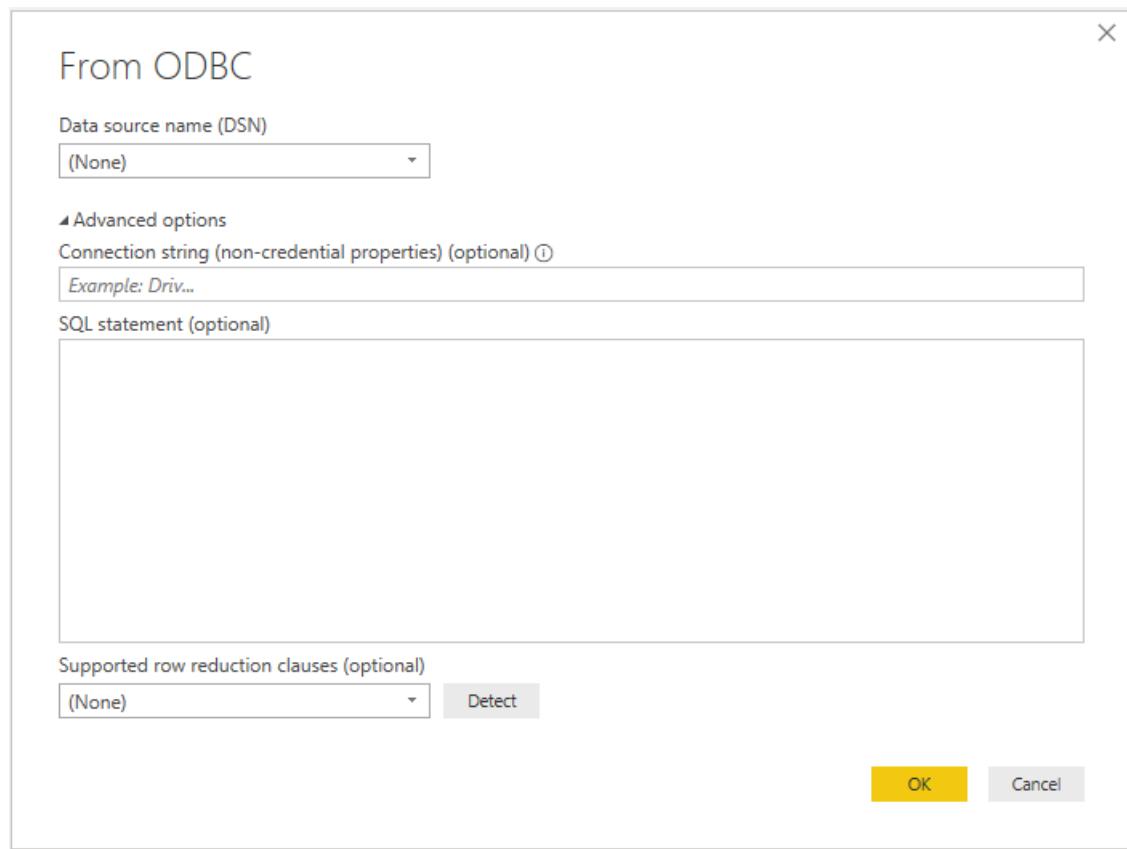
- Import
- Advanced options
 - Connection string (non-credential properties)
 - SQL statement
 - Supported row reduction clauses

Connect to an ODBC data source

Before you get started, make sure you've properly configured the connection in the [Windows ODBC Data Source Administrator](#). The exact process here will depend on the driver.

To make the connection, take the following steps:

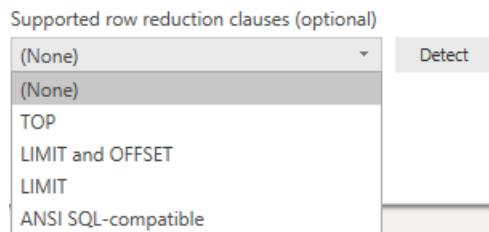
1. From the Power Query **Get Data** dialog (or **Data** tab in the Excel ribbon), select **Database > ODBC**.



2. In the **From ODBC** dialog that appears, provide the connection string (optional).

You may also choose to provide a SQL statement, depending on the capabilities of the driver. Ask your vendor for more information.

3. To enable folding support for `Table.FirstN`, select **Detect** to find supported row reduction clauses, or select from one of the drop down options.



This option is not applicable when using a native SQL statement.

4. Once you're done, select **Connect**. Select the authentication type and input those credentials in the dialogue when prompted.

Analyze data in Azure Data Lake Storage Gen2 by using Power BI

3 minutes to read • [Edit Online](#)

In this article you'll learn how to use Power BI Desktop to analyze and visualize data that is stored in a storage account that has a hierarchical namespace (Azure Data Lake Storage Gen2).

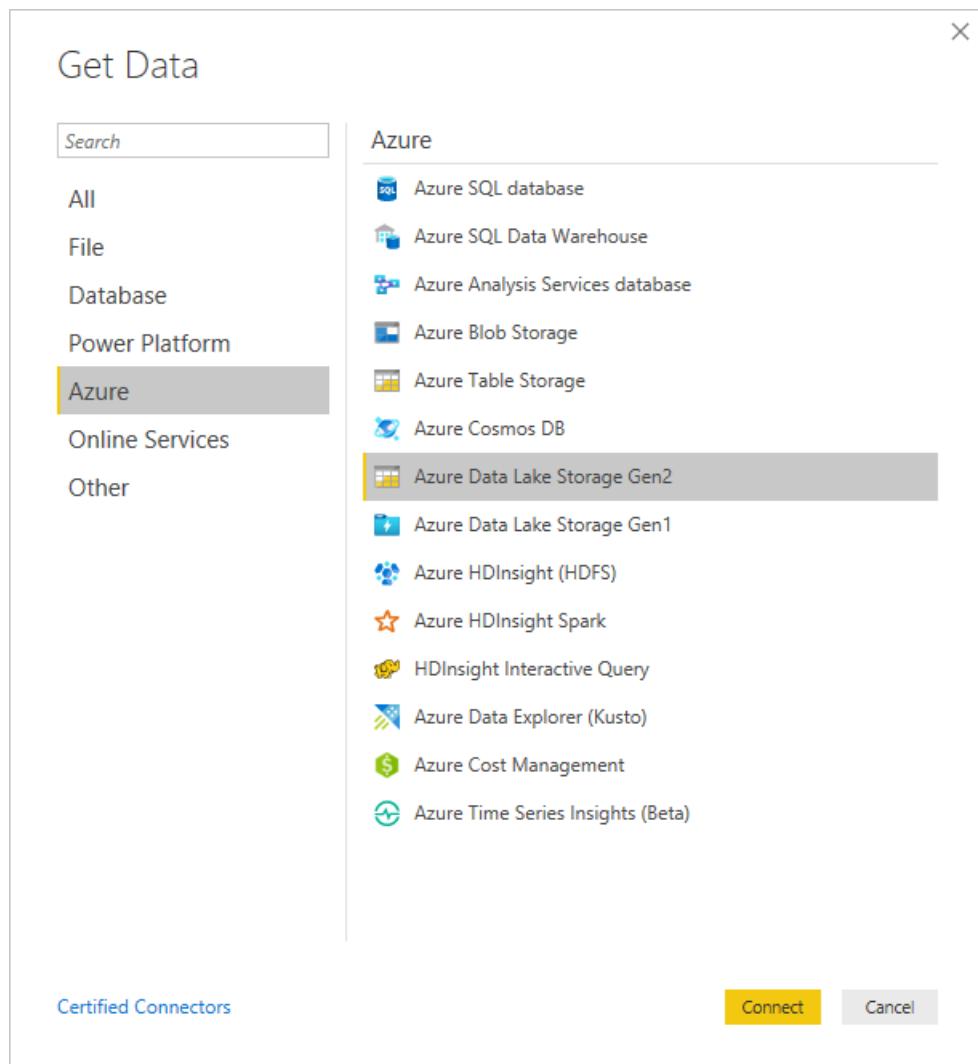
Prerequisites

Before you begin this tutorial, you must have the following:

- An Azure subscription. See [Get Azure free trial](#).
- A storage account that has a hierarchical namespace. Follow [these](#) instructions to create one. This article assumes that you've created a storage account named `myadlsg2`.
- You are granted one of the following roles for the storage account: **Blob Data Reader**, **Blob Data Contributor**, or **Blob Data Owner**.
- A sample data file named `Drivers.txt` located in your storage account. You can download this sample from [Azure Data Lake Git Repository](#), and then upload that file to your storage account.
- **Power BI Desktop**. You can download this from the [Microsoft Download Center](#).

Create a report in Power BI Desktop

1. Launch Power BI Desktop on your computer.
2. From the **Home** tab of the Ribbon, select **Get Data**, and then select **More**.
3. In the **Get Data** dialog box, select **Azure > Azure Data Lake Store Gen2**, and then select **Connect**.



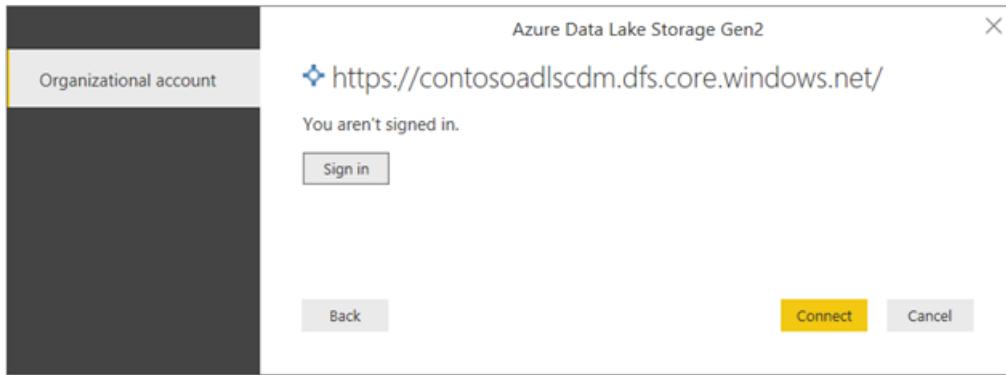
4. In the **Azure Data Lake Storage Gen2** dialog box, you can provide the URL to your Azure Data Lake Storage Gen2 account, filesystem, or subfolder using the container endpoint format. URLs for Data Lake Storage Gen2 have the following pattern:

```
https://<accountname>.dfs.core.windows.net/<filesystemname>/<subfolder>
```

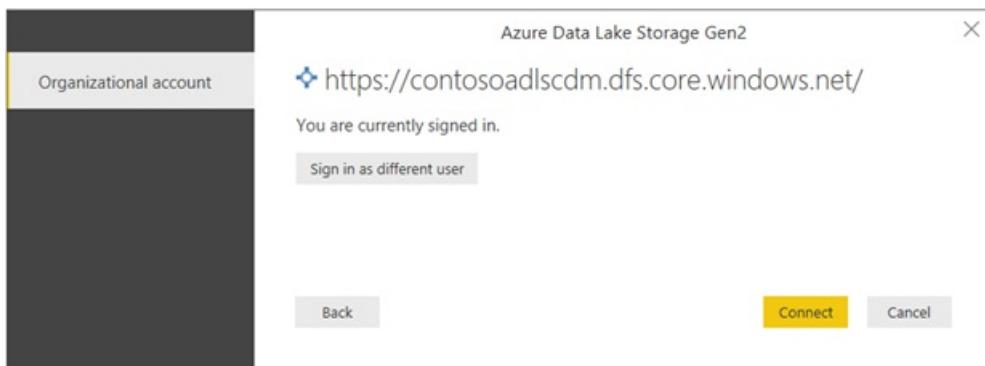
Select **OK** to continue.



5. In the next dialog box, select **Sign in** to sign into your storage account. You'll be redirected to your organization's sign in page. Follow the prompts to sign into the account.



6. After you've successfully signed in, select **Connect**.



7. The next dialog box shows all files under the URL you provided in step 4 above, including the file that you uploaded to your storage account. Verify the information, and then select **Load**.

A screenshot of a Power BI data load dialog box. The URL is "https://contosoадlscdm.dfs.core.windows.net/filesystem1". A table shows one file: "Drivers.txt" (Content: Binary, Name: Drivers.txt, Extension: .txt, Date accessed: null, Date modified: 4/17/2019 4:59:30 PM, Date created: null, Attributes: Record, Folder Path: https://contosoадlscdm.dfs.core.windows.net/filesystem1). At the bottom are "Combine" (highlighted in yellow), "Load", "Edit", and "Cancel" buttons.

8. After the data has been successfully loaded into Power BI, you'll see the following fields in the **Fields** tab.

The screenshot shows the 'FIELDS' pane in Power BI. It displays a tree structure under 'filesystem1'. The expanded items are 'Date accessed', 'Date created', 'Date modified', 'Extension', 'Folder Path', and 'Name'. Each item has a small icon next to it.

However, to visualize and analyze the data, you might prefer the data to be available using the following fields.

0	1	2	3	4	5	6
1	Maria Anders	Obere Str. 57	Berlin	12209	Germany	
2	Ana Trujillo	Avda. de la Constitución, 2222	México D.F.	5021	Mexico	

In the next steps, you'll update the query to convert the imported data to the desired format.

- From the **Home** tab on the ribbon, select **Edit Queries**.

The screenshot shows the 'Queries [1]' pane in Power BI. It displays a table with one row. The columns are 'Content' (containing 'Drivers.txt'), 'Name' (containing '.txt'), 'Extension' (containing '.txt'), and 'Date accessed' (containing a timestamp).

- In the **Query Editor**, under the **Content** column, select **Binary**. The file will automatically be detected as CSV and you should see an output as shown below. Your data is now available in a format that you can use to create visualizations.

The screenshot shows the 'Queries [1]' pane in Power BI. It displays a large table with many rows and columns. The 'Content' column is now displayed as a CSV file. The 'Properties' pane on the right shows the file is now a CSV.

- From the **Home** tab on the ribbon, select **Close & Apply**.

The screenshot shows the Power BI ribbon with the 'File' tab selected. The 'Close & Apply' button is highlighted. The 'Queries [1]' pane shows the transformed data.

12. Once the query is updated, the **Fields** tab will show the new fields available for visualization.

The screenshot shows the 'FIELDS' tab with a search bar at the top. Below it is a tree view of columns under a node named 'filesystem1'. The columns listed are Column1, Column2, Column3, Column4, Column5, Column6, Column7, and Column8. Each column has a small square icon next to its name.

13. Now you can create a pie chart to represent the drivers in each city for a given country. To do so, make the following selections.

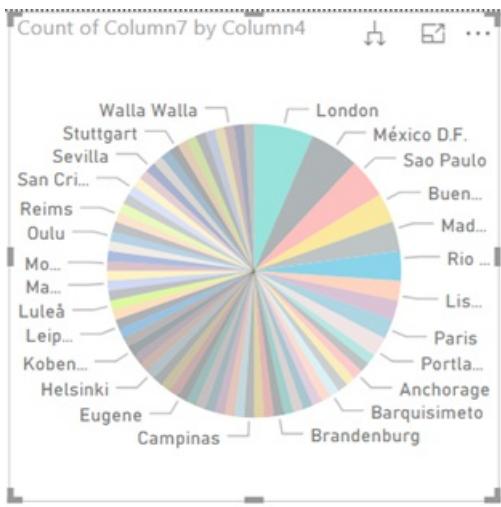
From the **Visualizations** tab, select the symbol for a pie chart.

The screenshot shows the 'Visualizations' tab with a grid of icons representing different types of charts. One specific icon, which looks like a pie chart, is highlighted with a red box.

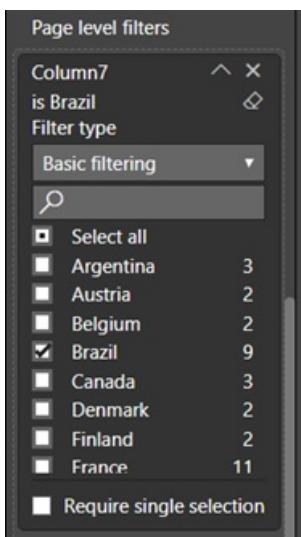
In this example, the columns you're going to use are Column 4 (name of the city) and Column 7 (name of the country). Drag these columns from the **Fields** tab to the **Visualizations** tab as shown below.

This screenshot shows the 'Visualizations' tab interface. On the left, there's a sidebar with options like 'Legend', 'Drag data fields here', 'Details', 'Values', and 'Filters'. In the 'Values' section, there's a dropdown menu with 'Count of Column7' selected. Red arrows numbered 1, 2, and 3 point from the 'Column4' and 'Column7' checkboxes in the 'Fields' tab to the 'Values' dropdown and the filter section respectively. The 'Fields' tab is also visible on the right side of the interface.

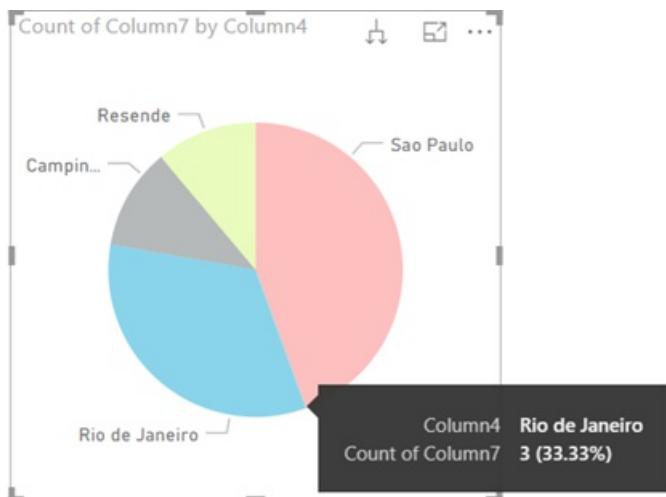
The pie chart should now resemble the one shown below.



14. By selecting a specific country from the page level filters, you can now see the number of drivers in each city of the selected country. For example, under the **Visualizations** tab, under **Page level filters**, select **Brazil**.



15. The pie chart is automatically updated to display the drivers in the cities of Brazil.



16. From the **File** menu, select **Save** to save the visualization as a Power BI Desktop file.

Publish report to Power BI service

After you've created the visualizations in Power BI Desktop, you can share it with others by publishing it to the Power BI service. For instructions on how to do that, see [Publish from Power BI Desktop](#).

Handling Authentication

7 minutes to read • [Edit Online](#)

Authentication Kinds

An extension can support one or more kinds of Authentication. Each authentication kind is a different type of credential. The authentication UI displayed to end users in Power Query is driven by the type of credential(s) that an extension supports.

The list of supported authentication types is defined as part of an extension's [Data Source Kind](#) definition. Each Authentication value is a record with specific fields. The table below lists the expected fields for each kind. All fields are required unless marked otherwise.

AUTHENTICATION KIND	FIELD	DESCRIPTION
Implicit		The Implicit (anonymous) authentication kind does not have any fields.
OAuth	StartLogin	Function that provides the URL and state information for initiating an OAuth flow. See Implementing an OAuth Flow below.
	FinishLogin	Function that extracts the access_token and other properties related to the OAuth flow.
	Refresh	(optional) Function that retrieves a new access token from a refresh token.
	Logout	(optional) Function that invalidates the user's current access token.
	Label	(optional) A text value that allows you to override the default label for this AuthenticationKind.
UsernamePassword	UsernameLabel	(optional) A text value to replace the default label for the <i>Username</i> text box on the credentials UI.
	PasswordLabel	(optional) A text value to replace the default label for the <i>Password</i> text box on the credentials UI.
	Label	(optional) A text value that allows you to override the default label for this AuthenticationKind.

AUTHENTICATION KIND	FIELD	DESCRIPTION
Windows	UsernameLabel	(optional) A text value to replace the default label for the <i>Username</i> text box on the credentials UI.
	PasswordLabel	(optional) A text value to replace the default label for the <i>Password</i> text box on the credentials UI.
	Label	(optional) A text value that allows you to override the default label for this AuthenticationKind.
Key	KeyLabel	(optional) A text value to replace the default label for the <i>API Key</i> text box on the credentials UI.
	Label	(optional) A text value that allows you to override the default label for this AuthenticationKind.

The sample below shows the Authentication record for a connector that supports OAuth, Key, Windows, Basic (Username and Password), and anonymous credentials.

Example:

```
Authentication = [
    OAuth = [
        StartLogin = StartLogin,
        FinishLogin = FinishLogin,
        Refresh = Refresh,
        Logout = Logout
    ],
    Key = [],
    UsernamePassword = [],
    Windows = [],
    Implicit = []
]
```

Accessing the Current Credentials

The current credentials can be retrieved using the `Extension.CurrentCredential()` function.

M data source functions that have been enabled for extensibility will automatically inherit your extension's credential scope. In most cases, you won't need to explicitly access the current credentials, however, there are exceptions, such as:

- Passing in the credential in a custom header or query string parameter (such as when you are using the API Key auth type)
- Setting connection string properties for ODBC or ADO.NET extensions
- Checking custom properties on an OAuth token
- Using the credentials as part of an OAuth v1 flow

The `Extension.CurrentCredential()` function returns a record object. The fields it contains will be authentication type specific. See the table below for details.

FIELD	DESCRIPTION	USED BY
AuthenticationKind	Contains the name of the authentication kind assigned to this credential (UsernamePassword, OAuth, and so on).	All
Username	Username value	UsernamePassword, Windows
Password	Password value. Typically used with UsernamePassword, but it is also set for Key.	Key, UsernamePassword, Windows
access_token	OAuth access token value.	OAuth
Properties	A record containing other custom properties for a given credential. Typically used with OAuth to store additional properties (such as the refresh_token) returned with the access_token during the authentication flow.	OAuth
Key	The API key value. Note, the key value is also available in the Password field as well. By default, the mashup engine will insert this in an Authorization header as if this value were a basic auth password (with no username). If this is not the behavior you want, you must specify the ManualCredentials = true option in the options record.	Key
EncryptConnection	A logical value that determined whether to require an encrypted connection to the data source. This value is available for all Authentication Kinds, but will only be set if EncryptConnection is specified in the Data Source definition.	All

The following is an example of accessing the current credential for an API key and using it to populate a custom header (`x-APIKey`).

Example:

```
MyConnector.Raw = (_url as text) as binary =>
let
    apiKey = Extension.CurrentCredential()[Key],
    headers = [
        #"x-APIKey" = apiKey,
        Accept = "application/vnd.api+json",
        #"Content-Type" = "application/json"
    ],
    request = Web.Contents(_url, [ Headers = headers, ManualCredentials = true ])
in
    request
```

Implementing an OAuth Flow

The OAuth authentication type allows an extension to implement custom logic for their service. To do this, an

extension will provide functions for `StartLogin` (returning the authorization URI to initiate the OAuth flow) and `FinishLogin` (exchanging the authorization code for an access token). Extensions can optionally implement `Refresh` (exchanging a refresh token for a new access token) and `Logout` (expiring the current refresh and access tokens) functions as well.

NOTE

Power Query extensions are evaluated in applications running on client machines. Data Connectors *should not* use confidential secrets in their OAuth flows, as users may inspect the extension or network traffic to learn the secret. See the [Proof Key for Code Exchange by OAuth Public Clients RFC](#) (also known as PKCE) for further details on providing flows that don't rely on shared secrets.

There are two sets of OAuth function signatures; the original signature that contains a minimal number of parameters, and an advanced signature that accepts additional parameters. Most OAuth flows can be implemented using the original signatures. You can also mix and match signature types in your implementation. The function calls are matches based on the number of parameters (and their types) - the parameter names are not taken into consideration.

See the [MyGraph](#) and [Github](#) samples for more details.

Original OAuth Signatures

```
StartLogin = (dataSourcePath, state, display) => ...;  
FinishLogin = (context, callbackUri, state) => ...;  
Refresh = (dataSourcePath, refreshToken) => ...;  
Logout = (accessToken) => ...;
```

Advanced OAuth Signatures

Notes about the advanced signatures:

- All signatures accept a `clientApplication` record value, which is reserved for future use.
- All signatures accept a `dataSourcePath` (also referred to as `resourceUrl` in most samples).
- The `Refresh` function accepts an `oldCredential` parameter, which is the previous `record` returned by your `FinishLogin` function (or previous call to `Refresh`).

```
StartLogin = (clientApplication, dataSourcePath, state, display) => ...;  
FinishLogin = (clientApplication, dataSourcePath, context, callbackUri, state) => ...;  
Refresh = (clientApplication, dataSourcePath, oldCredential) => ...;  
Logout = (clientApplication, dataSourcePath, accessToken) => ...;
```

Data Source Paths

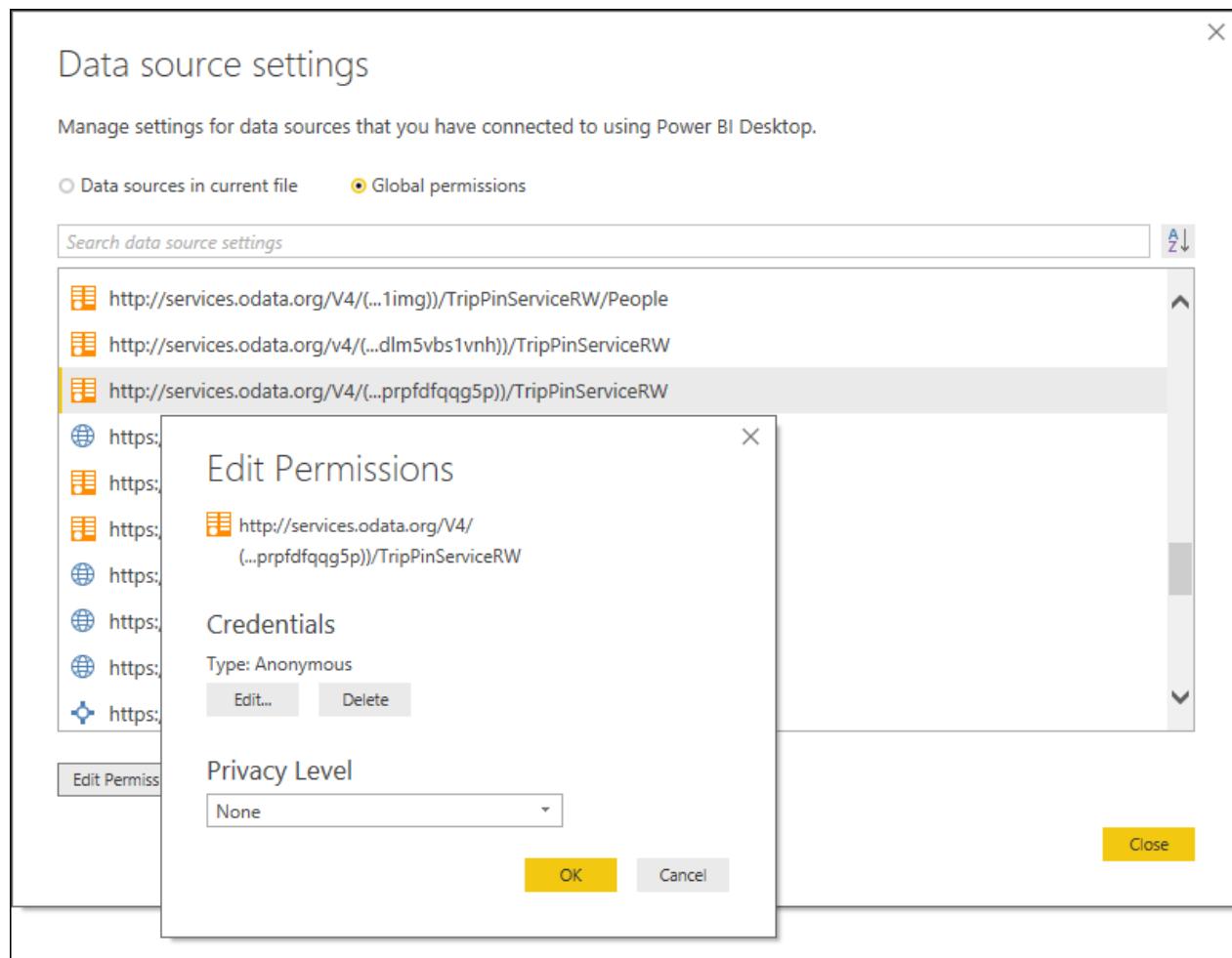
The M engine identifies a data source using a combination of its *Kind* and *Path*. When a data source is encountered during a query evaluation, the M engine will try to find matching credentials. If no credentials are found, the engine returns a special error that results in a credential prompt in Power Query.

The *Kind* value comes from the [Data Source Kind](#) definition.

The *Path* value is derived from the *required parameters* of your [data source function](#). Optional parameters aren't factored into the data source path identifier. As a result, all data source functions associated with a data source kind must have the same parameters. There's special handling for functions that have a single parameter of type

`Uri.Type`. See the [section below](#) for details.

You can see an example of how credentials are stored in the **Data source settings** dialog in Power BI Desktop. In this dialog, the Kind is represented by an icon, and the Path value is displayed as text.



NOTE

If you change your data source function's required parameters during development, previously stored credentials will no longer work (because the path values no longer match). You should delete any stored credentials any time you change your data source function parameters. If incompatible credentials are found, you may receive an error at runtime.

Data Source Path Format

The *Path* value for a data source is derived from the data source function's required parameters.

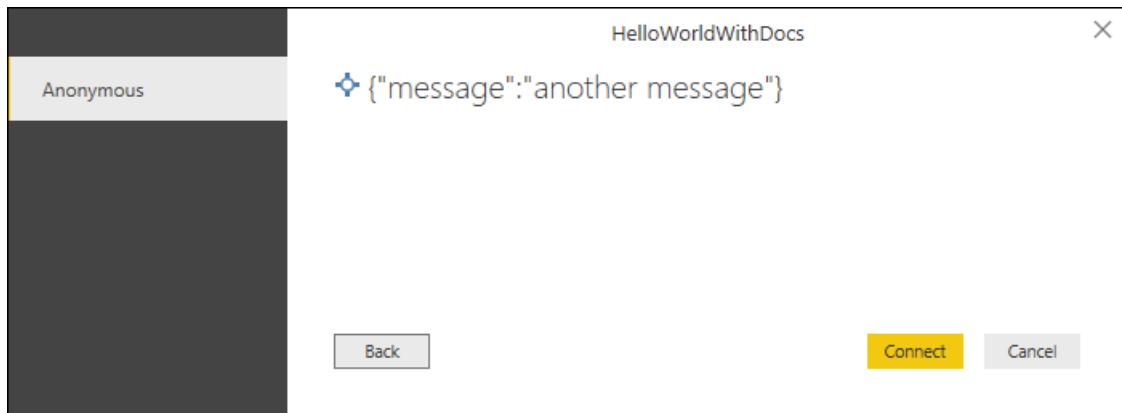
By default, you can see the actual string value in the Data source settings dialog in Power BI Desktop, and in the credential prompt. If the Data Source Kind definition has included a `Label` value, you'll see the label value instead.

For example, the data source function in the [HelloWorldWithDocs sample](#) has the following signature:

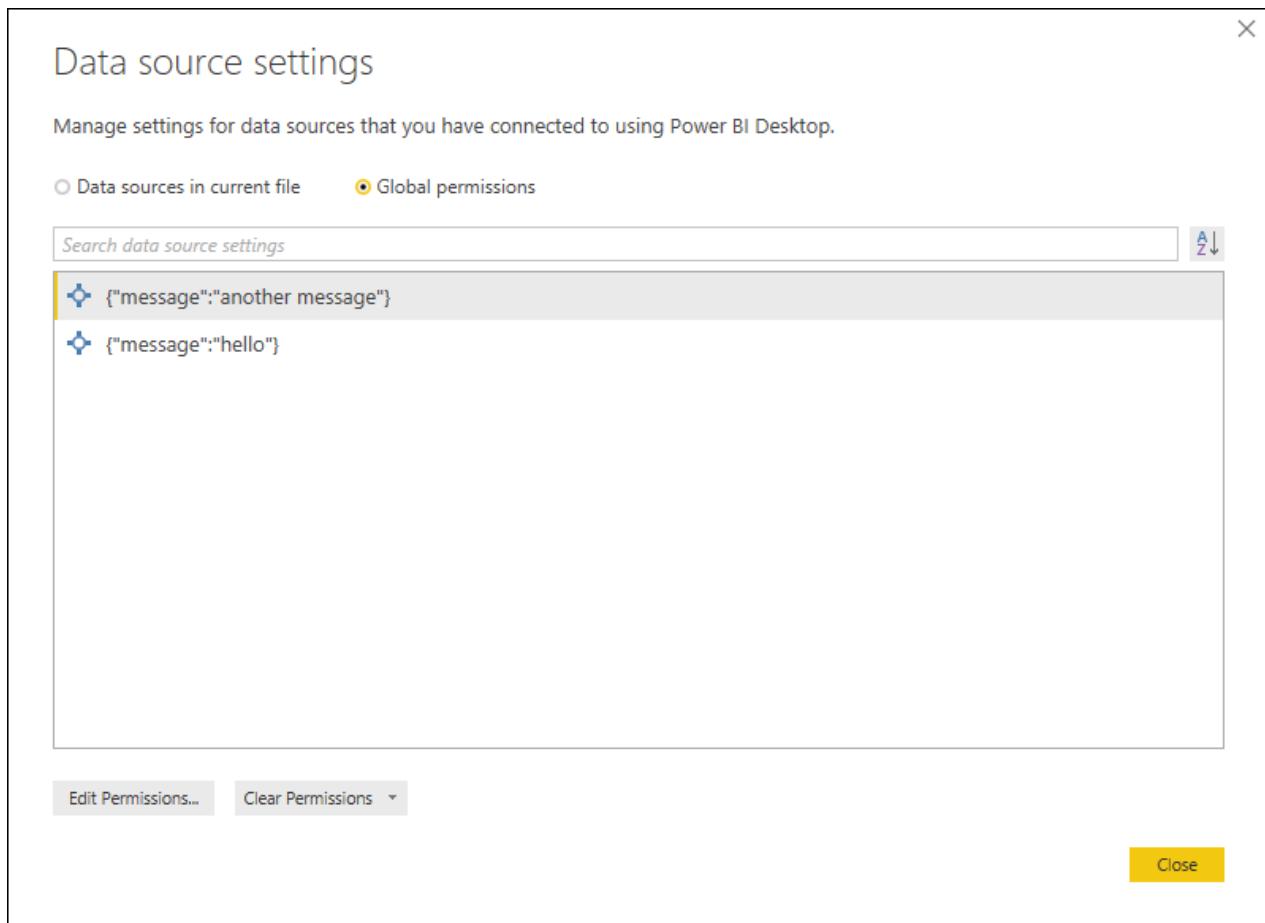
```
HelloWorldWithDocs.Contents = (message as text, optional count as number) as table => ...
```

The function has a single required parameter (`message`) of type `text`, and will be used to calculate the data source path. The optional parameter (`count`) would be ignored. The path would be displayed

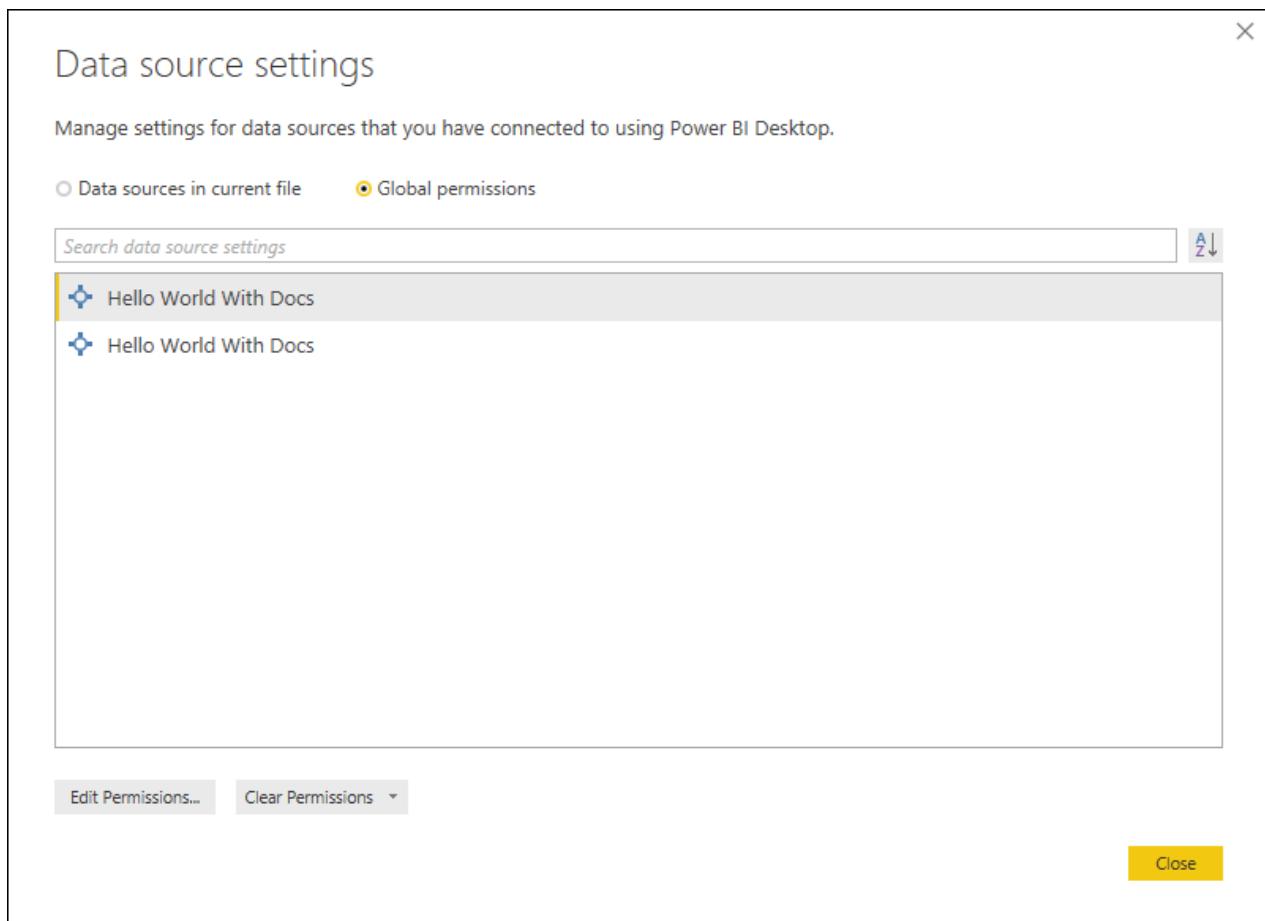
Credential prompt:



Data source settings UI:



When a Label value is defined, the data source path value would not be shown:

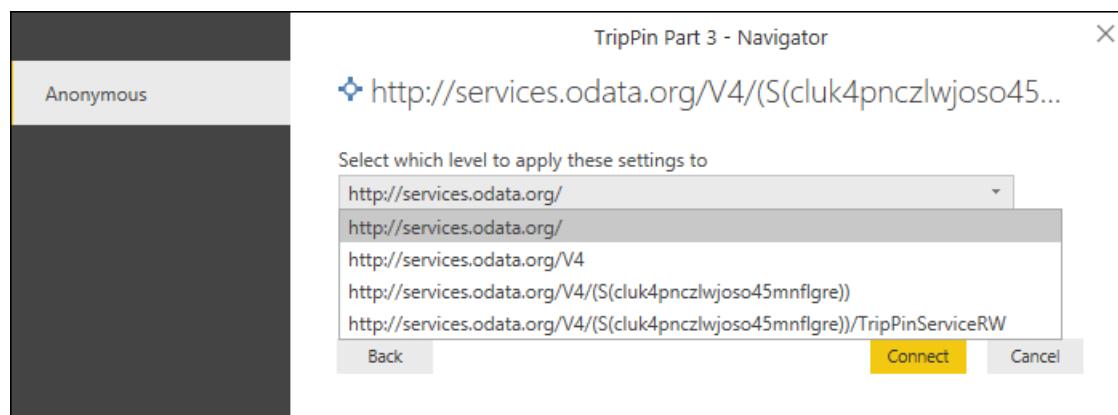


NOTE

We currently recommend you *do not* include a Label for your data source if your function has required parameters, as users won't be able to distinguish between the different credentials they've entered. We are hoping to improve this in the future (that is, allowing data connectors to display their own custom data source paths).

Functions with an Uri parameter

Because data sources with an Uri based identifier are so common, there's special handling in the Power Query UI when dealing with Uri based data source paths. When an Uri-based data source is encountered, the credential dialog provides a drop down allowing the user to select the base path, rather than the full path (and all paths in between).



As `Uri.Type` is an *ascribed type* rather than a *primitive type* in the M language, you'll need to use the `Value.ReplaceType` function to indicate that your text parameter should be treated as an Uri.

```
shared GithubSample.Contents = Value.ReplaceType(Github.Contents, type function (url as Uri.Type) as any);
```

Handling Data Access

3 minutes to read • [Edit Online](#)

Data Source Functions

A Data Connector wraps and customizes the behavior of a [data source function in the M Library](#). For example, an extension for a REST API would make use of the [Web.Contents](#) function to make HTTP requests. Currently, a limited set of data source functions have been enabled to support extensibility.

- [Web.Contents](#)
- [OData.Feed](#)
- [Odbc.DataSource](#)
- [AdoDotNet.DataSource](#)
- [OleDb.DataSource](#)

Example:

```
[DataSource.Kind="HelloWorld", Publish="HelloWorld.Publish"]
shared HelloWorld.Contents = (optional message as text) =>
    let
        message = if (message <> null) then message else "Hello world"
    in
        message;
```

Data Source Kind

Functions marked as `shared` in your extension can be associated with a specific data source by including a `DataSource.Kind` metadata record on the function with the name of a Data Source definition record. The Data Source record defines the authentication types supported by your data source, and basic branding information (like the display name / label). The name of the record becomes is unique identifier.

Functions associated with a data source must have the same required function parameters (including name, type, and order). Functions for a specific Data Source Kind can only use credentials associated with that Kind. Credentials are identified at runtime by performing a lookup based on the combination of the function's required parameters. For more information about how credentials are identified, see [Data Source Paths](#).

Example:

```
HelloWorld = [
    Authentication = [
        Implicit = []
    ],
    Label = Extension.LoadString("DataSourceLabel")
];
```

Properties

The following table lists the fields for your Data Source definition record.

FIELD	TYPE	DETAILS
Authentication	record	Specifies one or more types of authentication supported by your data source. At least one kind is required. Each kind will be displayed as an option in the Power Query credential prompt. For more information, see Authentication Kinds .
Label	text	(optional) Friendly display name for this extension in credential dialogs.
SupportsEncryption	logical	(optional) When true, the UI will present the option to connect to the data source using an encrypted connection. This is typically used for data sources with a non-encrypted fallback mechanism (generally ODBC or ADO.NET based sources).

Publish to UI

Similar to the (Data Source)[#data-source-kind] definition record, the Publish record provides the Power Query UI the information it needs to expose this extension in the **Get Data** dialog.

Example:

```
HelloWorld.Publish = [
    Beta = true,
    ButtonText = { Extension.LoadString("FormulaTitle"), Extension.LoadString("FormulaHelp") },
    SourceImage = HelloWorld.Icons,
    SourceTypeImage = HelloWorld.Icons
];

HelloWorld.Icons = [
    Icon16 = { Extension.Contents("HelloWorld16.png"), Extension.Contents("HelloWorld20.png"),
    Extension.Contents("HelloWorld24.png"), Extension.Contents("HelloWorld32.png") },
    Icon32 = { Extension.Contents("HelloWorld32.png"), Extension.Contents("HelloWorld40.png"),
    Extension.Contents("HelloWorld48.png"), Extension.Contents("HelloWorld64.png") }
];
```

Properties

The following table lists the fields for your Publish record.

FIELD	TYPE	DETAILS
ButtonText	list	List of text items that will be displayed next to the data source's icon in the Power BI Get Data dialog.

FIELD	TYPE	DETAILS
Category	text	Where the extension should be displayed in the Get Data dialog. Currently the only category values with special handling are <code>Azure</code> and <code>Database</code> . All other values will end up under the Other category.
Beta	logical	(optional) When set to true, the UI will display a Preview/Beta identifier next to your connector name and a warning dialog that the implementation of the connector is subject to breaking changes.
LearnMoreUrl	text	(optional) Url to website containing more information about this data source or connector.
SupportsDirectQuery	logical	(optional) Enables Direct Query for your extension. This is currently only supported for ODBC extensions.
SourceImage	record	(optional) A record containing a list of binary images (sourced from the extension file using the Extension.Contents method). The record contains two fields (<code>Icon16</code> , <code>Icon32</code>), each with its own list. Each icon should be a different size.
SourceTypeImage	record	(optional) Similar to SourceImage, except the convention for many out of the box connectors is to display a sheet icon with the source specific icon in the bottom right corner. Having a different set of icons for SourceTypeImage is optional—many extensions simply reuse the same set of icons for both fields.

Enabling Direct Query for an ODBC based connector

21 minutes to read • [Edit Online](#)

Overview

Using M's built-in [Odbc.DataSource](#) function is the recommended way to create custom connectors for data sources that have an existing ODBC driver and/or support a SQL query syntax. Wrapping the [Odbc.DataSource](#) function will allow your connector to inherit default query folding behavior based on the capabilities reported by your driver. This will enable the M engine to generate SQL statements based on filters and other transformations defined by the user within the Power Query experience, without having to provide this logic within the connector itself.

ODBC extensions can optionally enable Direct Query mode, allowing Power BI to dynamically generate queries at runtime without pre-caching the user's data model.

NOTE

Enabling Direct Query support raises the difficulty and complexity level of your connector. When Direct Query is enabled, Power BI will prevent the M engine from compensating for operations that cannot be fully pushed to the underlying data source.

This section builds on the concepts presented in the M Extensibility Reference, and assumes familiarity with the creation of a basic Data Connector.

Refer to the [SqlODBC sample](#) for most of the code examples in the sections below. Additional samples can be found in the ODBC samples directory.

ODBC Extensibility Functions

The M engine provides two ODBC related data source functions: [Odbc.DataSource](#), and [Odbc.Query](#).

The [Odbc.DataSource](#) function provides a default navigation table with all databases, tables, and views from your system, supports query folding, and allows for a range of customization options. The majority of ODBC based extensions will use this as their primary extensibility function. The function accepts two arguments—a connection string, and an options record to provide behavior overrides.

The [Odbc.Query](#) function allows you to execute SQL statements through an ODBC driver. It acts as a passthrough for query execution. Unlike the [Odbc.DataSource](#) function, it doesn't provide query folding functionality, and requires that SQL queries be provided by the connector (or end user). When building a custom connector, this function is typically used internally to run queries to retrieve metadata that might not be exposed through regular ODBC channels. The function accepts two arguments—a connection string, and a SQL query.

Parameters for your Data Source Function

Custom connectors can accept any number of function arguments, but to remain consistent with the built-in data source functions shipped with Power Query, the following guidelines are recommended:

- Require the minimal set of parameters used to establish a connection to your server. The less parameters end users need to provide, the easier your connector will be to use.
- Although you can define parameters with a fixed number of values (that is, a dropdown list in the UI), parameters are entered before the user is authenticated. Any values that can be discovered

programmatically after the user is authenticated (such as catalog or database name) should be selectable through the Navigator. The default behavior for the [Odbc.DataSource](#) function will be to return a hierarchical navigation table consisting of Catalog (Database), Schema, and Table names, although this can be overridden within your connector.

- If you feel your users will typically know what values to enter for items they would select from the Navigator (such as the database name), make these parameters optional. Parameters that can be discovered programmatically should not be made required.
- The last parameter for your function should be an optional record called "options". This parameter typically allows advanced users to set common ODBC related properties (such as CommandTimeout), set behavior overrides specific to your connector, and allows for future extensibility without impacting backwards compatibility for your function.
- Security/credential related arguments MUST never be part of your data source function parameters, as values entered in the connect dialog will be persisted to the user's query. Credential related parameters should be specified as part of the connector's supported Authentication methods.

By default, all required parameters for your data source function are factored into the Data Source Path value used to identify user credentials.

Note that while the UI for the built-in [Odbc.DataSource](#) function provides a dropdown that allows the user to select a DSN, this functionality is not available through extensibility. If your data source configuration is complex enough to require a fully customizable configuration dialog, it's recommended you require your end users to pre-configure a system DSN, and have your function take in the DSN name as a text field.

Parameters for Odbc.DataSource

The [Odbc.DataSource](#) function takes two parameters—a connectionString for your driver, and an options record that lets you override various driver behaviors. Through the options record you can override capabilities and other information reported by the driver, control the navigator behavior, and affect the SQL queries generated by the M engine.

The supported options records fields fall into two categories—those that are public / always available, and those that are only available in an extensibility context.

The following table describes the public fields in the options record.

FIELD	DESCRIPTION
CommandTimeout	A duration value that controls how long the server-side query is allowed to run before it's cancelled. Default: 10 minutes
ConnectionTimeout	A duration value that controls how long to wait before abandoning an attempt to make a connection to the server. Default: 15 seconds

FIELD	DESCRIPTION
CreateNavigationProperties	<p>A logical value that sets whether to generate navigation properties on the returned tables. Navigation properties are based on foreign key relationships reported by the driver, and show up as "virtual" columns that can be expanded in the query editor, creating the appropriate join.</p> <p>If calculating foreign key dependencies is an expensive operation for your driver, you may want to set this value to false.</p> <p>Default: true</p>
HierarchicalNavigation	<p>A logical value that sets whether to view the tables grouped by their schema names. When set to false, tables will be displayed in a flat list under each database.</p> <p>Default: false</p>
SqlCompatibleWindowsAuth	<p>A logical value that determines whether to produce a SQL Server compatible connection string when using Windows Authentication—Trusted_Connection=Yes.</p> <p>If your driver supports Windows Authentication, but requires additional or alternative settings in your connection string, you should set this value to false and use the CredentialConnectionString option record field described below.</p> <p>Default: true</p>

The following table describes the options record fields that are only available through extensibility. Fields that aren't simple literal values are described in subsequent sections.

FIELD	DESCRIPTION
AstVisitor	<p>A record containing one or more overrides to control SQL query generation. The most common usage of this field is to provide logic to generate a LIMIT/OFFSET clause for drivers that don't support TOP.</p> <p>Fields include:</p> <ul style="list-style-type: none"> • Constant • LimitClause <p>See the AstVisitor section for more information.</p>
ClientConnectionPooling	<p>A logical value that enables client-side connection pooling for the ODBC driver. Most drivers will want to set this value to true.</p> <p>Default: false</p>

FIELD	DESCRIPTION
CredentialConnectionString	<p>A text or record value used to specify credential related connection string properties.</p> <p>See the Credential section for more information.</p>
HideNativeQuery	<p>A logical value that controls whether your connector allows native SQL statements to be passed in by a query using the Value.NativeQuery() function.</p> <p>Note: this functionality is currently not exposed in the Power Query user experience. Users would need to manually edit their queries to take advantage of this capability.</p> <p>Default: false</p>
ImplicitTypeConversions	<p>A table value containing implicit type conversions supported by your driver or backend server. Values in this table are additive to the conversions reported by the driver itself.</p> <p>This field is typically used in conjunction with the SQLGetTypeInfo field when overriding data type information reported by the driver.</p> <p>See the ImplicitTypeConversions section for more information.</p>
OnError	<p>An error handling function that receives an errorRecord parameter of type record.</p> <p>Common uses of this function include handling SSL connection failures, providing a download link if your driver isn't found on the system, and reporting authentication errors.</p> <p>See the OnError section for more information.</p>
SoftNumbers	<p>Allows the M engine to select a compatible data type when conversion between two specific numeric types isn't declared as supported in the SQL_CONVERT_* capabilities.</p> <p>Default: false</p>
SqlCapabilities	<p>A record providing various overrides of driver capabilities, and a way to specify capabilities that aren't expressed through ODBC 3.8.</p> <p>See the SqlCapabilities section for more information.</p>
SQLColumns	<p>A function that allows you to modify column metadata returned by the SQLColumns function.</p> <p>See the SQLColumns section for more information.</p>

FIELD	DESCRIPTION
SQLGetFunctions	<p>A record that allows you to override values returned by calls to SQLGetFunctions.</p> <p>A common use of this field is to disable the use of parameter binding, or to specify that generated queries should use CAST rather than CONVERT.</p> <p>See the SQLGetFunctions section for more information.</p>
SQLGetInfo	<p>A record that allows you to override values returned by calls to SQLGetInfo.</p> <p>See the SQLGetInfo section for more information.</p>
SQLGetTypeInfo	<p>A table, or function that returns a table, that overrides the type information returned by SQLGetTypeInfo.</p> <p>When the value is set to a table, the value completely replaces the type information reported by the driver. SQLGetTypeInfo won't be called.</p> <p>When the value is set to a function, your function will receive the result of the original call to SQLGetTypeInfo, allowing you to modify the table.</p> <p>This field is typically used when there's a mismatch between data types reported by SQLGetTypeInfo and SQLColumns.</p> <p>See the SQLGetTypeInfo section for more information.</p>
SQLTables	<p>A function that allows you to modify the table metadata returned by a call to SQLTables.</p> <p>See the SQLTables section for more information.</p>
TolerateConcatOverflow	<p>Allows conversion of numeric and text types to larger types if an operation would cause the value to fall out of range of the original type.</p> <p>For example, when adding Int32.MaxValue + Int32.MaxValue, the engine will cast the result to Int64 when this setting is set to true. When adding a VARCHAR(4000) and a VARCHAR(4000) field on a system that supports a maximize VARCHAR size of 4000, the engine will cast the result into a CLOB type.</p> <p>Default: false</p>

FIELD	DESCRIPTION
UseEmbeddedDriver	<p>(internal use): A logical value that controls whether the ODBC driver should be loaded from a local directory (using new functionality defined in the ODBC 4.0 specification). This is generally only set by connectors created by Microsoft that ship with Power Query.</p> <p>When set to false, the system ODBC driver manager will be used to locate and load the driver.</p> <p>Most connectors should not need to set this field.</p> <p>Default: false</p>

Overriding AstVisitor

The AstVisitor field is set through the [Odbc.DataSource](#) options record. It's used to modify SQL statements generated for specific query scenarios.

NOTE

Drivers that support `LIMIT` and `OFFSET` clauses (rather than `TOP`) will want to provide a LimitClause override for AstVisitor.

Constant

Providing an override for this value has been deprecated and may be removed from future implementations.

LimitClause

This field is a function that receives two `Int64.Type` arguments (skip, take), and returns a record with two text fields (Text, Location).

```
LimitClause = (skip as nullable number, take as number) as record => ...
```

The skip parameter is the number of rows to skip (that is, the argument to `OFFSET`). If an offset is not specified, the skip value will be null. If your driver supports `LIMIT`, but does not support `OFFSET`, the LimitClause function should return an unimplemented error (...) when skip is greater than 0.

The take parameter is the number of rows to take (that is, the argument to `LIMIT`).

The `Text` field of the result contains the SQL text to add to the generated query.

The `Location` field specifies where to insert the clause. The following table describes supported values.

VALUE	DESCRIPTION	EXAMPLE
AfterQuerySpecification	<p>LIMIT clause is put at the end of the generated SQL.</p> <p>This is the most commonly supported LIMIT syntax.</p>	<pre>SELECT a, b, c FROM table WHERE a > 10 LIMIT 5</pre>

Value	Description	Example
BeforeQuerySpecification	LIMIT clause is put before the generated SQL statement.	LIMIT 5 ROWS SELECT a, b, c FROM table WHERE a > 10
AfterSelect	LIMIT goes after the SELECT statement, and after any modifiers (such as DISTINCT).	SELECT DISTINCT LIMIT 5 a, b, c FROM table WHERE a > 10
AfterSelectBeforeModifiers	LIMIT goes after the SELECT statement, but before any modifiers (such as DISTINCT).	SELECT LIMIT 5 DISTINCT a, b, c FROM table WHERE a > 10

The following code snippet provides a LimitClause implementation for a driver that expects a LIMIT clause, with an optional OFFSET, in the following format: `[OFFSET <offset> ROWS] LIMIT <row_count>`

```
LimitClause = (skip, take) =>
  let
    offset = if (skip > 0) then Text.Format("OFFSET #{0} ROWS", {skip}) else "",
    limit = if (take <> null) then Text.Format("LIMIT #{0}", {take}) else ""
  in
  [
    Text = Text.Format("#{0} #{1}", {offset, limit}),
    Location = "AfterQuerySpecification"
  ]
```

The following code snippet provides a LimitClause implementation for a driver that supports LIMIT, but not OFFSET. Format: `LIMIT <row_count>`.

```
LimitClause = (skip, take) =>
  if (skip > 0) then error "Skip/Offset not supported"
  else
  [
    Text = Text.Format("LIMIT #{0}", {take}),
    Location = "AfterQuerySpecification"
  ]
```

Overriding SqlCapabilities

Field	Details
FractionalSecondsScale	A number value ranging from 1 to 7 that indicates the number of decimal places supported for millisecond values. This value should be set by connectors that want to enable query folding over datetime values. Default: null

FIELD	DETAILS
PrepareStatements	<p>A logical value that indicates that statements should be prepared using SQLPrepare.</p> <p>Default: false</p>
SupportsTop	<p>A logical value that indicates the driver supports the TOP clause to limit the number of returned rows.</p> <p>Default: false</p>
StringLiteralEscapeCharacters	<p>A list of text values that specify the character(s) to use when escaping string literals and LIKE expressions.</p> <p>Ex. {"\r\n"}</p> <p>Default: null</p>
SupportsDerivedTable	<p>A logical value that indicates the driver supports derived tables (sub-selects).</p> <p>This value is assumed to be true for drivers that set their conformance level to SQL_SC_SQL92_FULL (reported by the driver or overridden with the <code>Sql92Conformance</code> setting (see below)). For all other conformance levels, this value defaults to false.</p> <p>If your driver doesn't report the SQL_SC_SQL92_FULL compliance level, but does support derived tables, set this value to true.</p> <p>Note that supporting derived tables is required for many Direct Query scenarios.</p>
SupportsNumericLiterals	<p>A logical value that indicates whether the generated SQL should include numeric literals values. When set to false, numeric values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsStringLiterals	<p>A logical value that indicates whether the generated SQL should include string literals values. When set to false, string values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsOdbcDateLiterals	<p>A logical value that indicates whether the generated SQL should include date literals values. When set to false, date values will always be specified using Parameter Binding.</p> <p>Default: false</p>

FIELD	DETAILS
SupportsOdbcTimeLiterals	A logical value that indicates whether the generated SQL should include time literals values. When set to false, time values will always be specified using Parameter Binding. Default: false
SupportsOdbcTimestampLiterals	A logical value that indicates whether the generated SQL should include timestamp literals values. When set to false, timestamp values will always be specified using Parameter Binding. Default: false

Overriding SQLColumns

`SQLColumns` is a function handler that receives the results of an ODBC call to [SQLColumns](#). The source parameter contains a table with the data type information. This override is typically used to fix up data type mismatches between calls to `SQLGetTypeInfo` and `SQLColumns`.

For details of the format of the source table parameter, see:

<https://docs.microsoft.com/sql/odbc/reference/syntax/sqlcolumns-function>

Overriding SQLGetFunctions

This field is used to override SQLFunctions values returned by an ODBC driver. It contains a record whose field names are equal to the FunctionId constants defined for the ODBC [SQLGetFunctions](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#).

FIELD	DETAILS
SQL_CONVERT_FUNCTIONS	Indicates which function(s) are supported when doing type conversions. By default, the M Engine will attempt to use the CONVERT function. Drivers that prefer the use of CAST can override this value to report that only SQL_FN_CVT_CAST (numeric value of 0x2) is supported.
SQL_API_SQLBINDCOL	A logical (true/false) value that indicates whether the Mashup Engine should use the SQLBindCol API when retrieving data. When set to false, SQLGetData is used instead. Default: false

The following code snippet provides an example explicitly telling the M engine to use CAST rather than CONVERT.

```
SQLGetFunctions = [
    SQL_CONVERT_FUNCTIONS = 0x2 /* SQL_FN_CVT_CAST */
]
```

Overriding SQLGetInfo

This field is used to override SQLGetInfo values returned by an ODBC driver. It contains a record whose fields are names equal to the InfoType constants defined for the ODBC [SQLGetInfo](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#). The full list of InfoTypes that are checked can be found in the Mashup Engine trace files.

The following table contains commonly overridden SQLGetInfo properties:

FIELD	DETAILS
SQL_SQL_CONFORMANCE	An integer value that indicates the level of SQL-92 supported by the driver: (1) SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant. (2) SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant. (4) SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant. (8) SQL_SC_SQL92_FULL = Full level SQL-92 compliant. Note that in Power Query scenarios, the connector will be used in a Read Only mode. Most drivers will want to report a SQL_SC_SQL92_FULL compliance level, and override specific SQL generation behavior using the SQLGetInfo and SQLGetFunctions properties.
SQL_SQL92_PREDICATES	A bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92. See the SQL_SP_* constants in the ODBC specification.
SQL_AGGREGATE_FUNCTIONS	A bitmask enumerating support for aggregation functions. SQL_AF_ALL SQL_AF_AVG SQL_AF_COUNT SQL_AF_DISTINCT SQL_AF_MAX SQL_AF_MIN SQL_AF_SUM See the SQL_AF_* constants in the ODBC specification.

FIELD	DETAILS
SQL_GROUP_BY	<p>A integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the select list:</p> <p>SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column.</p> <p>SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.</p> <p>SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.</p> <p>SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is data source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>See the SQL_GB_* constants in the ODBC specification.</p>

The following helper function can be used to create bitmask values from a list of integer values:

```
Flags = (flags as list) =>
let
    Loop = List.Generate(
        ()=> [i = 0, Combined = 0],
        each [i] < List.Count(flags),
        each [i = [i]+1, Combined =*Number.BitwiseOr([Combined], flags{i})],
        each [Combined]),
    Result = List.Last(Loop, 0)
in
    Result;
```

Overriding SQLGetTypeInfo

`SQLGetTypeInfo` can be specified in two ways:

- A fixed `table` value that contains the same type information as an ODBC call to `SQLGetTypeInfo`.
- A function that accepts a table argument, and returns a table. The argument will contain the original results of the ODBC call to `SQLGetTypeInfo`. Your function implementation can modify/add to this table.

The first approach is used to completely override the values returned by the ODBC driver. The second approach is used if you want to add to or modify these values.

For details of the format of the types table parameter and expected return value, see the [SQLGetTypeInfo function reference](#).

SQLGetTypeInfo using a static table

The following code snippet provides a static implementation for `SQLGetTypeInfo`.

```
SQLGetTypeInfo = #table{
```

```

    { "TYPE_NAME",      "DATA_TYPE", "COLUMN_SIZE", "LITERAL_PREF", "LITERAL_SUFFIX", "CREATE_PARAS",
"NULLABLE", "CASE_SENSITIVE", "SEARCHABLE", "UNSIGNED_ATTRIBUTE", "FIXED_PREC_SCALE", "AUTO_UNIQUE_VALUE",
"LOCAL_TYPE_NAME", "MINIMUM_SCALE", "MAXIMUM_SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "NUM_PREC_RADIX",
"INTERNAL_PRECISION", "USER_DATA_TYPE" }, {
    { "char",           1,          65535,      "",      "",      "max. length",
1,           1,          3,          null,      0,          null,
"char",        null,        null,      -8,          null,      null,
0,           },      1,          19,         0,          10,         0,
    { "int8",          -5,          2,          0,          -5,          2,
1,           0,          0,          0,          null,      2,
"int8",        0,          0,          0,          null,      0,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          9,          10,         "",         null,
1,           0,          2,          null,      0,          null,
"date",        null,        null,      9,          1,          null,
0,           },      1,          3,          28,         null,      null,
1,           0,          2,          0,          0,          0,
"numeric",      0,          0,          2,          null,      10,
0,           },      1,          8,          15,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          6,          17,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          -11,         2,         37,         null,
1,           0,          2,         null,      0,          null,
"uuid",        null,        null,      -11,         null,      null,
0,           },      1,          4,          10,         null,      null,
1,           0,          2,          0,          4,          0,
"int4",        0,          0,          4,          null,      2,
0,           },      1,          -1,         65535,      "",      null,
1,           1,          3,          null,      0,          null,
"text",        null,        null,      -10,         null,      null,
0,           },      1,          -4,         255,         "",         null,
1,           0,          2,          null,      0,          null,
"lo",        null,        null,      -4,          null,      null,
0,           },      1,          2,          28,         null,      null,
1,           0,          2,          0,          10,         0,
"numeric",      0,          6,          2,          null,      10,
0,           },      1,          7,          9,          null,      null,
1,           0,          2,          0,          10,         0,
"float4",      null,        null,      7,          null,      2,
0,           },      1,          5,          19,         null,      null,
1,           0,          2,          0,          5,          null,
"int2",        0,          0,          5,          null,      2,
0,           },      1,          -6,         2,          5,          null,
1,           0,          0,          0,          10,         0,
"int2",        0,          0,          5,          null,      2,
0,           }

```

```

        },
        { "timestamp",      11,      26,      "",      "",      null,
1,          0,           2,       null,      9,      0,      null,
"timestamp",      0,           38,       null,      9,      3,      null,
0,           },      91,      10,      "",      "",      null,
1,          0,           2,       null,      9,      1,      null,
"date",      null,       null,      9,      1,      null,
0,           },      93,      26,      "",      "",      null,
1,          0,           2,       null,      9,      3,      null,
"timestamp",      0,           38,       null,      9,      0,      null,
0,           },      -3,      255,      "",      "",      null,
1,          0,           2,       null,      -3,      0,      null,
"bytea",      null,       null,      -3,      null,      null,
0,           },      12,      65535,      "",      "",      "max. length",
1,          0,           2,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -8,      0,      null,
"char",      null,       null,      -8,      null,      null,
0,           },      -10,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -10,      0,      null,
"text",      null,       null,      -10,      null,      null,
0,           },      -9,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"bpchar",      null,       null,      -9,      null,      null,
0,           } } }
);

```

SQLGetTypeInfo using a function

The following code snippets append the `bpchar` type to the existing types returned by the driver.

```

SQLGetTypeInfo = (types as table) as table =>
let
    newTypes = #table(
    {
        "TYPE_NAME",
        "DATA_TYPE",
        "COLUMN_SIZE",
        "LITERAL_PREF",
        "LITERAL_SUFFIX",
        "CREATE_PARAS",
        "NULLABLE",
        "CASE_SENSITIVE",
        "SEARCHABLE",
        "UNSIGNED_ATTRIBUTE",
        "FIXED_PREC_SCALE",
        "AUTO_UNIQUE_VALUE",
        "LOCAL_TYPE_NAME",
        "MINIMUM_SCALE",
        "MAXIMUM_SCALE",
        "SQL_DATA_TYPE",
        "SQL_DATETIME_SUB",
        "NUM_PREC_RADIX",
        "INTERNAL_PRECISION",
        "USER_DATA_TYPE"
    },
    // we add a new entry for each type we want to add
    {
        {
            "bpchar",
            -8,
            65535,
            """",
            """",
            "max. length",
            1,
            1,
            3,
            null,
            0,
            null,
            "bpchar",
            null,
            null,
            -9,
            null,
            null,
            0,
            0
        }
    )),
    append = Table.Combine({types, newTypes})
in
    append;

```

Setting the Connection String

The connection string for your ODBC driver is set using the first argument to the [Odbc.DataSource](#) and/or [Odbc.Query](#) functions. The value can be text, or an M record. When using the record, each field in the record will become a property in the connection string. All connection strings will require a Driver field (or DSN field if you require users to pre-configure a system level DSN). Credential related properties will be set separately (see below). Other properties will be driver specific.

The code snippet below shows the definition of a new data source function, creation of the `ConnectionString` record, and invocation of the [Odbc.DataSource](#) function.

```
[DataSource.Kind="SqlODBC", Publish="SqlODBC.Publish"]
shared SqlODBC.Contents = (server as text) =>
    let
        ConnectionString = [
            Driver = "SQL Server Native Client 11.0",
            Server = server,
            MultiSubnetFailover = "Yes",
            ApplicationIntent = "ReadOnly",
            APP = "PowerBICustomConnector"
        ],
        OdbcDatasource = Odbc.DataSource(ConnectionString)
    in
        OdbcDatasource;
```

Troubleshooting and Testing

To enable tracing in Power BI Desktop:

1. Go to **File > Options and settings > Options**.
2. Select on the **Diagnostics** tab.
3. Select the **Enable tracing** option.
4. Select the **Open traces folder** link (should be `%LOCALAPPDATA%/Microsoft/Power BI Desktop/Traces`).
5. Delete existing trace files.
6. Perform your tests.
7. Close Power BI Desktop to ensure all log files are flushed to disk.

Here are steps you can take for initial testing in Power BI Desktop:

1. Close Power BI Desktop.
2. Clear your trace directory.
3. Open Power BI desktop, and enable tracing.
4. Connect to your data source, and select Direct Query mode.
5. Select a table in the navigator, and select **Edit**.
6. Manipulate the query in various ways, including:

- Take the First N rows (for example, 10).
- Set equality filters on different data types (int, string, bool, and so on).
- Set other range filters (greater than, less than).
- Filter on NULL / NOT NULL.
- Select a sub-set of columns.
- Aggregate / Group By different column combinations.
- Add a column calculated from other columns (`[C] = [A] + [B]`).
- Sort on one column, multiple columns. 7. Expressions that fail to fold will result in a warning bar. Note the failure, remove the step, and move to the next test case. Details about the cause of the failure should be emitted to the trace logs. 8. Close Power BI Desktop. 9. Copy the trace files to a new directory. 10. Use the recommend Power BI workbook to parse and analyze the trace files.

Once you have simple queries working, you can then try Direct Query scenarios (for example, building reports in the Report Views). The queries generated in Direct Query mode will be significantly more complex (that is, use of sub-selects, COALESCE statements, and aggregations).

Concatenation of strings in Direct Query mode

The M engine does basic type size limit validation as part of its query folding logic. If you are receiving a folding

error when trying to concatenate two strings that potentially overflow the maximum size of the underlying database type:

1. Ensure that your database can support up-conversion to CLOB types when string concat overflow occurs.
2. Set the `TolerateConcatOverflow` option for Odbc.DataSource to `true`.

The [DAX CONCATENATE function](#) is currently not supported by Power Query/ODBC extensions. Extension authors should ensure string concatenation works through the query editor by adding calculated columns (`[stringCol1] & [stringCol2]`). When the capability to fold the CONCATENATE operation is added in the future, it should work seamlessly with existing extensions.

Enabling Direct Query for an ODBC based connector

21 minutes to read • [Edit Online](#)

Overview

Using M's built-in [Odbc.DataSource](#) function is the recommended way to create custom connectors for data sources that have an existing ODBC driver and/or support a SQL query syntax. Wrapping the [Odbc.DataSource](#) function will allow your connector to inherit default query folding behavior based on the capabilities reported by your driver. This will enable the M engine to generate SQL statements based on filters and other transformations defined by the user within the Power Query experience, without having to provide this logic within the connector itself.

ODBC extensions can optionally enable Direct Query mode, allowing Power BI to dynamically generate queries at runtime without pre-caching the user's data model.

NOTE

Enabling Direct Query support raises the difficulty and complexity level of your connector. When Direct Query is enabled, Power BI will prevent the M engine from compensating for operations that cannot be fully pushed to the underlying data source.

This section builds on the concepts presented in the M Extensibility Reference, and assumes familiarity with the creation of a basic Data Connector.

Refer to the [SqlODBC sample](#) for most of the code examples in the sections below. Additional samples can be found in the ODBC samples directory.

ODBC Extensibility Functions

The M engine provides two ODBC related data source functions: [Odbc.DataSource](#), and [Odbc.Query](#).

The [Odbc.DataSource](#) function provides a default navigation table with all databases, tables, and views from your system, supports query folding, and allows for a range of customization options. The majority of ODBC based extensions will use this as their primary extensibility function. The function accepts two arguments—a connection string, and an options record to provide behavior overrides.

The [Odbc.Query](#) function allows you to execute SQL statements through an ODBC driver. It acts as a passthrough for query execution. Unlike the [Odbc.DataSource](#) function, it doesn't provide query folding functionality, and requires that SQL queries be provided by the connector (or end user). When building a custom connector, this function is typically used internally to run queries to retrieve metadata that might not be exposed through regular ODBC channels. The function accepts two arguments—a connection string, and a SQL query.

Parameters for your Data Source Function

Custom connectors can accept any number of function arguments, but to remain consistent with the built-in data source functions shipped with Power Query, the following guidelines are recommended:

- Require the minimal set of parameters used to establish a connection to your server. The less parameters end users need to provide, the easier your connector will be to use.
- Although you can define parameters with a fixed number of values (that is, a dropdown list in the UI), parameters are entered before the user is authenticated. Any values that can be discovered

programmatically after the user is authenticated (such as catalog or database name) should be selectable through the Navigator. The default behavior for the [Odbc.DataSource](#) function will be to return a hierarchical navigation table consisting of Catalog (Database), Schema, and Table names, although this can be overridden within your connector.

- If you feel your users will typically know what values to enter for items they would select from the Navigator (such as the database name), make these parameters optional. Parameters that can be discovered programmatically should not be made required.
- The last parameter for your function should be an optional record called "options". This parameter typically allows advanced users to set common ODBC related properties (such as CommandTimeout), set behavior overrides specific to your connector, and allows for future extensibility without impacting backwards compatibility for your function.
- Security/credential related arguments MUST never be part of your data source function parameters, as values entered in the connect dialog will be persisted to the user's query. Credential related parameters should be specified as part of the connector's supported Authentication methods.

By default, all required parameters for your data source function are factored into the Data Source Path value used to identify user credentials.

Note that while the UI for the built-in [Odbc.DataSource](#) function provides a dropdown that allows the user to select a DSN, this functionality is not available through extensibility. If your data source configuration is complex enough to require a fully customizable configuration dialog, it's recommended you require your end users to pre-configure a system DSN, and have your function take in the DSN name as a text field.

Parameters for Odbc.DataSource

The [Odbc.DataSource](#) function takes two parameters—a connectionString for your driver, and an options record that lets you override various driver behaviors. Through the options record you can override capabilities and other information reported by the driver, control the navigator behavior, and affect the SQL queries generated by the M engine.

The supported options records fields fall into two categories—those that are public / always available, and those that are only available in an extensibility context.

The following table describes the public fields in the options record.

FIELD	DESCRIPTION
CommandTimeout	A duration value that controls how long the server-side query is allowed to run before it's cancelled. Default: 10 minutes
ConnectionTimeout	A duration value that controls how long to wait before abandoning an attempt to make a connection to the server. Default: 15 seconds

FIELD	DESCRIPTION
CreateNavigationProperties	<p>A logical value that sets whether to generate navigation properties on the returned tables. Navigation properties are based on foreign key relationships reported by the driver, and show up as "virtual" columns that can be expanded in the query editor, creating the appropriate join.</p> <p>If calculating foreign key dependencies is an expensive operation for your driver, you may want to set this value to false.</p> <p>Default: true</p>
HierarchicalNavigation	<p>A logical value that sets whether to view the tables grouped by their schema names. When set to false, tables will be displayed in a flat list under each database.</p> <p>Default: false</p>
SqlCompatibleWindowsAuth	<p>A logical value that determines whether to produce a SQL Server compatible connection string when using Windows Authentication—Trusted_Connection=Yes.</p> <p>If your driver supports Windows Authentication, but requires additional or alternative settings in your connection string, you should set this value to false and use the CredentialConnectionString option record field described below.</p> <p>Default: true</p>

The following table describes the options record fields that are only available through extensibility. Fields that aren't simple literal values are described in subsequent sections.

FIELD	DESCRIPTION
AstVisitor	<p>A record containing one or more overrides to control SQL query generation. The most common usage of this field is to provide logic to generate a LIMIT/OFFSET clause for drivers that don't support TOP.</p> <p>Fields include:</p> <ul style="list-style-type: none"> • Constant • LimitClause <p>See the AstVisitor section for more information.</p>
ClientConnectionPooling	<p>A logical value that enables client-side connection pooling for the ODBC driver. Most drivers will want to set this value to true.</p> <p>Default: false</p>

FIELD	DESCRIPTION
CredentialConnectionString	<p>A text or record value used to specify credential related connection string properties.</p> <p>See the Credential section for more information.</p>
HideNativeQuery	<p>A logical value that controls whether your connector allows native SQL statements to be passed in by a query using the Value.NativeQuery() function.</p> <p>Note: this functionality is currently not exposed in the Power Query user experience. Users would need to manually edit their queries to take advantage of this capability.</p> <p>Default: false</p>
ImplicitTypeConversions	<p>A table value containing implicit type conversions supported by your driver or backend server. Values in this table are additive to the conversions reported by the driver itself.</p> <p>This field is typically used in conjunction with the SQLGetTypeInfo field when overriding data type information reported by the driver.</p> <p>See the ImplicitTypeConversions section for more information.</p>
OnError	<p>An error handling function that receives an errorRecord parameter of type record.</p> <p>Common uses of this function include handling SSL connection failures, providing a download link if your driver isn't found on the system, and reporting authentication errors.</p> <p>See the OnError section for more information.</p>
SoftNumbers	<p>Allows the M engine to select a compatible data type when conversion between two specific numeric types isn't declared as supported in the SQL_CONVERT_* capabilities.</p> <p>Default: false</p>
SqlCapabilities	<p>A record providing various overrides of driver capabilities, and a way to specify capabilities that aren't expressed through ODBC 3.8.</p> <p>See the SqlCapabilities section for more information.</p>
SQLColumns	<p>A function that allows you to modify column metadata returned by the SQLColumns function.</p> <p>See the SQLColumns section for more information.</p>

FIELD	DESCRIPTION
SQLGetFunctions	<p>A record that allows you to override values returned by calls to SQLGetFunctions.</p> <p>A common use of this field is to disable the use of parameter binding, or to specify that generated queries should use CAST rather than CONVERT.</p> <p>See the SQLGetFunctions section for more information.</p>
SQLGetInfo	<p>A record that allows you to override values returned by calls to SQLGetInfo.</p> <p>See the SQLGetInfo section for more information.</p>
SQLGetTypeInfo	<p>A table, or function that returns a table, that overrides the type information returned by SQLGetTypeInfo.</p> <p>When the value is set to a table, the value completely replaces the type information reported by the driver. SQLGetTypeInfo won't be called.</p> <p>When the value is set to a function, your function will receive the result of the original call to SQLGetTypeInfo, allowing you to modify the table.</p> <p>This field is typically used when there's a mismatch between data types reported by SQLGetTypeInfo and SQLColumns.</p> <p>See the SQLGetTypeInfo section for more information.</p>
SQLTables	<p>A function that allows you to modify the table metadata returned by a call to SQLTables.</p> <p>See the SQLTables section for more information.</p>
TolerateConcatOverflow	<p>Allows conversion of numeric and text types to larger types if an operation would cause the value to fall out of range of the original type.</p> <p>For example, when adding Int32.MaxValue + Int32.MaxValue, the engine will cast the result to Int64 when this setting is set to true. When adding a VARCHAR(4000) and a VARCHAR(4000) field on a system that supports a maximize VARCHAR size of 4000, the engine will cast the result into a CLOB type.</p> <p>Default: false</p>

FIELD	DESCRIPTION
UseEmbeddedDriver	<p>(internal use): A logical value that controls whether the ODBC driver should be loaded from a local directory (using new functionality defined in the ODBC 4.0 specification). This is generally only set by connectors created by Microsoft that ship with Power Query.</p> <p>When set to false, the system ODBC driver manager will be used to locate and load the driver.</p> <p>Most connectors should not need to set this field.</p> <p>Default: false</p>

Overriding AstVisitor

The AstVisitor field is set through the [Odbc.DataSource](#) options record. It's used to modify SQL statements generated for specific query scenarios.

NOTE

Drivers that support `LIMIT` and `OFFSET` clauses (rather than `TOP`) will want to provide a LimitClause override for AstVisitor.

Constant

Providing an override for this value has been deprecated and may be removed from future implementations.

LimitClause

This field is a function that receives two `Int64.Type` arguments (skip, take), and returns a record with two text fields (Text, Location).

```
LimitClause = (skip as nullable number, take as number) as record => ...
```

The skip parameter is the number of rows to skip (that is, the argument to `OFFSET`). If an offset is not specified, the skip value will be null. If your driver supports `LIMIT`, but does not support `OFFSET`, the LimitClause function should return an unimplemented error (...) when skip is greater than 0.

The take parameter is the number of rows to take (that is, the argument to `LIMIT`).

The `Text` field of the result contains the SQL text to add to the generated query.

The `Location` field specifies where to insert the clause. The following table describes supported values.

VALUE	DESCRIPTION	EXAMPLE
AfterQuerySpecification	<p>LIMIT clause is put at the end of the generated SQL.</p> <p>This is the most commonly supported LIMIT syntax.</p>	<pre>SELECT a, b, c FROM table WHERE a > 10 LIMIT 5</pre>

Value	Description	Example
BeforeQuerySpecification	LIMIT clause is put before the generated SQL statement.	LIMIT 5 ROWS SELECT a, b, c FROM table WHERE a > 10
AfterSelect	LIMIT goes after the SELECT statement, and after any modifiers (such as DISTINCT).	SELECT DISTINCT LIMIT 5 a, b, c FROM table WHERE a > 10
AfterSelectBeforeModifiers	LIMIT goes after the SELECT statement, but before any modifiers (such as DISTINCT).	SELECT LIMIT 5 DISTINCT a, b, c FROM table WHERE a > 10

The following code snippet provides a LimitClause implementation for a driver that expects a LIMIT clause, with an optional OFFSET, in the following format: `[OFFSET <offset> ROWS] LIMIT <row_count>`

```
LimitClause = (skip, take) =>
  let
    offset = if (skip > 0) then Text.Format("OFFSET #{0} ROWS", {skip}) else "",
    limit = if (take <> null) then Text.Format("LIMIT #{0}", {take}) else ""
  in
  [
    Text = Text.Format("#{0} #{1}", {offset, limit}),
    Location = "AfterQuerySpecification"
  ]
```

The following code snippet provides a LimitClause implementation for a driver that supports LIMIT, but not OFFSET. Format: `LIMIT <row_count>`.

```
LimitClause = (skip, take) =>
  if (skip > 0) then error "Skip/Offset not supported"
  else
  [
    Text = Text.Format("LIMIT #{0}", {take}),
    Location = "AfterQuerySpecification"
  ]
```

Overriding SqlCapabilities

Field	Details
FractionalSecondsScale	A number value ranging from 1 to 7 that indicates the number of decimal places supported for millisecond values. This value should be set by connectors that want to enable query folding over datetime values. Default: null

FIELD	DETAILS
PrepareStatements	<p>A logical value that indicates that statements should be prepared using SQLPrepare.</p> <p>Default: false</p>
SupportsTop	<p>A logical value that indicates the driver supports the TOP clause to limit the number of returned rows.</p> <p>Default: false</p>
StringLiteralEscapeCharacters	<p>A list of text values that specify the character(s) to use when escaping string literals and LIKE expressions.</p> <p>Ex. {"\r\n"}</p> <p>Default: null</p>
SupportsDerivedTable	<p>A logical value that indicates the driver supports derived tables (sub-selects).</p> <p>This value is assumed to be true for drivers that set their conformance level to SQL_SC_SQL92_FULL (reported by the driver or overridden with the <code>Sql92Conformance</code> setting (see below)). For all other conformance levels, this value defaults to false.</p> <p>If your driver doesn't report the SQL_SC_SQL92_FULL compliance level, but does support derived tables, set this value to true.</p> <p>Note that supporting derived tables is required for many Direct Query scenarios.</p>
SupportsNumericLiterals	<p>A logical value that indicates whether the generated SQL should include numeric literals values. When set to false, numeric values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsStringLiterals	<p>A logical value that indicates whether the generated SQL should include string literals values. When set to false, string values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsOdbcDateLiterals	<p>A logical value that indicates whether the generated SQL should include date literals values. When set to false, date values will always be specified using Parameter Binding.</p> <p>Default: false</p>

FIELD	DETAILS
SupportsOdbcTimeLiterals	A logical value that indicates whether the generated SQL should include time literals values. When set to false, time values will always be specified using Parameter Binding. Default: false
SupportsOdbcTimestampLiterals	A logical value that indicates whether the generated SQL should include timestamp literals values. When set to false, timestamp values will always be specified using Parameter Binding. Default: false

Overriding SQLColumns

`SQLColumns` is a function handler that receives the results of an ODBC call to [SQLColumns](#). The source parameter contains a table with the data type information. This override is typically used to fix up data type mismatches between calls to `SQLGetTypeInfo` and `SQLColumns`.

For details of the format of the source table parameter, see:

<https://docs.microsoft.com/sql/odbc/reference/syntax/sqlcolumns-function>

Overriding SQLGetFunctions

This field is used to override SQLFunctions values returned by an ODBC driver. It contains a record whose field names are equal to the FunctionId constants defined for the ODBC [SQLGetFunctions](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#).

FIELD	DETAILS
SQL_CONVERT_FUNCTIONS	Indicates which function(s) are supported when doing type conversions. By default, the M Engine will attempt to use the CONVERT function. Drivers that prefer the use of CAST can override this value to report that only SQL_FN_CVT_CAST (numeric value of 0x2) is supported.
SQL_API_SQLBINDCOL	A logical (true/false) value that indicates whether the Mashup Engine should use the SQLBindCol API when retrieving data. When set to false, SQLGetData is used instead. Default: false

The following code snippet provides an example explicitly telling the M engine to use CAST rather than CONVERT.

```
SQLGetFunctions = [
    SQL_CONVERT_FUNCTIONS = 0x2 /* SQL_FN_CVT_CAST */
]
```

Overriding SQLGetInfo

This field is used to override SQLGetInfo values returned by an ODBC driver. It contains a record whose fields are names equal to the InfoType constants defined for the ODBC [SQLGetInfo](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#). The full list of InfoTypes that are checked can be found in the Mashup Engine trace files.

The following table contains commonly overridden SQLGetInfo properties:

FIELD	DETAILS
SQL_SQL_CONFORMANCE	An integer value that indicates the level of SQL-92 supported by the driver: (1) SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant. (2) SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant. (4) SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant. (8) SQL_SC_SQL92_FULL = Full level SQL-92 compliant. Note that in Power Query scenarios, the connector will be used in a Read Only mode. Most drivers will want to report a SQL_SC_SQL92_FULL compliance level, and override specific SQL generation behavior using the SQLGetInfo and SQLGetFunctions properties.
SQL_SQL92_PREDICATES	A bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92. See the SQL_SP_* constants in the ODBC specification.
SQL_AGGREGATE_FUNCTIONS	A bitmask enumerating support for aggregation functions. SQL_AF_ALL SQL_AF_AVG SQL_AF_COUNT SQL_AF_DISTINCT SQL_AF_MAX SQL_AF_MIN SQL_AF_SUM See the SQL_AF_* constants in the ODBC specification.

FIELD	DETAILS
SQL_GROUP_BY	<p>A integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the select list:</p> <p>SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column.</p> <p>SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.</p> <p>SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.</p> <p>SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is data source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>See the SQL_GB_* constants in the ODBC specification.</p>

The following helper function can be used to create bitmask values from a list of integer values:

```
Flags = (flags as list) =>
let
    Loop = List.Generate(
        ()=> [i = 0, Combined = 0],
        each [i] < List.Count(flags),
        each [i = [i]+1, Combined =*Number.BitwiseOr([Combined], flags{i})],
        each [Combined]),
    Result = List.Last(Loop, 0)
in
    Result;
```

Overriding SQLGetTypeInfo

`SQLGetTypeInfo` can be specified in two ways:

- A fixed `table` value that contains the same type information as an ODBC call to `SQLGetTypeInfo`.
- A function that accepts a table argument, and returns a table. The argument will contain the original results of the ODBC call to `SQLGetTypeInfo`. Your function implementation can modify/add to this table.

The first approach is used to completely override the values returned by the ODBC driver. The second approach is used if you want to add to or modify these values.

For details of the format of the types table parameter and expected return value, see the [SQLGetTypeInfo function reference](#).

SQLGetTypeInfo using a static table

The following code snippet provides a static implementation for `SQLGetTypeInfo`.

```
SQLGetTypeInfo = #table{
```

```

    { "TYPE_NAME",      "DATA_TYPE", "COLUMN_SIZE", "LITERAL_PREF", "LITERAL_SUFFIX", "CREATE_PARAS",
"NULLABLE", "CASE_SENSITIVE", "SEARCHABLE", "UNSIGNED_ATTRIBUTE", "FIXED_PREC_SCALE", "AUTO_UNIQUE_VALUE",
"LOCAL_TYPE_NAME", "MINIMUM_SCALE", "MAXIMUM_SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "NUM_PREC_RADIX",
"INTERNAL_PRECISION", "USER_DATA_TYPE" }, {
    { "char",           1,          65535,      "",      "",      "max. length",
1,           1,          3,          null,      0,          null,
"char",        null,        null,      -8,          null,      null,
0,           },      1,          19,         0,          10,         0,
    { "int8",          -5,          2,          0,          -5,          2,
1,           0,          0,          0,          null,      2,
"int8",        0,          0,          0,          null,      0,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          9,          10,         "",         null,
1,           0,          2,          null,      0,          null,
"date",        null,        null,      9,          1,          null,
0,           },      1,          3,          28,         null,      null,
1,           0,          2,          0,          0,          0,
"numeric",      0,          0,          2,          null,      10,
0,           },      1,          8,          15,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          6,          17,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          -11,         2,         37,         null,
1,           0,          2,         null,      0,          null,
"uuid",        null,        null,      -11,         null,      null,
0,           },      1,          4,          10,         null,      null,
1,           0,          2,          0,          4,          0,
"int4",        0,          0,          4,          null,      2,
0,           },      1,          -1,         65535,      "",      null,
1,           1,          3,          null,      0,          null,
"text",        null,        null,      -10,         null,      null,
0,           },      1,          -4,         255,         "",         null,
1,           0,          2,          null,      0,          null,
"lo",        null,        null,      -4,          null,      null,
0,           },      1,          2,          28,         null,      null,
1,           0,          2,          0,          10,         0,
"numeric",      0,          6,          2,          null,      10,
0,           },      1,          7,          9,          null,      null,
1,           0,          2,          0,          10,         0,
"float4",      null,        null,      7,          null,      2,
0,           },      1,          5,          19,         null,      null,
1,           0,          2,          0,          5,          null,
"int2",        0,          0,          5,          null,      2,
0,           },      1,          -6,         2,          5,          null,
1,           0,          0,          0,          10,         0,
"int2",        0,          0,          5,          null,      2,
0,           }

```

```

        },
        { "timestamp",      11,      26,      "",      "",      null,
1,          0,           2,       null,      9,      0,      null,
"timestamp",      0,           38,       null,      9,      3,      null,
0,          },      91,      10,      "",      "",      null,
1,          0,           2,       null,      9,      1,      null,
"date",      null,       null,      9,      1,      null,
0,          },      93,      26,      "",      "",      null,
1,          0,           2,       null,      9,      3,      null,
"timestamp",      0,           38,       null,      9,      0,      null,
0,          },      -3,      255,      "",      "",      null,
1,          0,           2,       null,      -3,      0,      null,
"bytea",      null,       null,      -3,      null,      null,
0,          },      12,      65535,      "",      "",      "max. length",
1,          0,           2,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,          },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -8,      0,      null,
"char",      null,       null,      -8,      null,      null,
0,          },      -10,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -10,      0,      null,
"text",      null,       null,      -10,      null,      null,
0,          },      -9,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,          },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"bpchar",      null,       null,      -9,      null,      null,
0,          } } }
);

```

SQLGetTypeInfo using a function

The following code snippets append the `bpchar` type to the existing types returned by the driver.

```

SQLGetTypeInfo = (types as table) as table =>
let
    newTypes = #table(
    {
        "TYPE_NAME",
        "DATA_TYPE",
        "COLUMN_SIZE",
        "LITERAL_PREF",
        "LITERAL_SUFFIX",
        "CREATE_PARAS",
        "NULLABLE",
        "CASE_SENSITIVE",
        "SEARCHABLE",
        "UNSIGNED_ATTRIBUTE",
        "FIXED_PREC_SCALE",
        "AUTO_UNIQUE_VALUE",
        "LOCAL_TYPE_NAME",
        "MINIMUM_SCALE",
        "MAXIMUM_SCALE",
        "SQL_DATA_TYPE",
        "SQL_DATETIME_SUB",
        "NUM_PREC_RADIX",
        "INTERNAL_PRECISION",
        "USER_DATA_TYPE"
    },
    // we add a new entry for each type we want to add
    {
        {
            "bpchar",
            -8,
            65535,
            """",
            """",
            "max. length",
            1,
            1,
            3,
            null,
            0,
            null,
            "bpchar",
            null,
            null,
            -9,
            null,
            null,
            0,
            0
        }
    )),
    append = Table.Combine({types, newTypes})
in
    append;

```

Setting the Connection String

The connection string for your ODBC driver is set using the first argument to the [Odbc.DataSource](#) and/or [Odbc.Query](#) functions. The value can be text, or an M record. When using the record, each field in the record will become a property in the connection string. All connection strings will require a Driver field (or DSN field if you require users to pre-configure a system level DSN). Credential related properties will be set separately (see below). Other properties will be driver specific.

The code snippet below shows the definition of a new data source function, creation of the `ConnectionString` record, and invocation of the [Odbc.DataSource](#) function.

```
[DataSource.Kind="SqlODBC", Publish="SqlODBC.Publish"]
shared SqlODBC.Contents = (server as text) =>
let
    ConnectionString = [
        Driver = "SQL Server Native Client 11.0",
        Server = server,
        MultiSubnetFailover = "Yes",
        ApplicationIntent = "ReadOnly",
        APP = "PowerBICustomConnector"
    ],
    OdbcDatasource = Odbc.DataSource(ConnectionString)
in
    OdbcDatasource;
```

Troubleshooting and Testing

To enable tracing in Power BI Desktop:

1. Go to **File > Options and settings > Options**.
2. Select on the **Diagnostics** tab.
3. Select the **Enable tracing** option.
4. Select the **Open traces folder** link (should be `%LOCALAPPDATA%/Microsoft/Power BI Desktop/Traces`).
5. Delete existing trace files.
6. Perform your tests.
7. Close Power BI Desktop to ensure all log files are flushed to disk.

Here are steps you can take for initial testing in Power BI Desktop:

1. Close Power BI Desktop.
2. Clear your trace directory.
3. Open Power BI desktop, and enable tracing.
4. Connect to your data source, and select Direct Query mode.
5. Select a table in the navigator, and select **Edit**.
6. Manipulate the query in various ways, including:

- Take the First N rows (for example, 10).
- Set equality filters on different data types (int, string, bool, and so on).
- Set other range filters (greater than, less than).
- Filter on NULL / NOT NULL.
- Select a sub-set of columns.
- Aggregate / Group By different column combinations.
- Add a column calculated from other columns (`[C] = [A] + [B]`).
- Sort on one column, multiple columns. 7. Expressions that fail to fold will result in a warning bar. Note the failure, remove the step, and move to the next test case. Details about the cause of the failure should be emitted to the trace logs. 8. Close Power BI Desktop. 9. Copy the trace files to a new directory. 10. Use the recommend Power BI workbook to parse and analyze the trace files.

Once you have simple queries working, you can then try Direct Query scenarios (for example, building reports in the Report Views). The queries generated in Direct Query mode will be significantly more complex (that is, use of sub-selects, COALESCE statements, and aggregations).

Concatenation of strings in Direct Query mode

The M engine does basic type size limit validation as part of its query folding logic. If you are receiving a folding

error when trying to concatenate two strings that potentially overflow the maximum size of the underlying database type:

1. Ensure that your database can support up-conversion to CLOB types when string concat overflow occurs.
2. Set the `TolerateConcatOverflow` option for Odbc.DataSource to `true`.

The [DAX CONCATENATE function](#) is currently not supported by Power Query/ODBC extensions. Extension authors should ensure string concatenation works through the query editor by adding calculated columns (`[stringCol1] & [stringCol2]`). When the capability to fold the CONCATENATE operation is added in the future, it should work seamlessly with existing extensions.

Enabling Direct Query for an ODBC based connector

21 minutes to read • [Edit Online](#)

Overview

Using M's built-in [Odbc.DataSource](#) function is the recommended way to create custom connectors for data sources that have an existing ODBC driver and/or support a SQL query syntax. Wrapping the [Odbc.DataSource](#) function will allow your connector to inherit default query folding behavior based on the capabilities reported by your driver. This will enable the M engine to generate SQL statements based on filters and other transformations defined by the user within the Power Query experience, without having to provide this logic within the connector itself.

ODBC extensions can optionally enable Direct Query mode, allowing Power BI to dynamically generate queries at runtime without pre-caching the user's data model.

NOTE

Enabling Direct Query support raises the difficulty and complexity level of your connector. When Direct Query is enabled, Power BI will prevent the M engine from compensating for operations that cannot be fully pushed to the underlying data source.

This section builds on the concepts presented in the M Extensibility Reference, and assumes familiarity with the creation of a basic Data Connector.

Refer to the [SqlODBC sample](#) for most of the code examples in the sections below. Additional samples can be found in the ODBC samples directory.

ODBC Extensibility Functions

The M engine provides two ODBC related data source functions: [Odbc.DataSource](#), and [Odbc.Query](#).

The [Odbc.DataSource](#) function provides a default navigation table with all databases, tables, and views from your system, supports query folding, and allows for a range of customization options. The majority of ODBC based extensions will use this as their primary extensibility function. The function accepts two arguments—a connection string, and an options record to provide behavior overrides.

The [Odbc.Query](#) function allows you to execute SQL statements through an ODBC driver. It acts as a passthrough for query execution. Unlike the [Odbc.DataSource](#) function, it doesn't provide query folding functionality, and requires that SQL queries be provided by the connector (or end user). When building a custom connector, this function is typically used internally to run queries to retrieve metadata that might not be exposed through regular ODBC channels. The function accepts two arguments—a connection string, and a SQL query.

Parameters for your Data Source Function

Custom connectors can accept any number of function arguments, but to remain consistent with the built-in data source functions shipped with Power Query, the following guidelines are recommended:

- Require the minimal set of parameters used to establish a connection to your server. The less parameters end users need to provide, the easier your connector will be to use.
- Although you can define parameters with a fixed number of values (that is, a dropdown list in the UI), parameters are entered before the user is authenticated. Any values that can be discovered

programmatically after the user is authenticated (such as catalog or database name) should be selectable through the Navigator. The default behavior for the [Odbc.DataSource](#) function will be to return a hierarchical navigation table consisting of Catalog (Database), Schema, and Table names, although this can be overridden within your connector.

- If you feel your users will typically know what values to enter for items they would select from the Navigator (such as the database name), make these parameters optional. Parameters that can be discovered programmatically should not be made required.
- The last parameter for your function should be an optional record called "options". This parameter typically allows advanced users to set common ODBC related properties (such as CommandTimeout), set behavior overrides specific to your connector, and allows for future extensibility without impacting backwards compatibility for your function.
- Security/credential related arguments MUST never be part of your data source function parameters, as values entered in the connect dialog will be persisted to the user's query. Credential related parameters should be specified as part of the connector's supported Authentication methods.

By default, all required parameters for your data source function are factored into the Data Source Path value used to identify user credentials.

Note that while the UI for the built-in [Odbc.DataSource](#) function provides a dropdown that allows the user to select a DSN, this functionality is not available through extensibility. If your data source configuration is complex enough to require a fully customizable configuration dialog, it's recommended you require your end users to pre-configure a system DSN, and have your function take in the DSN name as a text field.

Parameters for Odbc.DataSource

The [Odbc.DataSource](#) function takes two parameters—a connectionString for your driver, and an options record that lets you override various driver behaviors. Through the options record you can override capabilities and other information reported by the driver, control the navigator behavior, and affect the SQL queries generated by the M engine.

The supported options records fields fall into two categories—those that are public / always available, and those that are only available in an extensibility context.

The following table describes the public fields in the options record.

FIELD	DESCRIPTION
CommandTimeout	A duration value that controls how long the server-side query is allowed to run before it's cancelled. Default: 10 minutes
ConnectionTimeout	A duration value that controls how long to wait before abandoning an attempt to make a connection to the server. Default: 15 seconds

FIELD	DESCRIPTION
CreateNavigationProperties	<p>A logical value that sets whether to generate navigation properties on the returned tables. Navigation properties are based on foreign key relationships reported by the driver, and show up as "virtual" columns that can be expanded in the query editor, creating the appropriate join.</p> <p>If calculating foreign key dependencies is an expensive operation for your driver, you may want to set this value to false.</p> <p>Default: true</p>
HierarchicalNavigation	<p>A logical value that sets whether to view the tables grouped by their schema names. When set to false, tables will be displayed in a flat list under each database.</p> <p>Default: false</p>
SqlCompatibleWindowsAuth	<p>A logical value that determines whether to produce a SQL Server compatible connection string when using Windows Authentication—Trusted_Connection=Yes.</p> <p>If your driver supports Windows Authentication, but requires additional or alternative settings in your connection string, you should set this value to false and use the CredentialConnectionString option record field described below.</p> <p>Default: true</p>

The following table describes the options record fields that are only available through extensibility. Fields that aren't simple literal values are described in subsequent sections.

FIELD	DESCRIPTION
AstVisitor	<p>A record containing one or more overrides to control SQL query generation. The most common usage of this field is to provide logic to generate a LIMIT/OFFSET clause for drivers that don't support TOP.</p> <p>Fields include:</p> <ul style="list-style-type: none"> • Constant • LimitClause <p>See the AstVisitor section for more information.</p>
ClientConnectionPooling	<p>A logical value that enables client-side connection pooling for the ODBC driver. Most drivers will want to set this value to true.</p> <p>Default: false</p>

FIELD	DESCRIPTION
CredentialConnectionString	<p>A text or record value used to specify credential related connection string properties.</p> <p>See the Credential section for more information.</p>
HideNativeQuery	<p>A logical value that controls whether your connector allows native SQL statements to be passed in by a query using the Value.NativeQuery() function.</p> <p>Note: this functionality is currently not exposed in the Power Query user experience. Users would need to manually edit their queries to take advantage of this capability.</p> <p>Default: false</p>
ImplicitTypeConversions	<p>A table value containing implicit type conversions supported by your driver or backend server. Values in this table are additive to the conversions reported by the driver itself.</p> <p>This field is typically used in conjunction with the SQLGetTypeInfo field when overriding data type information reported by the driver.</p> <p>See the ImplicitTypeConversions section for more information.</p>
OnError	<p>An error handling function that receives an errorRecord parameter of type record.</p> <p>Common uses of this function include handling SSL connection failures, providing a download link if your driver isn't found on the system, and reporting authentication errors.</p> <p>See the OnError section for more information.</p>
SoftNumbers	<p>Allows the M engine to select a compatible data type when conversion between two specific numeric types isn't declared as supported in the SQL_CONVERT_* capabilities.</p> <p>Default: false</p>
SqlCapabilities	<p>A record providing various overrides of driver capabilities, and a way to specify capabilities that aren't expressed through ODBC 3.8.</p> <p>See the SqlCapabilities section for more information.</p>
SQLColumns	<p>A function that allows you to modify column metadata returned by the SQLColumns function.</p> <p>See the SQLColumns section for more information.</p>

FIELD	DESCRIPTION
SQLGetFunctions	<p>A record that allows you to override values returned by calls to SQLGetFunctions.</p> <p>A common use of this field is to disable the use of parameter binding, or to specify that generated queries should use CAST rather than CONVERT.</p> <p>See the SQLGetFunctions section for more information.</p>
SQLGetInfo	<p>A record that allows you to override values returned by calls to SQLGetInfo.</p> <p>See the SQLGetInfo section for more information.</p>
SQLGetTypeInfo	<p>A table, or function that returns a table, that overrides the type information returned by SQLGetTypeInfo.</p> <p>When the value is set to a table, the value completely replaces the type information reported by the driver. SQLGetTypeInfo won't be called.</p> <p>When the value is set to a function, your function will receive the result of the original call to SQLGetTypeInfo, allowing you to modify the table.</p> <p>This field is typically used when there's a mismatch between data types reported by SQLGetTypeInfo and SQLColumns.</p> <p>See the SQLGetTypeInfo section for more information.</p>
SQLTables	<p>A function that allows you to modify the table metadata returned by a call to SQLTables.</p> <p>See the SQLTables section for more information.</p>
TolerateConcatOverflow	<p>Allows conversion of numeric and text types to larger types if an operation would cause the value to fall out of range of the original type.</p> <p>For example, when adding Int32.Max + Int32.Max, the engine will cast the result to Int64 when this setting is set to true. When adding a VARCHAR(4000) and a VARCHAR(4000) field on a system that supports a maximize VARCHAR size of 4000, the engine will cast the result into a CLOB type.</p> <p>Default: false</p>

FIELD	DESCRIPTION
UseEmbeddedDriver	<p>(internal use): A logical value that controls whether the ODBC driver should be loaded from a local directory (using new functionality defined in the ODBC 4.0 specification). This is generally only set by connectors created by Microsoft that ship with Power Query.</p> <p>When set to false, the system ODBC driver manager will be used to locate and load the driver.</p> <p>Most connectors should not need to set this field.</p> <p>Default: false</p>

Overriding AstVisitor

The AstVisitor field is set through the [Odbc.DataSource](#) options record. It's used to modify SQL statements generated for specific query scenarios.

NOTE

Drivers that support `LIMIT` and `OFFSET` clauses (rather than `TOP`) will want to provide a LimitClause override for AstVisitor.

Constant

Providing an override for this value has been deprecated and may be removed from future implementations.

LimitClause

This field is a function that receives two `Int64.Type` arguments (skip, take), and returns a record with two text fields (Text, Location).

```
LimitClause = (skip as nullable number, take as number) as record => ...
```

The skip parameter is the number of rows to skip (that is, the argument to `OFFSET`). If an offset is not specified, the skip value will be null. If your driver supports `LIMIT`, but does not support `OFFSET`, the LimitClause function should return an unimplemented error (...) when skip is greater than 0.

The take parameter is the number of rows to take (that is, the argument to `LIMIT`).

The `Text` field of the result contains the SQL text to add to the generated query.

The `Location` field specifies where to insert the clause. The following table describes supported values.

VALUE	DESCRIPTION	EXAMPLE
AfterQuerySpecification	<p>LIMIT clause is put at the end of the generated SQL.</p> <p>This is the most commonly supported LIMIT syntax.</p>	<pre>SELECT a, b, c FROM table WHERE a > 10 LIMIT 5</pre>

Value	Description	Example
BeforeQuerySpecification	LIMIT clause is put before the generated SQL statement.	LIMIT 5 ROWS SELECT a, b, c FROM table WHERE a > 10
AfterSelect	LIMIT goes after the SELECT statement, and after any modifiers (such as DISTINCT).	SELECT DISTINCT LIMIT 5 a, b, c FROM table WHERE a > 10
AfterSelectBeforeModifiers	LIMIT goes after the SELECT statement, but before any modifiers (such as DISTINCT).	SELECT LIMIT 5 DISTINCT a, b, c FROM table WHERE a > 10

The following code snippet provides a LimitClause implementation for a driver that expects a LIMIT clause, with an optional OFFSET, in the following format: `[OFFSET <offset> ROWS] LIMIT <row_count>`

```
LimitClause = (skip, take) =>
  let
    offset = if (skip > 0) then Text.Format("OFFSET #{0} ROWS", {skip}) else "",
    limit = if (take <> null) then Text.Format("LIMIT #{0}", {take}) else ""
  in
  [
    Text = Text.Format("#{0} #{1}", {offset, limit}),
    Location = "AfterQuerySpecification"
  ]
```

The following code snippet provides a LimitClause implementation for a driver that supports LIMIT, but not OFFSET. Format: `LIMIT <row_count>`.

```
LimitClause = (skip, take) =>
  if (skip > 0) then error "Skip/Offset not supported"
  else
  [
    Text = Text.Format("LIMIT #{0}", {take}),
    Location = "AfterQuerySpecification"
  ]
```

Overriding SqlCapabilities

Field	Details
FractionalSecondsScale	A number value ranging from 1 to 7 that indicates the number of decimal places supported for millisecond values. This value should be set by connectors that want to enable query folding over datetime values. Default: null

FIELD	DETAILS
PrepareStatements	<p>A logical value that indicates that statements should be prepared using SQLPrepare.</p> <p>Default: false</p>
SupportsTop	<p>A logical value that indicates the driver supports the TOP clause to limit the number of returned rows.</p> <p>Default: false</p>
StringLiteralEscapeCharacters	<p>A list of text values that specify the character(s) to use when escaping string literals and LIKE expressions.</p> <p>Ex. {"\r\n"}</p> <p>Default: null</p>
SupportsDerivedTable	<p>A logical value that indicates the driver supports derived tables (sub-selects).</p> <p>This value is assumed to be true for drivers that set their conformance level to SQL_SC_SQL92_FULL (reported by the driver or overridden with the <code>Sql92Conformance</code> setting (see below)). For all other conformance levels, this value defaults to false.</p> <p>If your driver doesn't report the SQL_SC_SQL92_FULL compliance level, but does support derived tables, set this value to true.</p> <p>Note that supporting derived tables is required for many Direct Query scenarios.</p>
SupportsNumericLiterals	<p>A logical value that indicates whether the generated SQL should include numeric literals values. When set to false, numeric values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsStringLiterals	<p>A logical value that indicates whether the generated SQL should include string literals values. When set to false, string values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsOdbcDateLiterals	<p>A logical value that indicates whether the generated SQL should include date literals values. When set to false, date values will always be specified using Parameter Binding.</p> <p>Default: false</p>

FIELD	DETAILS
SupportsOdbcTimeLiterals	A logical value that indicates whether the generated SQL should include time literals values. When set to false, time values will always be specified using Parameter Binding. Default: false
SupportsOdbcTimestampLiterals	A logical value that indicates whether the generated SQL should include timestamp literals values. When set to false, timestamp values will always be specified using Parameter Binding. Default: false

Overriding SQLColumns

`SQLColumns` is a function handler that receives the results of an ODBC call to [SQLColumns](#). The source parameter contains a table with the data type information. This override is typically used to fix up data type mismatches between calls to `SQLGetTypeInfo` and `SQLColumns`.

For details of the format of the source table parameter, see:

<https://docs.microsoft.com/sql/odbc/reference/syntax/sqlcolumns-function>

Overriding SQLGetFunctions

This field is used to override SQLFunctions values returned by an ODBC driver. It contains a record whose field names are equal to the FunctionId constants defined for the ODBC [SQLGetFunctions](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#).

FIELD	DETAILS
SQL_CONVERT_FUNCTIONS	Indicates which function(s) are supported when doing type conversions. By default, the M Engine will attempt to use the CONVERT function. Drivers that prefer the use of CAST can override this value to report that only SQL_FN_CVT_CAST (numeric value of 0x2) is supported.
SQL_API_SQLBINDCOL	A logical (true/false) value that indicates whether the Mashup Engine should use the SQLBindCol API when retrieving data. When set to false, SQLGetData is used instead. Default: false

The following code snippet provides an example explicitly telling the M engine to use CAST rather than CONVERT.

```
SQLGetFunctions = [
    SQL_CONVERT_FUNCTIONS = 0x2 /* SQL_FN_CVT_CAST */
]
```

Overriding SQLGetInfo

This field is used to override SQLGetInfo values returned by an ODBC driver. It contains a record whose fields are names equal to the InfoType constants defined for the ODBC [SQLGetInfo](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#). The full list of InfoTypes that are checked can be found in the Mashup Engine trace files.

The following table contains commonly overridden SQLGetInfo properties:

FIELD	DETAILS
SQL_SQL_CONFORMANCE	An integer value that indicates the level of SQL-92 supported by the driver: (1) SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant. (2) SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant. (4) SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant. (8) SQL_SC_SQL92_FULL = Full level SQL-92 compliant. Note that in Power Query scenarios, the connector will be used in a Read Only mode. Most drivers will want to report a SQL_SC_SQL92_FULL compliance level, and override specific SQL generation behavior using the SQLGetInfo and SQLGetFunctions properties.
SQL_SQL92_PREDICATES	A bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92. See the SQL_SP_* constants in the ODBC specification.
SQL_AGGREGATE_FUNCTIONS	A bitmask enumerating support for aggregation functions. SQL_AF_ALL SQL_AF_AVG SQL_AF_COUNT SQL_AF_DISTINCT SQL_AF_MAX SQL_AF_MIN SQL_AF_SUM See the SQL_AF_* constants in the ODBC specification.

FIELD	DETAILS
SQL_GROUP_BY	<p>A integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the select list:</p> <p>SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column.</p> <p>SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.</p> <p>SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.</p> <p>SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is data source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>See the SQL_GB_* constants in the ODBC specification.</p>

The following helper function can be used to create bitmask values from a list of integer values:

```
Flags = (flags as list) =>
let
    Loop = List.Generate(
        ()=> [i = 0, Combined = 0],
        each [i] < List.Count(flags),
        each [i = [i]+1, Combined =*Number.BitwiseOr([Combined], flags{i})],
        each [Combined]),
    Result = List.Last(Loop, 0)
in
    Result;
```

Overriding SQLGetTypeInfo

`SQLGetTypeInfo` can be specified in two ways:

- A fixed `table` value that contains the same type information as an ODBC call to `SQLGetTypeInfo`.
- A function that accepts a table argument, and returns a table. The argument will contain the original results of the ODBC call to `SQLGetTypeInfo`. Your function implementation can modify/add to this table.

The first approach is used to completely override the values returned by the ODBC driver. The second approach is used if you want to add to or modify these values.

For details of the format of the types table parameter and expected return value, see the [SQLGetTypeInfo function reference](#).

SQLGetTypeInfo using a static table

The following code snippet provides a static implementation for `SQLGetTypeInfo`.

```
SQLGetTypeInfo = #table{
```

```

    { "TYPE_NAME",      "DATA_TYPE", "COLUMN_SIZE", "LITERAL_PREF", "LITERAL_SUFFIX", "CREATE_PARAS",
"NULLABLE", "CASE_SENSITIVE", "SEARCHABLE", "UNSIGNED_ATTRIBUTE", "FIXED_PREC_SCALE", "AUTO_UNIQUE_VALUE",
"LOCAL_TYPE_NAME", "MINIMUM_SCALE", "MAXIMUM_SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "NUM_PREC_RADIX",
"INTERNAL_PRECISION", "USER_DATA_TYPE" }, {
    { "char",           1,          65535,      "",      "",      "max. length",
1,           1,          3,          null,      0,          null,
"char",        null,        null,      -8,          null,      null,
0,           },      1,          19,          0,          10,          0,
    { "int8",          -5,          2,          0,          -5,          2,
1,           0,          0,          0,          null,          2,
"int8",        0,          0,          0,          null,          0,
0,           },      1,          -7,          1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          -7,          1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          9,          10,          "",          null,
1,           0,          2,          null,      0,          null,
"date",        null,        null,      9,          1,          null,
0,           },      1,          3,          28,          null,      null,
1,           0,          2,          0,          0,          0,
"numeric",      0,          0,          2,          null,          10,
0,           },      1,          8,          15,          null,      null,
1,           0,          2,          0,          0,          0,
"float8",        null,        null,      6,          null,          2,
0,           },      1,          6,          17,          null,      null,
1,           0,          2,          0,          0,          0,
"float8",        null,        null,      6,          null,          2,
0,           },      1,          -11,          2,          37,          null,
1,           0,          2,          null,      0,          null,
"uuid",        null,        null,      -11,          null,      null,
0,           },      1,          4,          10,          null,      null,
1,           0,          2,          0,          0,          0,
"int4",        0,          0,          4,          null,          2,
0,           },      1,          -1,          65535,      "",      null,
1,           1,          3,          null,      0,          null,
"text",        null,        null,      -10,          null,      null,
0,           },      1,          -4,          255,      "",      null,
1,           0,          2,          null,      0,          null,
"lo",        null,        null,      -4,          null,      null,
0,           },      1,          2,          28,          null,      null,
1,           0,          2,          0,          10,          0,
"numeric",      0,          6,          2,          null,          10,
0,           },      1,          7,          9,          null,      null,
1,           0,          2,          0,          10,          0,
"float4",        null,        null,      7,          null,          2,
0,           },      1,          5,          19,          null,      null,
1,           0,          2,          0,          10,          0,
"int2",        0,          0,          5,          null,          2,
0,           },      1,          -6,          5,          null,      null,
1,           0,          2,          0,          10,          0,
"int2",        0,          0,          5,          null,          2,
0,           }
}

```

```

        },
        { "timestamp",      11,      26,      "",      "",      null,
1,          0,           2,       null,      9,      0,      null,
"timestamp",      0,           38,       null,      9,      3,      null,
0,           },      91,      10,      "",      "",      null,
1,          0,           2,       null,      9,      1,      null,
"date",      null,       null,      9,      1,      null,
0,           },      93,      26,      "",      "",      null,
1,          0,           2,       null,      9,      3,      null,
"timestamp",      0,           38,       null,      9,      0,      null,
0,           },      -3,      255,      "",      "",      null,
1,          0,           2,       null,      -3,      0,      null,
"bytea",      null,       null,      -3,      null,      null,
0,           },      12,      65535,      "",      "",      "max. length",
1,          0,           2,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -8,      0,      null,
"char",      null,       null,      -8,      null,      null,
0,           },      -10,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -10,      0,      null,
"text",      null,       null,      -10,      null,      null,
0,           },      -9,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"bpchar",      null,       null,      -9,      null,      null,
0,           } } }
);

```

SQLGetTypeInfo using a function

The following code snippets append the `bpchar` type to the existing types returned by the driver.

```

SQLGetTypeInfo = (types as table) as table =>
let
    newTypes = #table(
    {
        "TYPE_NAME",
        "DATA_TYPE",
        "COLUMN_SIZE",
        "LITERAL_PREF",
        "LITERAL_SUFFIX",
        "CREATE_PARAS",
        "NULLABLE",
        "CASE_SENSITIVE",
        "SEARCHABLE",
        "UNSIGNED_ATTRIBUTE",
        "FIXED_PREC_SCALE",
        "AUTO_UNIQUE_VALUE",
        "LOCAL_TYPE_NAME",
        "MINIMUM_SCALE",
        "MAXIMUM_SCALE",
        "SQL_DATA_TYPE",
        "SQL_DATETIME_SUB",
        "NUM_PREC_RADIX",
        "INTERNAL_PRECISION",
        "USER_DATA_TYPE"
    },
    // we add a new entry for each type we want to add
    {
        {
            "bpchar",
            -8,
            65535,
            """",
            """",
            "max. length",
            1,
            1,
            3,
            null,
            0,
            null,
            "bpchar",
            null,
            null,
            -9,
            null,
            null,
            0,
            0
        }
    )),
    append = Table.Combine({types, newTypes})
in
    append;

```

Setting the Connection String

The connection string for your ODBC driver is set using the first argument to the [Odbc.DataSource](#) and/or [Odbc.Query](#) functions. The value can be text, or an M record. When using the record, each field in the record will become a property in the connection string. All connection strings will require a Driver field (or DSN field if you require users to pre-configure a system level DSN). Credential related properties will be set separately (see below). Other properties will be driver specific.

The code snippet below shows the definition of a new data source function, creation of the `ConnectionString` record, and invocation of the [Odbc.DataSource](#) function.

```
[DataSource.Kind="SqlODBC", Publish="SqlODBC.Publish"]
shared SqlODBC.Contents = (server as text) =>
let
    ConnectionString = [
        Driver = "SQL Server Native Client 11.0",
        Server = server,
        MultiSubnetFailover = "Yes",
        ApplicationIntent = "ReadOnly",
        APP = "PowerBICustomConnector"
    ],
    OdbcDatasource = Odbc.DataSource(ConnectionString)
in
    OdbcDatasource;
```

Troubleshooting and Testing

To enable tracing in Power BI Desktop:

1. Go to **File > Options and settings > Options**.
2. Select on the **Diagnostics** tab.
3. Select the **Enable tracing** option.
4. Select the **Open traces folder** link (should be `%LOCALAPPDATA%/Microsoft/Power BI Desktop/Traces`).
5. Delete existing trace files.
6. Perform your tests.
7. Close Power BI Desktop to ensure all log files are flushed to disk.

Here are steps you can take for initial testing in Power BI Desktop:

1. Close Power BI Desktop.
2. Clear your trace directory.
3. Open Power BI desktop, and enable tracing.
4. Connect to your data source, and select Direct Query mode.
5. Select a table in the navigator, and select **Edit**.
6. Manipulate the query in various ways, including:

- Take the First N rows (for example, 10).
- Set equality filters on different data types (int, string, bool, and so on).
- Set other range filters (greater than, less than).
- Filter on NULL / NOT NULL.
- Select a sub-set of columns.
- Aggregate / Group By different column combinations.
- Add a column calculated from other columns (`[C] = [A] + [B]`).
- Sort on one column, multiple columns. 7. Expressions that fail to fold will result in a warning bar. Note the failure, remove the step, and move to the next test case. Details about the cause of the failure should be emitted to the trace logs. 8. Close Power BI Desktop. 9. Copy the trace files to a new directory. 10. Use the recommend Power BI workbook to parse and analyze the trace files.

Once you have simple queries working, you can then try Direct Query scenarios (for example, building reports in the Report Views). The queries generated in Direct Query mode will be significantly more complex (that is, use of sub-selects, COALESCE statements, and aggregations).

Concatenation of strings in Direct Query mode

The M engine does basic type size limit validation as part of its query folding logic. If you are receiving a folding

error when trying to concatenate two strings that potentially overflow the maximum size of the underlying database type:

1. Ensure that your database can support up-conversion to CLOB types when string concat overflow occurs.
2. Set the `TolerateConcatOverflow` option for Odbc.DataSource to `true`.

The [DAX CONCATENATE function](#) is currently not supported by Power Query/ODBC extensions. Extension authors should ensure string concatenation works through the query editor by adding calculated columns (`[stringCol1] & [stringCol2]`). When the capability to fold the CONCATENATE operation is added in the future, it should work seamlessly with existing extensions.

Enabling Direct Query for an ODBC based connector

21 minutes to read • [Edit Online](#)

Overview

Using M's built-in [Odbc.DataSource](#) function is the recommended way to create custom connectors for data sources that have an existing ODBC driver and/or support a SQL query syntax. Wrapping the [Odbc.DataSource](#) function will allow your connector to inherit default query folding behavior based on the capabilities reported by your driver. This will enable the M engine to generate SQL statements based on filters and other transformations defined by the user within the Power Query experience, without having to provide this logic within the connector itself.

ODBC extensions can optionally enable Direct Query mode, allowing Power BI to dynamically generate queries at runtime without pre-caching the user's data model.

NOTE

Enabling Direct Query support raises the difficulty and complexity level of your connector. When Direct Query is enabled, Power BI will prevent the M engine from compensating for operations that cannot be fully pushed to the underlying data source.

This section builds on the concepts presented in the M Extensibility Reference, and assumes familiarity with the creation of a basic Data Connector.

Refer to the [SqlODBC sample](#) for most of the code examples in the sections below. Additional samples can be found in the ODBC samples directory.

ODBC Extensibility Functions

The M engine provides two ODBC related data source functions: [Odbc.DataSource](#), and [Odbc.Query](#).

The [Odbc.DataSource](#) function provides a default navigation table with all databases, tables, and views from your system, supports query folding, and allows for a range of customization options. The majority of ODBC based extensions will use this as their primary extensibility function. The function accepts two arguments—a connection string, and an options record to provide behavior overrides.

The [Odbc.Query](#) function allows you to execute SQL statements through an ODBC driver. It acts as a passthrough for query execution. Unlike the [Odbc.DataSource](#) function, it doesn't provide query folding functionality, and requires that SQL queries be provided by the connector (or end user). When building a custom connector, this function is typically used internally to run queries to retrieve metadata that might not be exposed through regular ODBC channels. The function accepts two arguments—a connection string, and a SQL query.

Parameters for your Data Source Function

Custom connectors can accept any number of function arguments, but to remain consistent with the built-in data source functions shipped with Power Query, the following guidelines are recommended:

- Require the minimal set of parameters used to establish a connection to your server. The less parameters end users need to provide, the easier your connector will be to use.
- Although you can define parameters with a fixed number of values (that is, a dropdown list in the UI), parameters are entered before the user is authenticated. Any values that can be discovered

programmatically after the user is authenticated (such as catalog or database name) should be selectable through the Navigator. The default behavior for the [Odbc.DataSource](#) function will be to return a hierarchical navigation table consisting of Catalog (Database), Schema, and Table names, although this can be overridden within your connector.

- If you feel your users will typically know what values to enter for items they would select from the Navigator (such as the database name), make these parameters optional. Parameters that can be discovered programmatically should not be made required.
- The last parameter for your function should be an optional record called "options". This parameter typically allows advanced users to set common ODBC related properties (such as CommandTimeout), set behavior overrides specific to your connector, and allows for future extensibility without impacting backwards compatibility for your function.
- Security/credential related arguments MUST never be part of your data source function parameters, as values entered in the connect dialog will be persisted to the user's query. Credential related parameters should be specified as part of the connector's supported Authentication methods.

By default, all required parameters for your data source function are factored into the Data Source Path value used to identify user credentials.

Note that while the UI for the built-in [Odbc.DataSource](#) function provides a dropdown that allows the user to select a DSN, this functionality is not available through extensibility. If your data source configuration is complex enough to require a fully customizable configuration dialog, it's recommended you require your end users to pre-configure a system DSN, and have your function take in the DSN name as a text field.

Parameters for Odbc.DataSource

The [Odbc.DataSource](#) function takes two parameters—a connectionString for your driver, and an options record that lets you override various driver behaviors. Through the options record you can override capabilities and other information reported by the driver, control the navigator behavior, and affect the SQL queries generated by the M engine.

The supported options records fields fall into two categories—those that are public / always available, and those that are only available in an extensibility context.

The following table describes the public fields in the options record.

FIELD	DESCRIPTION
CommandTimeout	A duration value that controls how long the server-side query is allowed to run before it's cancelled. Default: 10 minutes
ConnectionTimeout	A duration value that controls how long to wait before abandoning an attempt to make a connection to the server. Default: 15 seconds

FIELD	DESCRIPTION
CreateNavigationProperties	<p>A logical value that sets whether to generate navigation properties on the returned tables. Navigation properties are based on foreign key relationships reported by the driver, and show up as "virtual" columns that can be expanded in the query editor, creating the appropriate join.</p> <p>If calculating foreign key dependencies is an expensive operation for your driver, you may want to set this value to false.</p> <p>Default: true</p>
HierarchicalNavigation	<p>A logical value that sets whether to view the tables grouped by their schema names. When set to false, tables will be displayed in a flat list under each database.</p> <p>Default: false</p>
SqlCompatibleWindowsAuth	<p>A logical value that determines whether to produce a SQL Server compatible connection string when using Windows Authentication—Trusted_Connection=Yes.</p> <p>If your driver supports Windows Authentication, but requires additional or alternative settings in your connection string, you should set this value to false and use the CredentialConnectionString option record field described below.</p> <p>Default: true</p>

The following table describes the options record fields that are only available through extensibility. Fields that aren't simple literal values are described in subsequent sections.

FIELD	DESCRIPTION
AstVisitor	<p>A record containing one or more overrides to control SQL query generation. The most common usage of this field is to provide logic to generate a LIMIT/OFFSET clause for drivers that don't support TOP.</p> <p>Fields include:</p> <ul style="list-style-type: none"> • Constant • LimitClause <p>See the AstVisitor section for more information.</p>
ClientConnectionPooling	<p>A logical value that enables client-side connection pooling for the ODBC driver. Most drivers will want to set this value to true.</p> <p>Default: false</p>

FIELD	DESCRIPTION
CredentialConnectionString	<p>A text or record value used to specify credential related connection string properties.</p> <p>See the Credential section for more information.</p>
HideNativeQuery	<p>A logical value that controls whether your connector allows native SQL statements to be passed in by a query using the Value.NativeQuery() function.</p> <p>Note: this functionality is currently not exposed in the Power Query user experience. Users would need to manually edit their queries to take advantage of this capability.</p> <p>Default: false</p>
ImplicitTypeConversions	<p>A table value containing implicit type conversions supported by your driver or backend server. Values in this table are additive to the conversions reported by the driver itself.</p> <p>This field is typically used in conjunction with the SQLGetTypeInfo field when overriding data type information reported by the driver.</p> <p>See the ImplicitTypeConversions section for more information.</p>
OnError	<p>An error handling function that receives an errorRecord parameter of type record.</p> <p>Common uses of this function include handling SSL connection failures, providing a download link if your driver isn't found on the system, and reporting authentication errors.</p> <p>See the OnError section for more information.</p>
SoftNumbers	<p>Allows the M engine to select a compatible data type when conversion between two specific numeric types isn't declared as supported in the SQL_CONVERT_* capabilities.</p> <p>Default: false</p>
SqlCapabilities	<p>A record providing various overrides of driver capabilities, and a way to specify capabilities that aren't expressed through ODBC 3.8.</p> <p>See the SqlCapabilities section for more information.</p>
SQLColumns	<p>A function that allows you to modify column metadata returned by the SQLColumns function.</p> <p>See the SQLColumns section for more information.</p>

FIELD	DESCRIPTION
SQLGetFunctions	<p>A record that allows you to override values returned by calls to SQLGetFunctions.</p> <p>A common use of this field is to disable the use of parameter binding, or to specify that generated queries should use CAST rather than CONVERT.</p> <p>See the SQLGetFunctions section for more information.</p>
SQLGetInfo	<p>A record that allows you to override values returned by calls to SQLGetInfo.</p> <p>See the SQLGetInfo section for more information.</p>
SQLGetTypeInfo	<p>A table, or function that returns a table, that overrides the type information returned by SQLGetTypeInfo.</p> <p>When the value is set to a table, the value completely replaces the type information reported by the driver. SQLGetTypeInfo won't be called.</p> <p>When the value is set to a function, your function will receive the result of the original call to SQLGetTypeInfo, allowing you to modify the table.</p> <p>This field is typically used when there's a mismatch between data types reported by SQLGetTypeInfo and SQLColumns.</p> <p>See the SQLGetTypeInfo section for more information.</p>
SQLTables	<p>A function that allows you to modify the table metadata returned by a call to SQLTables.</p> <p>See the SQLTables section for more information.</p>
TolerateConcatOverflow	<p>Allows conversion of numeric and text types to larger types if an operation would cause the value to fall out of range of the original type.</p> <p>For example, when adding Int32.Max + Int32.Max, the engine will cast the result to Int64 when this setting is set to true. When adding a VARCHAR(4000) and a VARCHAR(4000) field on a system that supports a maximize VARCHAR size of 4000, the engine will cast the result into a CLOB type.</p> <p>Default: false</p>

FIELD	DESCRIPTION
UseEmbeddedDriver	<p>(internal use): A logical value that controls whether the ODBC driver should be loaded from a local directory (using new functionality defined in the ODBC 4.0 specification). This is generally only set by connectors created by Microsoft that ship with Power Query.</p> <p>When set to false, the system ODBC driver manager will be used to locate and load the driver.</p> <p>Most connectors should not need to set this field.</p> <p>Default: false</p>

Overriding AstVisitor

The AstVisitor field is set through the [Odbc.DataSource](#) options record. It's used to modify SQL statements generated for specific query scenarios.

NOTE

Drivers that support `LIMIT` and `OFFSET` clauses (rather than `TOP`) will want to provide a LimitClause override for AstVisitor.

Constant

Providing an override for this value has been deprecated and may be removed from future implementations.

LimitClause

This field is a function that receives two `Int64.Type` arguments (skip, take), and returns a record with two text fields (Text, Location).

```
LimitClause = (skip as nullable number, take as number) as record => ...
```

The skip parameter is the number of rows to skip (that is, the argument to `OFFSET`). If an offset is not specified, the skip value will be null. If your driver supports `LIMIT`, but does not support `OFFSET`, the LimitClause function should return an unimplemented error (...) when skip is greater than 0.

The take parameter is the number of rows to take (that is, the argument to `LIMIT`).

The `Text` field of the result contains the SQL text to add to the generated query.

The `Location` field specifies where to insert the clause. The following table describes supported values.

VALUE	DESCRIPTION	EXAMPLE
AfterQuerySpecification	<p>LIMIT clause is put at the end of the generated SQL.</p> <p>This is the most commonly supported LIMIT syntax.</p>	<pre>SELECT a, b, c FROM table WHERE a > 10 LIMIT 5</pre>

Value	Description	Example
BeforeQuerySpecification	LIMIT clause is put before the generated SQL statement.	LIMIT 5 ROWS SELECT a, b, c FROM table WHERE a > 10
AfterSelect	LIMIT goes after the SELECT statement, and after any modifiers (such as DISTINCT).	SELECT DISTINCT LIMIT 5 a, b, c FROM table WHERE a > 10
AfterSelectBeforeModifiers	LIMIT goes after the SELECT statement, but before any modifiers (such as DISTINCT).	SELECT LIMIT 5 DISTINCT a, b, c FROM table WHERE a > 10

The following code snippet provides a LimitClause implementation for a driver that expects a LIMIT clause, with an optional OFFSET, in the following format: `[OFFSET <offset> ROWS] LIMIT <row_count>`

```
LimitClause = (skip, take) =>
  let
    offset = if (skip > 0) then Text.Format("OFFSET #{0} ROWS", {skip}) else "",
    limit = if (take <> null) then Text.Format("LIMIT #{0}", {take}) else ""
  in
  [
    Text = Text.Format("#{0} #{1}", {offset, limit}),
    Location = "AfterQuerySpecification"
  ]
```

The following code snippet provides a LimitClause implementation for a driver that supports LIMIT, but not OFFSET. Format: `LIMIT <row_count>`.

```
LimitClause = (skip, take) =>
  if (skip > 0) then error "Skip/Offset not supported"
  else
  [
    Text = Text.Format("LIMIT #{0}", {take}),
    Location = "AfterQuerySpecification"
  ]
```

Overriding SqlCapabilities

Field	Details
FractionalSecondsScale	A number value ranging from 1 to 7 that indicates the number of decimal places supported for millisecond values. This value should be set by connectors that want to enable query folding over datetime values. Default: null

FIELD	DETAILS
PrepareStatements	<p>A logical value that indicates that statements should be prepared using SQLPrepare.</p> <p>Default: false</p>
SupportsTop	<p>A logical value that indicates the driver supports the TOP clause to limit the number of returned rows.</p> <p>Default: false</p>
StringLiteralEscapeCharacters	<p>A list of text values that specify the character(s) to use when escaping string literals and LIKE expressions.</p> <p>Ex. {"\r\n"}</p> <p>Default: null</p>
SupportsDerivedTable	<p>A logical value that indicates the driver supports derived tables (sub-selects).</p> <p>This value is assumed to be true for drivers that set their conformance level to SQL_SC_SQL92_FULL (reported by the driver or overridden with the <code>Sql92Conformance</code> setting (see below)). For all other conformance levels, this value defaults to false.</p> <p>If your driver doesn't report the SQL_SC_SQL92_FULL compliance level, but does support derived tables, set this value to true.</p> <p>Note that supporting derived tables is required for many Direct Query scenarios.</p>
SupportsNumericLiterals	<p>A logical value that indicates whether the generated SQL should include numeric literals values. When set to false, numeric values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsStringLiterals	<p>A logical value that indicates whether the generated SQL should include string literals values. When set to false, string values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsOdbcDateLiterals	<p>A logical value that indicates whether the generated SQL should include date literals values. When set to false, date values will always be specified using Parameter Binding.</p> <p>Default: false</p>

FIELD	DETAILS
SupportsOdbcTimeLiterals	A logical value that indicates whether the generated SQL should include time literals values. When set to false, time values will always be specified using Parameter Binding. Default: false
SupportsOdbcTimestampLiterals	A logical value that indicates whether the generated SQL should include timestamp literals values. When set to false, timestamp values will always be specified using Parameter Binding. Default: false

Overriding SQLColumns

`SQLColumns` is a function handler that receives the results of an ODBC call to [SQLColumns](#). The source parameter contains a table with the data type information. This override is typically used to fix up data type mismatches between calls to `SQLGetTypeInfo` and `SQLColumns`.

For details of the format of the source table parameter, see:

<https://docs.microsoft.com/sql/odbc/reference/syntax/sqlcolumns-function>

Overriding SQLGetFunctions

This field is used to override SQLFunctions values returned by an ODBC driver. It contains a record whose field names are equal to the FunctionId constants defined for the ODBC [SQLGetFunctions](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#).

FIELD	DETAILS
SQL_CONVERT_FUNCTIONS	Indicates which function(s) are supported when doing type conversions. By default, the M Engine will attempt to use the CONVERT function. Drivers that prefer the use of CAST can override this value to report that only SQL_FN_CVT_CAST (numeric value of 0x2) is supported.
SQL_API_SQLBINDCOL	A logical (true/false) value that indicates whether the Mashup Engine should use the SQLBindCol API when retrieving data. When set to false, SQLGetData is used instead. Default: false

The following code snippet provides an example explicitly telling the M engine to use CAST rather than CONVERT.

```
SQLGetFunctions = [
    SQL_CONVERT_FUNCTIONS = 0x2 /* SQL_FN_CVT_CAST */
]
```

Overriding SQLGetInfo

This field is used to override SQLGetInfo values returned by an ODBC driver. It contains a record whose fields are names equal to the InfoType constants defined for the ODBC [SQLGetInfo](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#). The full list of InfoTypes that are checked can be found in the Mashup Engine trace files.

The following table contains commonly overridden SQLGetInfo properties:

FIELD	DETAILS
SQL_SQL_CONFORMANCE	An integer value that indicates the level of SQL-92 supported by the driver: (1) SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant. (2) SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant. (4) SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant. (8) SQL_SC_SQL92_FULL = Full level SQL-92 compliant. Note that in Power Query scenarios, the connector will be used in a Read Only mode. Most drivers will want to report a SQL_SC_SQL92_FULL compliance level, and override specific SQL generation behavior using the SQLGetInfo and SQLGetFunctions properties.
SQL_SQL92_PREDICATES	A bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92. See the SQL_SP_* constants in the ODBC specification.
SQL_AGGREGATE_FUNCTIONS	A bitmask enumerating support for aggregation functions. SQL_AF_ALL SQL_AF_AVG SQL_AF_COUNT SQL_AF_DISTINCT SQL_AF_MAX SQL_AF_MIN SQL_AF_SUM See the SQL_AF_* constants in the ODBC specification.

FIELD	DETAILS
SQL_GROUP_BY	<p>A integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the select list:</p> <p>SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column.</p> <p>SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.</p> <p>SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.</p> <p>SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is data source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>See the SQL_GB_* constants in the ODBC specification.</p>

The following helper function can be used to create bitmask values from a list of integer values:

```
Flags = (flags as list) =>
let
    Loop = List.Generate(
        ()=> [i = 0, Combined = 0],
        each [i] < List.Count(flags),
        each [i = [i]+1, Combined =*Number.BitwiseOr([Combined], flags{i})],
        each [Combined]),
    Result = List.Last(Loop, 0)
in
    Result;
```

Overriding SQLGetTypeInfo

`SQLGetTypeInfo` can be specified in two ways:

- A fixed `table` value that contains the same type information as an ODBC call to `SQLGetTypeInfo`.
- A function that accepts a table argument, and returns a table. The argument will contain the original results of the ODBC call to `SQLGetTypeInfo`. Your function implementation can modify/add to this table.

The first approach is used to completely override the values returned by the ODBC driver. The second approach is used if you want to add to or modify these values.

For details of the format of the types table parameter and expected return value, see the [SQLGetTypeInfo function reference](#).

SQLGetTypeInfo using a static table

The following code snippet provides a static implementation for `SQLGetTypeInfo`.

```
SQLGetTypeInfo = #table{
```

```

    { "TYPE_NAME",      "DATA_TYPE", "COLUMN_SIZE", "LITERAL_PREF", "LITERAL_SUFFIX", "CREATE_PARAS",
"NULLABLE", "CASE_SENSITIVE", "SEARCHABLE", "UNSIGNED_ATTRIBUTE", "FIXED_PREC_SCALE", "AUTO_UNIQUE_VALUE",
"LOCAL_TYPE_NAME", "MINIMUM_SCALE", "MAXIMUM_SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "NUM_PREC_RADIX",
"INTERNAL_PRECISION", "USER_DATA_TYPE" }, {
    { "char",           1,          65535,      "",      "",      "max. length",
1,           1,          3,          null,      0,          null,
"char",        null,        null,      -8,          null,      null,
0,           },      1,          19,         0,          10,         0,
    { "int8",          -5,          2,          0,          -5,          2,
1,           0,          0,          0,          null,      2,
"int8",        0,          0,          0,          null,      0,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          9,          10,         "",         null,
1,           0,          2,          null,      0,          null,
"date",        null,        null,      9,          1,          null,
0,           },      1,          3,          28,         null,      null,
1,           0,          2,          0,          0,          0,
"numeric",      0,          0,          2,          null,      10,
0,           },      1,          8,          15,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          6,          17,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          -11,         2,         37,         null,
1,           0,          2,         null,      0,          null,
"uuid",        null,        null,      -11,         null,      null,
0,           },      1,          4,          10,         null,      null,
1,           0,          2,          0,          4,          0,
"int4",        0,          0,          4,          null,      2,
0,           },      1,          -1,         65535,      "",      null,
1,           1,          3,          null,      0,          null,
"text",        null,        null,      -10,         null,      null,
0,           },      1,          -4,         255,         "",         null,
1,           0,          2,          null,      0,          null,
"lo",        null,        null,      -4,          null,      null,
0,           },      1,          2,          28,         null,      null,
1,           0,          2,          0,          10,         0,
"numeric",      0,          6,          2,          null,      10,
0,           },      1,          7,          9,          null,      null,
1,           0,          2,          0,          10,         0,
"float4",      null,        null,      7,          null,      2,
0,           },      1,          5,          19,         null,      null,
1,           0,          2,          0,          5,          null,
"int2",        0,          0,          5,          null,      2,
0,           },      1,          -6,         2,          5,          null,
1,           0,          0,          0,          10,         0,
"int2",        0,          0,          5,          null,      2,
0,           }

```

```

        },
        { "timestamp",      11,      26,      "",      "",      null,
1,          0,           2,       null,      9,      0,      null,
"timestamp",      0,           38,       null,      9,      3,      null,
0,           },      91,      10,      "",      "",      null,
1,          0,           2,       null,      9,      1,      null,
"date",      null,       null,      9,      1,      null,
0,           },      93,      26,      "",      "",      null,
1,          0,           2,       null,      9,      3,      null,
"timestamp",      0,           38,       null,      9,      0,      null,
0,           },      -3,      255,      "",      "",      null,
1,          0,           2,       null,      -3,      0,      null,
"bytea",      null,       null,      -3,      null,      null,
0,           },      12,      65535,      "",      "",      "max. length",
1,          0,           2,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -8,      0,      null,
"char",      null,       null,      -8,      null,      null,
0,           },      -10,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -10,      0,      null,
"text",      null,       null,      -10,      null,      null,
0,           },      -9,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,           },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"bpchar",      null,       null,      -9,      null,      null,
0,           } } }
);

```

SQLGetTypeInfo using a function

The following code snippets append the `bpchar` type to the existing types returned by the driver.

```

SQLGetTypeInfo = (types as table) as table =>
let
    newTypes = #table(
    {
        "TYPE_NAME",
        "DATA_TYPE",
        "COLUMN_SIZE",
        "LITERAL_PREF",
        "LITERAL_SUFFIX",
        "CREATE_PARAS",
        "NULLABLE",
        "CASE_SENSITIVE",
        "SEARCHABLE",
        "UNSIGNED_ATTRIBUTE",
        "FIXED_PREC_SCALE",
        "AUTO_UNIQUE_VALUE",
        "LOCAL_TYPE_NAME",
        "MINIMUM_SCALE",
        "MAXIMUM_SCALE",
        "SQL_DATA_TYPE",
        "SQL_DATETIME_SUB",
        "NUM_PREC_RADIX",
        "INTERNAL_PRECISION",
        "USER_DATA_TYPE"
    },
    // we add a new entry for each type we want to add
    {
        {
            "bpchar",
            -8,
            65535,
            """",
            """",
            "max. length",
            1,
            1,
            3,
            null,
            0,
            null,
            "bpchar",
            null,
            null,
            -9,
            null,
            null,
            0,
            0
        }
    )),
    append = Table.Combine({types, newTypes})
in
    append;

```

Setting the Connection String

The connection string for your ODBC driver is set using the first argument to the [Odbc.DataSource](#) and/or [Odbc.Query](#) functions. The value can be text, or an M record. When using the record, each field in the record will become a property in the connection string. All connection strings will require a Driver field (or DSN field if you require users to pre-configure a system level DSN). Credential related properties will be set separately (see below). Other properties will be driver specific.

The code snippet below shows the definition of a new data source function, creation of the `ConnectionString` record, and invocation of the [Odbc.DataSource](#) function.

```
[DataSource.Kind="SqlODBC", Publish="SqlODBC.Publish"]
shared SqlODBC.Contents = (server as text) =>
let
    ConnectionString = [
        Driver = "SQL Server Native Client 11.0",
        Server = server,
        MultiSubnetFailover = "Yes",
        ApplicationIntent = "ReadOnly",
        APP = "PowerBICustomConnector"
    ],
    OdbcDatasource = Odbc.DataSource(ConnectionString)
in
    OdbcDatasource;
```

Troubleshooting and Testing

To enable tracing in Power BI Desktop:

1. Go to **File > Options and settings > Options**.
2. Select on the **Diagnostics** tab.
3. Select the **Enable tracing** option.
4. Select the **Open traces folder** link (should be `%LOCALAPPDATA%/Microsoft/Power BI Desktop/Traces`).
5. Delete existing trace files.
6. Perform your tests.
7. Close Power BI Desktop to ensure all log files are flushed to disk.

Here are steps you can take for initial testing in Power BI Desktop:

1. Close Power BI Desktop.
2. Clear your trace directory.
3. Open Power BI desktop, and enable tracing.
4. Connect to your data source, and select Direct Query mode.
5. Select a table in the navigator, and select **Edit**.
6. Manipulate the query in various ways, including:

- Take the First N rows (for example, 10).
- Set equality filters on different data types (int, string, bool, and so on).
- Set other range filters (greater than, less than).
- Filter on NULL / NOT NULL.
- Select a sub-set of columns.
- Aggregate / Group By different column combinations.
- Add a column calculated from other columns (`[C] = [A] + [B]`).
- Sort on one column, multiple columns. 7. Expressions that fail to fold will result in a warning bar. Note the failure, remove the step, and move to the next test case. Details about the cause of the failure should be emitted to the trace logs. 8. Close Power BI Desktop. 9. Copy the trace files to a new directory. 10. Use the recommend Power BI workbook to parse and analyze the trace files.

Once you have simple queries working, you can then try Direct Query scenarios (for example, building reports in the Report Views). The queries generated in Direct Query mode will be significantly more complex (that is, use of sub-selects, COALESCE statements, and aggregations).

Concatenation of strings in Direct Query mode

The M engine does basic type size limit validation as part of its query folding logic. If you are receiving a folding

error when trying to concatenate two strings that potentially overflow the maximum size of the underlying database type:

1. Ensure that your database can support up-conversion to CLOB types when string concat overflow occurs.
2. Set the `TolerateConcatOverflow` option for Odbc.DataSource to `true`.

The [DAX CONCATENATE function](#) is currently not supported by Power Query/ODBC extensions. Extension authors should ensure string concatenation works through the query editor by adding calculated columns (`[stringCol1] & [stringCol2]`). When the capability to fold the CONCATENATE operation is added in the future, it should work seamlessly with existing extensions.

Enabling Direct Query for an ODBC based connector

21 minutes to read • [Edit Online](#)

Overview

Using M's built-in [Odbc.DataSource](#) function is the recommended way to create custom connectors for data sources that have an existing ODBC driver and/or support a SQL query syntax. Wrapping the [Odbc.DataSource](#) function will allow your connector to inherit default query folding behavior based on the capabilities reported by your driver. This will enable the M engine to generate SQL statements based on filters and other transformations defined by the user within the Power Query experience, without having to provide this logic within the connector itself.

ODBC extensions can optionally enable Direct Query mode, allowing Power BI to dynamically generate queries at runtime without pre-caching the user's data model.

NOTE

Enabling Direct Query support raises the difficulty and complexity level of your connector. When Direct Query is enabled, Power BI will prevent the M engine from compensating for operations that cannot be fully pushed to the underlying data source.

This section builds on the concepts presented in the M Extensibility Reference, and assumes familiarity with the creation of a basic Data Connector.

Refer to the [SqlODBC sample](#) for most of the code examples in the sections below. Additional samples can be found in the ODBC samples directory.

ODBC Extensibility Functions

The M engine provides two ODBC related data source functions: [Odbc.DataSource](#), and [Odbc.Query](#).

The [Odbc.DataSource](#) function provides a default navigation table with all databases, tables, and views from your system, supports query folding, and allows for a range of customization options. The majority of ODBC based extensions will use this as their primary extensibility function. The function accepts two arguments—a connection string, and an options record to provide behavior overrides.

The [Odbc.Query](#) function allows you to execute SQL statements through an ODBC driver. It acts as a passthrough for query execution. Unlike the [Odbc.DataSource](#) function, it doesn't provide query folding functionality, and requires that SQL queries be provided by the connector (or end user). When building a custom connector, this function is typically used internally to run queries to retrieve metadata that might not be exposed through regular ODBC channels. The function accepts two arguments—a connection string, and a SQL query.

Parameters for your Data Source Function

Custom connectors can accept any number of function arguments, but to remain consistent with the built-in data source functions shipped with Power Query, the following guidelines are recommended:

- Require the minimal set of parameters used to establish a connection to your server. The less parameters end users need to provide, the easier your connector will be to use.
- Although you can define parameters with a fixed number of values (that is, a dropdown list in the UI), parameters are entered before the user is authenticated. Any values that can be discovered

programmatically after the user is authenticated (such as catalog or database name) should be selectable through the Navigator. The default behavior for the [Odbc.DataSource](#) function will be to return a hierarchical navigation table consisting of Catalog (Database), Schema, and Table names, although this can be overridden within your connector.

- If you feel your users will typically know what values to enter for items they would select from the Navigator (such as the database name), make these parameters optional. Parameters that can be discovered programmatically should not be made required.
- The last parameter for your function should be an optional record called "options". This parameter typically allows advanced users to set common ODBC related properties (such as CommandTimeout), set behavior overrides specific to your connector, and allows for future extensibility without impacting backwards compatibility for your function.
- Security/credential related arguments MUST never be part of your data source function parameters, as values entered in the connect dialog will be persisted to the user's query. Credential related parameters should be specified as part of the connector's supported Authentication methods.

By default, all required parameters for your data source function are factored into the Data Source Path value used to identify user credentials.

Note that while the UI for the built-in [Odbc.DataSource](#) function provides a dropdown that allows the user to select a DSN, this functionality is not available through extensibility. If your data source configuration is complex enough to require a fully customizable configuration dialog, it's recommended you require your end users to pre-configure a system DSN, and have your function take in the DSN name as a text field.

Parameters for Odbc.DataSource

The [Odbc.DataSource](#) function takes two parameters—a connectionString for your driver, and an options record that lets you override various driver behaviors. Through the options record you can override capabilities and other information reported by the driver, control the navigator behavior, and affect the SQL queries generated by the M engine.

The supported options records fields fall into two categories—those that are public / always available, and those that are only available in an extensibility context.

The following table describes the public fields in the options record.

FIELD	DESCRIPTION
CommandTimeout	A duration value that controls how long the server-side query is allowed to run before it's cancelled. Default: 10 minutes
ConnectionTimeout	A duration value that controls how long to wait before abandoning an attempt to make a connection to the server. Default: 15 seconds

FIELD	DESCRIPTION
CreateNavigationProperties	<p>A logical value that sets whether to generate navigation properties on the returned tables. Navigation properties are based on foreign key relationships reported by the driver, and show up as "virtual" columns that can be expanded in the query editor, creating the appropriate join.</p> <p>If calculating foreign key dependencies is an expensive operation for your driver, you may want to set this value to false.</p> <p>Default: true</p>
HierarchicalNavigation	<p>A logical value that sets whether to view the tables grouped by their schema names. When set to false, tables will be displayed in a flat list under each database.</p> <p>Default: false</p>
SqlCompatibleWindowsAuth	<p>A logical value that determines whether to produce a SQL Server compatible connection string when using Windows Authentication—Trusted_Connection=Yes.</p> <p>If your driver supports Windows Authentication, but requires additional or alternative settings in your connection string, you should set this value to false and use the CredentialConnectionString option record field described below.</p> <p>Default: true</p>

The following table describes the options record fields that are only available through extensibility. Fields that aren't simple literal values are described in subsequent sections.

FIELD	DESCRIPTION
AstVisitor	<p>A record containing one or more overrides to control SQL query generation. The most common usage of this field is to provide logic to generate a LIMIT/OFFSET clause for drivers that don't support TOP.</p> <p>Fields include:</p> <ul style="list-style-type: none"> • Constant • LimitClause <p>See the AstVisitor section for more information.</p>
ClientConnectionPooling	<p>A logical value that enables client-side connection pooling for the ODBC driver. Most drivers will want to set this value to true.</p> <p>Default: false</p>

FIELD	DESCRIPTION
CredentialConnectionString	<p>A text or record value used to specify credential related connection string properties.</p> <p>See the Credential section for more information.</p>
HideNativeQuery	<p>A logical value that controls whether your connector allows native SQL statements to be passed in by a query using the Value.NativeQuery() function.</p> <p>Note: this functionality is currently not exposed in the Power Query user experience. Users would need to manually edit their queries to take advantage of this capability.</p> <p>Default: false</p>
ImplicitTypeConversions	<p>A table value containing implicit type conversions supported by your driver or backend server. Values in this table are additive to the conversions reported by the driver itself.</p> <p>This field is typically used in conjunction with the SQLGetTypeInfo field when overriding data type information reported by the driver.</p> <p>See the ImplicitTypeConversions section for more information.</p>
OnError	<p>An error handling function that receives an errorRecord parameter of type record.</p> <p>Common uses of this function include handling SSL connection failures, providing a download link if your driver isn't found on the system, and reporting authentication errors.</p> <p>See the OnError section for more information.</p>
SoftNumbers	<p>Allows the M engine to select a compatible data type when conversion between two specific numeric types isn't declared as supported in the SQL_CONVERT_* capabilities.</p> <p>Default: false</p>
SqlCapabilities	<p>A record providing various overrides of driver capabilities, and a way to specify capabilities that aren't expressed through ODBC 3.8.</p> <p>See the SqlCapabilities section for more information.</p>
SQLColumns	<p>A function that allows you to modify column metadata returned by the SQLColumns function.</p> <p>See the SQLColumns section for more information.</p>

FIELD	DESCRIPTION
SQLGetFunctions	<p>A record that allows you to override values returned by calls to SQLGetFunctions.</p> <p>A common use of this field is to disable the use of parameter binding, or to specify that generated queries should use CAST rather than CONVERT.</p> <p>See the SQLGetFunctions section for more information.</p>
SQLGetInfo	<p>A record that allows you to override values returned by calls to SQLGetInfo.</p> <p>See the SQLGetInfo section for more information.</p>
SQLGetTypeInfo	<p>A table, or function that returns a table, that overrides the type information returned by SQLGetTypeInfo.</p> <p>When the value is set to a table, the value completely replaces the type information reported by the driver. SQLGetTypeInfo won't be called.</p> <p>When the value is set to a function, your function will receive the result of the original call to SQLGetTypeInfo, allowing you to modify the table.</p> <p>This field is typically used when there's a mismatch between data types reported by SQLGetTypeInfo and SQLColumns.</p> <p>See the SQLGetTypeInfo section for more information.</p>
SQLTables	<p>A function that allows you to modify the table metadata returned by a call to SQLTables.</p> <p>See the SQLTables section for more information.</p>
TolerateConcatOverflow	<p>Allows conversion of numeric and text types to larger types if an operation would cause the value to fall out of range of the original type.</p> <p>For example, when adding Int32.Max + Int32.Max, the engine will cast the result to Int64 when this setting is set to true. When adding a VARCHAR(4000) and a VARCHAR(4000) field on a system that supports a maximize VARCHAR size of 4000, the engine will cast the result into a CLOB type.</p> <p>Default: false</p>

FIELD	DESCRIPTION
UseEmbeddedDriver	<p>(internal use): A logical value that controls whether the ODBC driver should be loaded from a local directory (using new functionality defined in the ODBC 4.0 specification). This is generally only set by connectors created by Microsoft that ship with Power Query.</p> <p>When set to false, the system ODBC driver manager will be used to locate and load the driver.</p> <p>Most connectors should not need to set this field.</p> <p>Default: false</p>

Overriding AstVisitor

The AstVisitor field is set through the [Odbc.DataSource](#) options record. It's used to modify SQL statements generated for specific query scenarios.

NOTE

Drivers that support `LIMIT` and `OFFSET` clauses (rather than `TOP`) will want to provide a LimitClause override for AstVisitor.

Constant

Providing an override for this value has been deprecated and may be removed from future implementations.

LimitClause

This field is a function that receives two `Int64.Type` arguments (skip, take), and returns a record with two text fields (Text, Location).

```
LimitClause = (skip as nullable number, take as number) as record => ...
```

The skip parameter is the number of rows to skip (that is, the argument to `OFFSET`). If an offset is not specified, the skip value will be null. If your driver supports `LIMIT`, but does not support `OFFSET`, the LimitClause function should return an unimplemented error (...) when skip is greater than 0.

The take parameter is the number of rows to take (that is, the argument to `LIMIT`).

The `Text` field of the result contains the SQL text to add to the generated query.

The `Location` field specifies where to insert the clause. The following table describes supported values.

VALUE	DESCRIPTION	EXAMPLE
AfterQuerySpecification	<p>LIMIT clause is put at the end of the generated SQL.</p> <p>This is the most commonly supported LIMIT syntax.</p>	<pre>SELECT a, b, c FROM table WHERE a > 10 LIMIT 5</pre>

Value	Description	Example
BeforeQuerySpecification	LIMIT clause is put before the generated SQL statement.	LIMIT 5 ROWS SELECT a, b, c FROM table WHERE a > 10
AfterSelect	LIMIT goes after the SELECT statement, and after any modifiers (such as DISTINCT).	SELECT DISTINCT LIMIT 5 a, b, c FROM table WHERE a > 10
AfterSelectBeforeModifiers	LIMIT goes after the SELECT statement, but before any modifiers (such as DISTINCT).	SELECT LIMIT 5 DISTINCT a, b, c FROM table WHERE a > 10

The following code snippet provides a LimitClause implementation for a driver that expects a LIMIT clause, with an optional OFFSET, in the following format: `[OFFSET <offset> ROWS] LIMIT <row_count>`

```
LimitClause = (skip, take) =>
  let
    offset = if (skip > 0) then Text.Format("OFFSET #{0} ROWS", {skip}) else "",
    limit = if (take <> null) then Text.Format("LIMIT #{0}", {take}) else ""
  in
  [
    Text = Text.Format("#{0} #{1}", {offset, limit}),
    Location = "AfterQuerySpecification"
  ]
```

The following code snippet provides a LimitClause implementation for a driver that supports LIMIT, but not OFFSET. Format: `LIMIT <row_count>`.

```
LimitClause = (skip, take) =>
  if (skip > 0) then error "Skip/Offset not supported"
  else
  [
    Text = Text.Format("LIMIT #{0}", {take}),
    Location = "AfterQuerySpecification"
  ]
```

Overriding SqlCapabilities

Field	Details
FractionalSecondsScale	A number value ranging from 1 to 7 that indicates the number of decimal places supported for millisecond values. This value should be set by connectors that want to enable query folding over datetime values. Default: null

FIELD	DETAILS
PrepareStatements	<p>A logical value that indicates that statements should be prepared using SQLPrepare.</p> <p>Default: false</p>
SupportsTop	<p>A logical value that indicates the driver supports the TOP clause to limit the number of returned rows.</p> <p>Default: false</p>
StringLiteralEscapeCharacters	<p>A list of text values that specify the character(s) to use when escaping string literals and LIKE expressions.</p> <p>Ex. {"\r\n"}</p> <p>Default: null</p>
SupportsDerivedTable	<p>A logical value that indicates the driver supports derived tables (sub-selects).</p> <p>This value is assumed to be true for drivers that set their conformance level to SQL_SC_SQL92_FULL (reported by the driver or overridden with the Sql92Conformance setting (see below)). For all other conformance levels, this value defaults to false.</p> <p>If your driver doesn't report the SQL_SC_SQL92_FULL compliance level, but does support derived tables, set this value to true.</p> <p>Note that supporting derived tables is required for many Direct Query scenarios.</p>
SupportsNumericLiterals	<p>A logical value that indicates whether the generated SQL should include numeric literals values. When set to false, numeric values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsStringLiterals	<p>A logical value that indicates whether the generated SQL should include string literals values. When set to false, string values will always be specified using Parameter Binding.</p> <p>Default: false</p>
SupportsOdbcDateLiterals	<p>A logical value that indicates whether the generated SQL should include date literals values. When set to false, date values will always be specified using Parameter Binding.</p> <p>Default: false</p>

FIELD	DETAILS
SupportsOdbcTimeLiterals	A logical value that indicates whether the generated SQL should include time literals values. When set to false, time values will always be specified using Parameter Binding. Default: false
SupportsOdbcTimestampLiterals	A logical value that indicates whether the generated SQL should include timestamp literals values. When set to false, timestamp values will always be specified using Parameter Binding. Default: false

Overriding SQLColumns

`SQLColumns` is a function handler that receives the results of an ODBC call to [SQLColumns](#). The source parameter contains a table with the data type information. This override is typically used to fix up data type mismatches between calls to `SQLGetTypeInfo` and `SQLColumns`.

For details of the format of the source table parameter, see:

<https://docs.microsoft.com/sql/odbc/reference/syntax/sqlcolumns-function>

Overriding SQLGetFunctions

This field is used to override SQLFunctions values returned by an ODBC driver. It contains a record whose field names are equal to the FunctionId constants defined for the ODBC [SQLGetFunctions](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#).

FIELD	DETAILS
SQL_CONVERT_FUNCTIONS	Indicates which function(s) are supported when doing type conversions. By default, the M Engine will attempt to use the CONVERT function. Drivers that prefer the use of CAST can override this value to report that only SQL_FN_CVT_CAST (numeric value of 0x2) is supported.
SQL_API_SQLBINDCOL	A logical (true/false) value that indicates whether the Mashup Engine should use the SQLBindCol API when retrieving data. When set to false, SQLGetData is used instead. Default: false

The following code snippet provides an example explicitly telling the M engine to use CAST rather than CONVERT.

```
SQLGetFunctions = [
    SQL_CONVERT_FUNCTIONS = 0x2 /* SQL_FN_CVT_CAST */
]
```

Overriding SQLGetInfo

This field is used to override SQLGetInfo values returned by an ODBC driver. It contains a record whose fields are names equal to the InfoType constants defined for the ODBC [SQLGetInfo](#) function. Numeric constants for each of these fields can be found in the [ODBC specification](#). The full list of InfoTypes that are checked can be found in the Mashup Engine trace files.

The following table contains commonly overridden SQLGetInfo properties:

FIELD	DETAILS
SQL_SQL_CONFORMANCE	An integer value that indicates the level of SQL-92 supported by the driver: (1) SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant. (2) SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant. (4) SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant. (8) SQL_SC_SQL92_FULL = Full level SQL-92 compliant. Note that in Power Query scenarios, the connector will be used in a Read Only mode. Most drivers will want to report a SQL_SC_SQL92_FULL compliance level, and override specific SQL generation behavior using the SQLGetInfo and SQLGetFunctions properties.
SQL_SQL92_PREDICATES	A bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92. See the SQL_SP_* constants in the ODBC specification.
SQL_AGGREGATE_FUNCTIONS	A bitmask enumerating support for aggregation functions. SQL_AF_ALL SQL_AF_AVG SQL_AF_COUNT SQL_AF_DISTINCT SQL_AF_MAX SQL_AF_MIN SQL_AF_SUM See the SQL_AF_* constants in the ODBC specification.

FIELD	DETAILS
SQL_GROUP_BY	<p>A integer value that specifies the relationship between the columns in the GROUP BY clause and the non-aggregated columns in the select list:</p> <p>SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column.</p> <p>SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported.</p> <p>SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.</p> <p>SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all non-aggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is data source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.</p> <p>See the SQL_GB_* constants in the ODBC specification.</p>

The following helper function can be used to create bitmask values from a list of integer values:

```
Flags = (flags as list) =>
let
    Loop = List.Generate(
        ()=> [i = 0, Combined = 0],
        each [i] < List.Count(flags),
        each [i = [i]+1, Combined =*Number.BitwiseOr([Combined], flags{i})],
        each [Combined]),
    Result = List.Last(Loop, 0)
in
    Result;
```

Overriding SQLGetTypeInfo

`SQLGetTypeInfo` can be specified in two ways:

- A fixed `table` value that contains the same type information as an ODBC call to `SQLGetTypeInfo`.
- A function that accepts a table argument, and returns a table. The argument will contain the original results of the ODBC call to `SQLGetTypeInfo`. Your function implementation can modify/add to this table.

The first approach is used to completely override the values returned by the ODBC driver. The second approach is used if you want to add to or modify these values.

For details of the format of the types table parameter and expected return value, see the [SQLGetTypeInfo function reference](#).

SQLGetTypeInfo using a static table

The following code snippet provides a static implementation for `SQLGetTypeInfo`.

```
SQLGetTypeInfo = #table{
```

```

    { "TYPE_NAME",      "DATA_TYPE", "COLUMN_SIZE", "LITERAL_PREF", "LITERAL_SUFFIX", "CREATE_PARAS",
"NULLABLE", "CASE_SENSITIVE", "SEARCHABLE", "UNSIGNED_ATTRIBUTE", "FIXED_PREC_SCALE", "AUTO_UNIQUE_VALUE",
"LOCAL_TYPE_NAME", "MINIMUM_SCALE", "MAXIMUM_SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "NUM_PREC_RADIX",
"INTERNAL_PRECISION", "USER_DATA_TYPE" }, {
    { "char",           1,          65535,      "",      "",      "max. length",
1,           1,          3,          null,      0,          null,
"char",        null,        null,      -8,          null,      null,
0,           },      1,          19,         0,          10,         0,
    { "int8",          -5,          2,          0,          -5,          2,
1,           0,          0,          0,          null,      2,
"int8",        0,          0,          0,          null,      0,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          -7,         1,          "",          null,
1,           1,          3,          null,      0,          null,
"bit",        null,        null,      -7,          null,      null,
0,           },      1,          9,          10,         "",         null,
1,           0,          2,          null,      0,          null,
"date",        null,        null,      9,          1,          null,
0,           },      1,          3,          28,         null,      null,
1,           0,          2,          0,          0,          0,
"numeric",      0,          0,          2,          null,      10,
0,           },      1,          8,          15,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          6,          17,         null,      null,
1,           0,          2,          0,          6,          null,
"float8",      null,        null,      6,          null,      2,
0,           },      1,          -11,         2,         37,         null,
1,           0,          2,         null,      0,          null,
"uuid",        null,        null,      -11,         null,      null,
0,           },      1,          4,          10,         null,      null,
1,           0,          2,          0,          4,          0,
"int4",        0,          0,          4,          null,      2,
0,           },      1,          -1,         65535,      "",      null,
1,           1,          3,          null,      0,          null,
"text",        null,        null,      -10,         null,      null,
0,           },      1,          -4,         255,         "",         null,
1,           0,          2,          null,      0,          null,
"lo",        null,        null,      -4,          null,      null,
0,           },      1,          2,          28,         null,      null,
1,           0,          2,          0,          10,         0,
"numeric",      0,          6,          2,          null,      10,
0,           },      1,          7,          9,          null,      null,
1,           0,          2,          0,          10,         0,
"float4",      null,        null,      7,          null,      2,
0,           },      1,          5,          19,         null,      null,
1,           0,          2,          0,          5,          null,
"int2",        0,          0,          5,          null,      2,
0,           },      1,          -6,         2,          5,          null,
1,           0,          0,          0,          10,         0,
"int2",        0,          0,          5,          null,      2,
0,           }

```

```

        },
        { "timestamp",      11,      26,      "",      "",      null,
1,          0,           2,       null,      9,      0,      null,
"timestamp",      0,           38,       null,      9,      3,      null,
0,          },      91,      10,      "",      "",      null,
1,          0,           2,       null,      9,      1,      null,
"date",      null,       null,      9,      1,      null,
0,          },      93,      26,      "",      "",      null,
1,          0,           2,       null,      9,      3,      null,
"timestamp",      0,           38,       null,      9,      0,      null,
0,          },      -3,      255,      "",      "",      null,
1,          0,           2,       null,      -3,      0,      null,
"bytea",      null,       null,      -3,      null,      null,
0,          },      12,      65535,      "",      "",      "max. length",
1,          0,           2,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,          },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -8,      0,      null,
"char",      null,       null,      -8,      null,      null,
0,          },      -10,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -10,      0,      null,
"text",      null,       null,      -10,      null,      null,
0,          },      -9,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"varchar",      null,       null,      -9,      null,      null,
0,          },      -8,      65535,      "",      "",      "max. length",
1,          1,           3,       null,      -9,      0,      null,
"bpchar",      null,       null,      -9,      null,      null,
0,          } } }
);

```

SQLGetTypeInfo using a function

The following code snippets append the `bpchar` type to the existing types returned by the driver.

```

SQLGetTypeInfo = (types as table) as table =>
let
    newTypes = #table(
    {
        "TYPE_NAME",
        "DATA_TYPE",
        "COLUMN_SIZE",
        "LITERAL_PREF",
        "LITERAL_SUFFIX",
        "CREATE_PARAS",
        "NULLABLE",
        "CASE_SENSITIVE",
        "SEARCHABLE",
        "UNSIGNED_ATTRIBUTE",
        "FIXED_PREC_SCALE",
        "AUTO_UNIQUE_VALUE",
        "LOCAL_TYPE_NAME",
        "MINIMUM_SCALE",
        "MAXIMUM_SCALE",
        "SQL_DATA_TYPE",
        "SQL_DATETIME_SUB",
        "NUM_PREC_RADIX",
        "INTERNAL_PRECISION",
        "USER_DATA_TYPE"
    },
    // we add a new entry for each type we want to add
    {
        {
            "bpchar",
            -8,
            65535,
            """",
            """",
            "max. length",
            1,
            1,
            3,
            null,
            0,
            null,
            "bpchar",
            null,
            null,
            -9,
            null,
            null,
            0,
            0
        }
    )),
    append = Table.Combine({types, newTypes})
in
    append;

```

Setting the Connection String

The connection string for your ODBC driver is set using the first argument to the [Odbc.DataSource](#) and/or [Odbc.Query](#) functions. The value can be text, or an M record. When using the record, each field in the record will become a property in the connection string. All connection strings will require a Driver field (or DSN field if you require users to pre-configure a system level DSN). Credential related properties will be set separately (see below). Other properties will be driver specific.

The code snippet below shows the definition of a new data source function, creation of the `ConnectionString` record, and invocation of the [Odbc.DataSource](#) function.

```
[DataSource.Kind="SqlODBC", Publish="SqlODBC.Publish"]
shared SqlODBC.Contents = (server as text) =>
let
    ConnectionString = [
        Driver = "SQL Server Native Client 11.0",
        Server = server,
        MultiSubnetFailover = "Yes",
        ApplicationIntent = "ReadOnly",
        APP = "PowerBICustomConnector"
    ],
    OdbcDatasource = Odbc.DataSource(ConnectionString)
in
    OdbcDatasource;
```

Troubleshooting and Testing

To enable tracing in Power BI Desktop:

1. Go to **File > Options and settings > Options**.
2. Select on the **Diagnostics** tab.
3. Select the **Enable tracing** option.
4. Select the **Open traces folder** link (should be `%LOCALAPPDATA%/Microsoft/Power BI Desktop/Traces`).
5. Delete existing trace files.
6. Perform your tests.
7. Close Power BI Desktop to ensure all log files are flushed to disk.

Here are steps you can take for initial testing in Power BI Desktop:

1. Close Power BI Desktop.
2. Clear your trace directory.
3. Open Power BI desktop, and enable tracing.
4. Connect to your data source, and select Direct Query mode.
5. Select a table in the navigator, and select **Edit**.
6. Manipulate the query in various ways, including:

- Take the First N rows (for example, 10).
- Set equality filters on different data types (int, string, bool, and so on).
- Set other range filters (greater than, less than).
- Filter on NULL / NOT NULL.
- Select a sub-set of columns.
- Aggregate / Group By different column combinations.
- Add a column calculated from other columns (`[C] = [A] + [B]`).
- Sort on one column, multiple columns. 7. Expressions that fail to fold will result in a warning bar. Note the failure, remove the step, and move to the next test case. Details about the cause of the failure should be emitted to the trace logs. 8. Close Power BI Desktop. 9. Copy the trace files to a new directory. 10. Use the recommend Power BI workbook to parse and analyze the trace files.

Once you have simple queries working, you can then try Direct Query scenarios (for example, building reports in the Report Views). The queries generated in Direct Query mode will be significantly more complex (that is, use of sub-selects, COALESCE statements, and aggregations).

Concatenation of strings in Direct Query mode

The M engine does basic type size limit validation as part of its query folding logic. If you are receiving a folding

error when trying to concatenate two strings that potentially overflow the maximum size of the underlying database type:

1. Ensure that your database can support up-conversion to CLOB types when string concat overflow occurs.
2. Set the `TolerateConcatOverflow` option for Odbc.DataSource to `true`.

The [DAX CONCATENATE function](#) is currently not supported by Power Query/ODBC extensions. Extension authors should ensure string concatenation works through the query editor by adding calculated columns (`[stringCol1] & [stringCol2]`). When the capability to fold the CONCATENATE operation is added in the future, it should work seamlessly with existing extensions.

Handling Resource Path

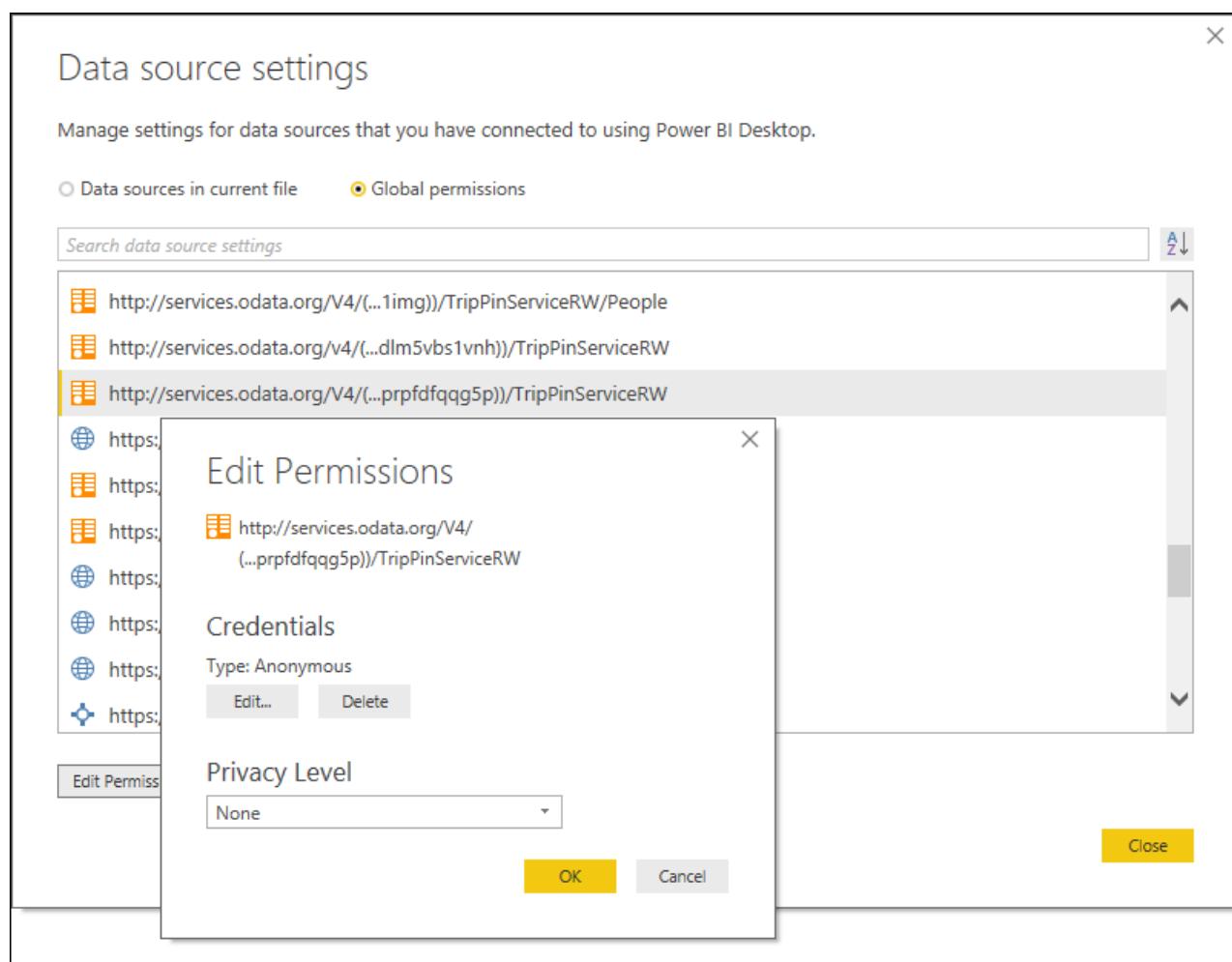
2 minutes to read • [Edit Online](#)

The M engine identifies a data source using a combination of its *Kind* and *Path*. When a data source is encountered during a query evaluation, the M engine will try to find matching credentials. If no credentials are found, the engine returns a special error that results in a credential prompt in Power Query.

The *Kind* value comes from [Data Source Kind](#) definition.

The *Path* value is derived from the *required parameters* of your data source function(s). Optional parameters aren't factored into the data source path identifier. As a result, all data source functions associated with a data source kind must have the same parameters. There's special handling for functions that have a single parameter of type `Uri.Type`. See below for further details.

You can see an example of how credentials are stored in the **Data source settings** dialog in Power BI Desktop. In this dialog, the Kind is represented by an icon, and the Path value is displayed as text.



[Note] If you change your data source function's required parameters during development, previously stored credentials will no longer work (because the path values no longer match). You should delete any stored credentials any time you change your data source function parameters. If incompatible credentials are found, you may receive an error at runtime.

Data Source Path Format

The **Path** value for a data source is derived from the data source function's required parameters.

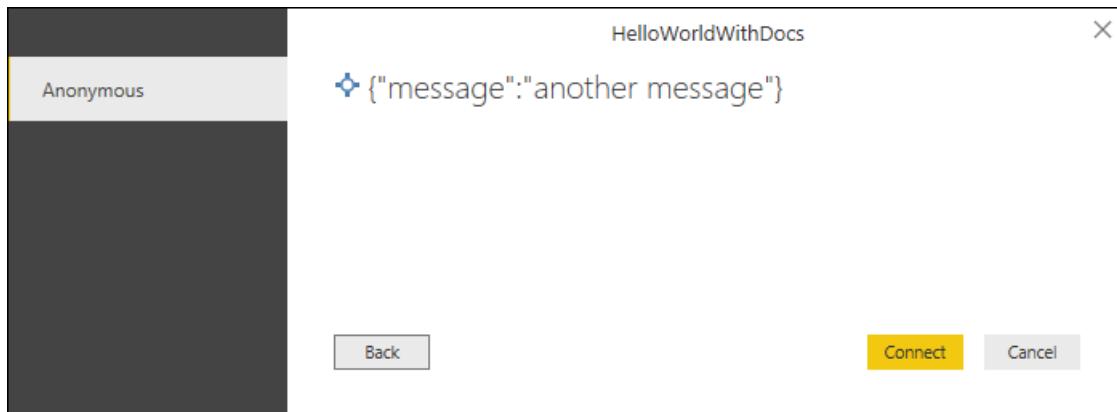
By default, you can see the actual string value in the **Data source settings** dialog in Power BI Desktop, and in the credential prompt. If the Data Source Kind definition has included a **Label** value, you'll see the label value instead.

For example, the data source function in the [HelloWorldWithDocs sample](#) has the following signature:

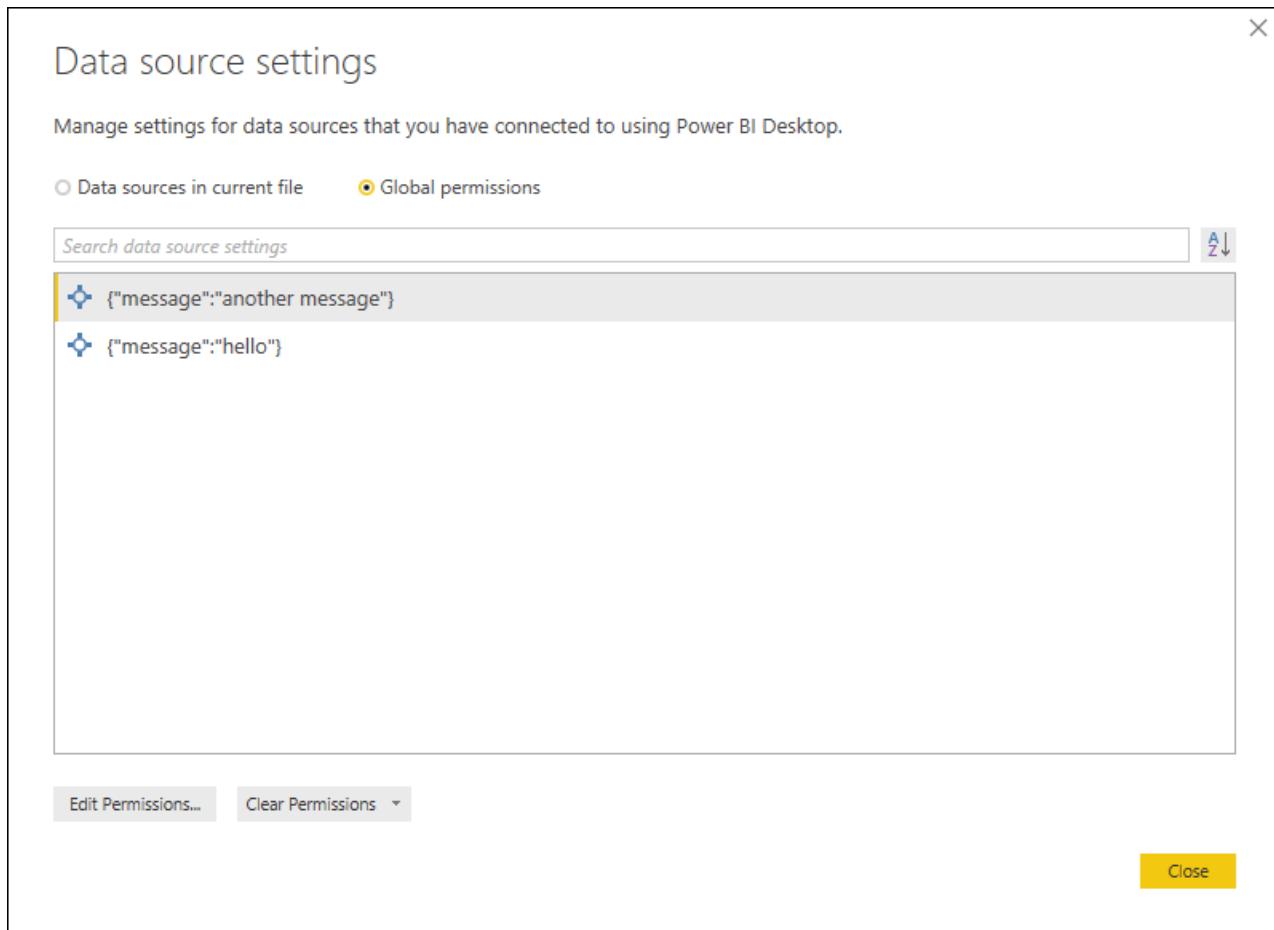
```
HelloWorldWithDocs.Contents = (message as text, optional count as number) as table => ...
```

The function has a single required parameter (**message**) of type **text**, and will be used to calculate the data source path. The optional parameter (**count**) will be ignored. The path would be displayed as follows:

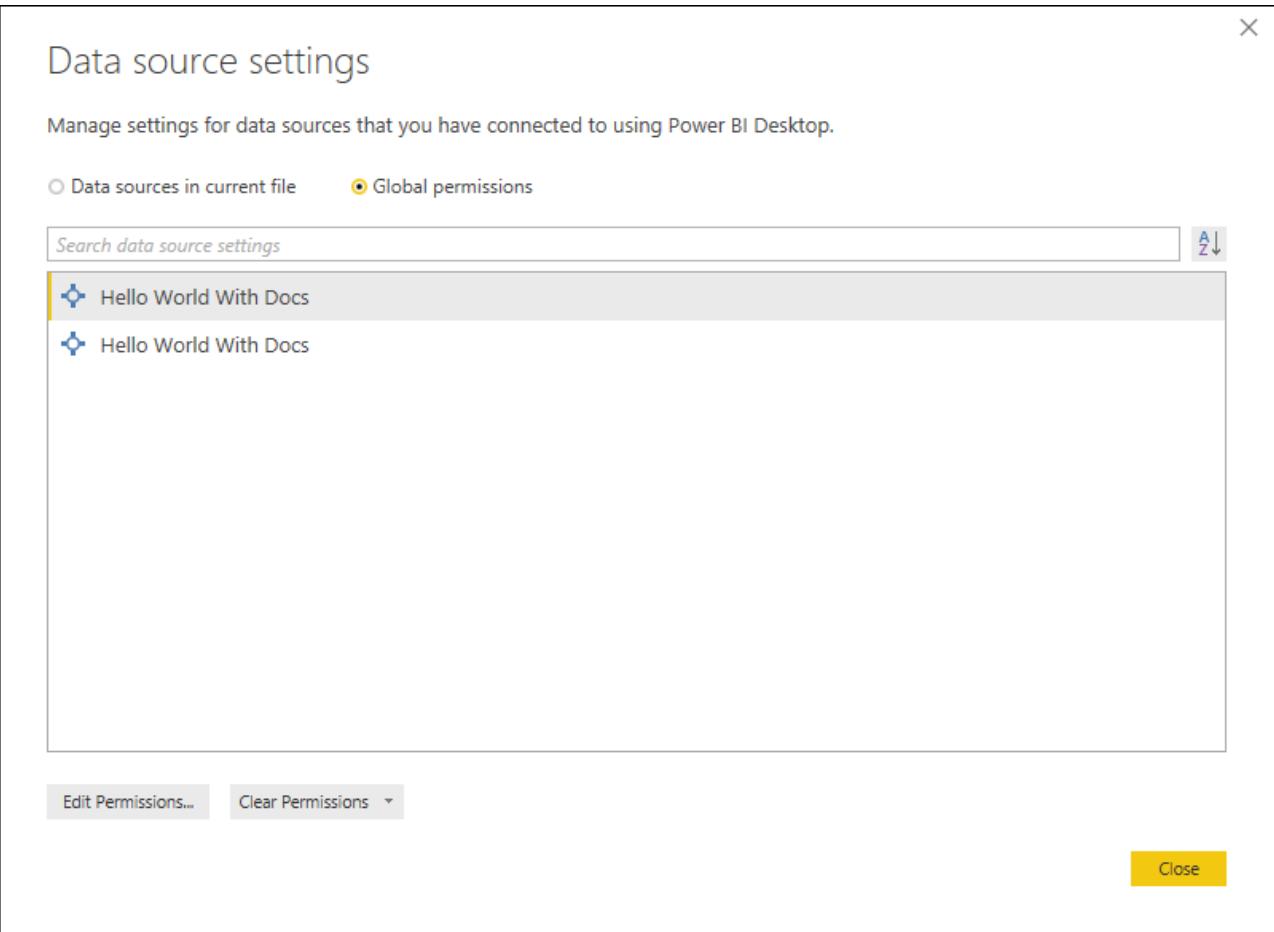
Credential prompt:



Data source settings UI:



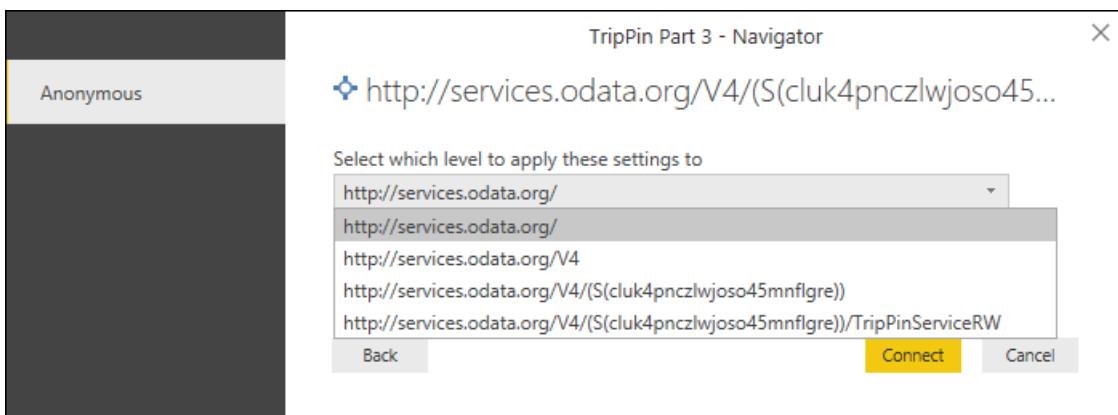
When a Label value is defined, the data source path value won't be shown:



[Note] We currently recommend that you *do not* include a Label for your data source if your function has required parameters, as users won't be able to distinguish between the different credentials they've entered. We are hoping to improve this in the future (that is, allowing data connectors to display their own custom data source paths).

Functions with a Uri parameter

Because data sources with a Uri-based identifier are so common, there's special handling in the Power Query UI when dealing with Uri-based data source paths. When a Uri-based data source is encountered, the credential dialog provides a dropdown allowing the user to select the base path, rather than the full path (and all paths in-between).



As `Uri.Type` is an *ascribed type* rather than a *primitive type* in the M language, you'll need to use the `Value.ReplaceType` function to indicate that your text parameter should be treated as a Uri.

```
shared GithubSample.Contents = Value.ReplaceType(Github.Contents, type function (url as Uri.type) as any);
```

Paging

2 minutes to read • [Edit Online](#)

REST APIs typically have some mechanism to transmit large volumes of records broken up into *pages* of results. Power Query has the flexibility to support many different paging mechanisms. However, since each paging mechanism is different, some amount of modification of the below examples is likely to be necessary to fit your situation.

Typical Patterns

The heavy lifting of compiling all page results into a single table is performed by the `Table.GenerateByPage()` helper function, which can generally be used with no modification. The following code snippets describe how to implement some common paging patterns. Regardless of pattern, you'll need to understand:

1. How do you request the next page of data?
2. Does the paging mechanism involve calculating values, or do you extract the URL for the next page from the response?
3. How do you know when to stop paging?
4. Are there parameters related to paging (such as "page size") that you should be aware of?

Handling Transformations

3 minutes to read • [Edit Online](#)

For situations where the data source response isn't presented in a format that Power BI can consume directly, Power Query can be used to perform a series of transformations.

Static Transformations

In most cases, the data is presented in a consistent way by the data source: column names, data types, and hierarchical structure are consistent for a given endpoint. In this situation it's appropriate to always apply the same set of transformations to get the data in a format acceptable to Power BI.

An example of static transformation can be found in the [TripPin Part 2 - Data Connector for a REST Service](#) tutorial when the data source is treated as a standard REST service:

```
let
    Source = TripPin.Feed("https://services.odata.org/v4/TripPinService/Airlines"),
    value = Source[value],
    toTable = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    expand = Table.ExpandRecordColumn(toTable, "Column1", {"AirlineCode", "Name"}, {"AirlineCode", "Name"})
in
    expand
```

The transformations in this example are:

1. `Source` is a Record returned from a call to `TripPin.Feed(...)`.
2. You pull the value from one of `Source`'s key-value pairs. The name of the key is `value`, and you store the result in a variable called `value`.
3. `value` is a list, which you convert to a table. Each element in `value` becomes a row in the table, which you can call `toTable`.
4. Each element in `value` is itself a Record. `toTable` has all of these in a single column: `"Column1"`. This step pulls all data with key `"AirlineCode"` into a column called `"AirlineCode"` and all data with key `"Name"` into a column called `"Name"`, for each row in `toTable`. `"Column1"` is replaced by these two new columns.

At the end of the day you're left with data in a simple tabular format that Power BI can consume and easily render:

	AirlineCode	Name
1	AA	American Airlines
2	FM	Shanghai Airline
3	MU	China Eastern Airlines

It's important to note that a sequence of static transformations of this specificity are only applicable to a *single* endpoint. In the example above, this sequence of transformations will only work if `"AirlineCode"` and `"Name"` exist in the REST endpoint response, since they are hard-coded into the M code. Thus, this sequence of transformations may not work if you try to hit the `/Event` endpoint.

This high level of specificity may be necessary for pushing data to a navigation table, but for more general data access functions it's recommended that you only perform transformations that are appropriate for all endpoints.

NOTE

Be sure to test transformations under a variety of data circumstances. If the user doesn't have any data at the `/airlines` endpoint, do your transformations result in an empty table with the correct schema? Or is an error encountered during evaluation? See [TripPin Part 7: Advanced Schema with M Types](#) for a discussion on unit testing.

Dynamic Transformations

More complex logic is sometimes needed to convert API responses into stable and consistent forms appropriate for Power BI data models.

Inconsistent API Responses

Basic M control flow (if statements, HTTP status codes, try...catch blocks, and so on) are typically sufficient to handle situations where there are a handful of ways in which the API responds.

Determining Schema On-The-Fly

Some APIs are designed such that multiple pieces of information must be combined to get the correct tabular format. Consider Smartsheet's `/sheets` endpoint response, which contains an array of column names and an array of data rows. The Smartsheet Connector is able to parse this response in the following way:

```
raw = Web.Contents(...),
columns = raw[columns],
columnTitles = List.Transform(columns, each [title]),
columnTitlesWithRowNumber = List.InsertRange(columnTitles, 0, {"RowNumber"}),

RowAsList = (row) =>
    let
        listOfCells = row[cells],
        cellValuesList = List.Transform(listOfCells, each if Record.HasFields(_, "value") then [value]
            else null),
        rowNumberFirst = List.InsertRange(cellValuesList, 0, {row[rowNumber]})
    in
        rowNumberFirst,

listOfRows = List.Transform(raw[rows], each RowAsList(_)),
result = Table.FromRows(listOfRows, columnTitlesWithRowNumber)
```

1. First deal with column header information. You can pull the `title` record of each column into a List, prepending with a `RowNumber` column that you know will always be represented as this first column.
2. Next you can define a function that allows you to parse a row into a List of cell `value`s. You can again prepend `rowNumber` information.
3. Apply your `RowAsList()` function to each of the `row`s returned in the API response.
4. Convert the List to a table, specifying the column headers.

Handling Transformations

3 minutes to read • [Edit Online](#)

For situations where the data source response isn't presented in a format that Power BI can consume directly, Power Query can be used to perform a series of transformations.

Static Transformations

In most cases, the data is presented in a consistent way by the data source: column names, data types, and hierarchical structure are consistent for a given endpoint. In this situation it's appropriate to always apply the same set of transformations to get the data in a format acceptable to Power BI.

An example of static transformation can be found in the [TripPin Part 2 - Data Connector for a REST Service](#) tutorial when the data source is treated as a standard REST service:

```
let
    Source = TripPin.Feed("https://services.odata.org/v4/TripPinService/Airlines"),
    value = Source[value],
    toTable = Table.FromList(value, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    expand = Table.ExpandRecordColumn(toTable, "Column1", {"AirlineCode", "Name"}, {"AirlineCode", "Name"})
in
    expand
```

The transformations in this example are:

1. `Source` is a Record returned from a call to `TripPin.Feed(...)`.
2. You pull the value from one of `Source`'s key-value pairs. The name of the key is `value`, and you store the result in a variable called `value`.
3. `value` is a list, which you convert to a table. Each element in `value` becomes a row in the table, which you can call `toTable`.
4. Each element in `value` is itself a Record. `toTable` has all of these in a single column: `"Column1"`. This step pulls all data with key `"AirlineCode"` into a column called `"AirlineCode"` and all data with key `"Name"` into a column called `"Name"`, for each row in `toTable`. `"Column1"` is replaced by these two new columns.

At the end of the day you're left with data in a simple tabular format that Power BI can consume and easily render:

	AirlineCode	Name
1	AA	American Airlines
2	FM	Shanghai Airline
3	MU	China Eastern Airlines

It's important to note that a sequence of static transformations of this specificity are only applicable to a *single* endpoint. In the example above, this sequence of transformations will only work if `"AirlineCode"` and `"Name"` exist in the REST endpoint response, since they are hard-coded into the M code. Thus, this sequence of transformations may not work if you try to hit the `/Event` endpoint.

This high level of specificity may be necessary for pushing data to a navigation table, but for more general data access functions it's recommended that you only perform transformations that are appropriate for all endpoints.

NOTE

Be sure to test transformations under a variety of data circumstances. If the user doesn't have any data at the `/airlines` endpoint, do your transformations result in an empty table with the correct schema? Or is an error encountered during evaluation? See [TripPin Part 7: Advanced Schema with M Types](#) for a discussion on unit testing.

Dynamic Transformations

More complex logic is sometimes needed to convert API responses into stable and consistent forms appropriate for Power BI data models.

Inconsistent API Responses

Basic M control flow (if statements, HTTP status codes, try...catch blocks, and so on) are typically sufficient to handle situations where there are a handful of ways in which the API responds.

Determining Schema On-The-Fly

Some APIs are designed such that multiple pieces of information must be combined to get the correct tabular format. Consider Smartsheet's `/sheets` endpoint response, which contains an array of column names and an array of data rows. The Smartsheet Connector is able to parse this response in the following way:

```
raw = Web.Contents(...),
columns = raw[columns],
columnTitles = List.Transform(columns, each [title]),
columnTitlesWithRowNumber = List.InsertRange(columnTitles, 0, {"RowNumber"}),

RowAsList = (row) =>
    let
        listOfCells = row[cells],
        cellValuesList = List.Transform(listOfCells, each if Record.HasFields(_, "value") then [value]
            else null),
        rowNumberFirst = List.InsertRange(cellValuesList, 0, {row[rowNumber]})
    in
        rowNumberFirst,

listOfRows = List.Transform(raw[rows], each RowAsList(_)),
result = Table.FromRows(listOfRows, columnTitlesWithRowNumber)
```

1. First deal with column header information. You can pull the `title` record of each column into a List, prepending with a `RowNumber` column that you know will always be represented as this first column.
2. Next you can define a function that allows you to parse a row into a List of cell `value`s. You can again prepend `rowNumber` information.
3. Apply your `RowAsList()` function to each of the `row`s returned in the API response.
4. Convert the List to a table, specifying the column headers.

Handling Schema

7 minutes to read • [Edit Online](#)

Depending on your data source, information about data types and column names may or may not be provided explicitly. OData REST APIs typically handle this using the [\\$metadata definition](#), and the Power Query [OData.Feed](#) method automatically handles parsing this information and applying it to the data returned from an [OData source](#).

Many REST APIs don't have a way to programmatically determine their schema. In these cases you'll need to include a schema definition in your connector.

Simple Hardcoded Approach

The simplest approach is to hardcode a schema definition into your connector. This is sufficient for most use cases.

Overall, enforcing a schema on the data returned by your connector has multiple benefits, such as:

- Setting the correct data types.
- Removing columns that don't need to be shown to end users (such as internal IDs or state information).
- Ensuring that each page of data has the same shape by adding any columns that might be missing from a response (REST APIs commonly indicate that fields should be null by omitting them entirely).

Viewing the Existing Schema with [Table.Schema](#)

Consider the following code that returns a simple table from the [TripPin OData sample service](#):

```
let
    url = "https://services.odata.org/TripPinWeb ApiService/Airlines",
    source = Json.Document(Web.Contents(url))[value],
    asTable = Table.FromRecords(source)
in
    asTable
```

NOTE

TripPin is an OData source, so realistically it would make more sense to simply use the [OData.Feed](#) function's automatic schema handling. In this example you'll be treating the source as a typical REST API and using [Web.Contents](#) to demonstrate the technique of hardcoding a schema by hand.

This table is the result:

	AirlineCode	Name
1	AA	American Airlines
2	FM	Shanghai Airline
3	MU	China Eastern Airlines

You can use the handy [Table.Schema](#) function to check the data type of the columns:

```

let
    url = "https://services.odata.org/TripPinWebApiService/Airlines",
    source = Json.Document(Web.Contents(url))[value],
    asTable = Table.FromRecords(source)
in
    Table.Schema(asTable)

```

	Name	Position	TypeName	Kind	IsNullable
1	AirlineCode	0	Any.Type	any	TRUE
2	Name	1	Any.Type	any	TRUE

Both AirlineCode and Name are of `any` type. `Table.Schema` returns a lot of metadata about the columns in a table, including names, positions, type information, and many advanced properties such as Precision, Scale, and MaxLength. For now you should only concern yourself with the ascribed type (`TypeName`), primitive type (`Kind`), and whether the column value might be null (`IsNullable`).

Defining a Simple Schema Table

Your schema table will be composed of two columns:

COLUMN	DETAILS
Name	The name of the column. This must match the name in the results returned by the service.
Type	The M data type you're going to set. This can be a primitive type (text, number, datetime, and so on), or an ascribed type (Int64.Type, Currency.Type, and so on).

The hardcoded schema table for the `Airlines` table will set its `AirlineCode` and `Name` columns to `text` and looks like this:

```

Airlines = #table({ "Name", "Type" }, {
    {"AirlineCode", type text},
    {"Name", type text}
})

```

As you look to some of the other endpoints, consider the following schema tables:

The `Airports` table has four fields you'll want to keep (including one of type `record`):

```

Airports = #table({ "Name", "Type" }, {
    {"IcaoCode", type text},
    {"Name", type text},
    {"IataCode", type text},
    {"Location", type record}
})

```

The `People` table has seven fields, including `list s` (`Emails`, `AddressInfo`), a `nullable` column (`Gender`), and a column with an *ascribed type* (`Concurrency`):

```

People = #table({ "Name", "Type"}, {
    {"UserName", type text},
    {"FirstName", type text},
    {"LastName", type text},
    {"Emails", type list},
    {"AddressInfo", type list},
    {"Gender", type nullable text},
    {"Concurrency", Int64.Type}
})

```

You can put all of these tables into a single master schema table `SchemaTable`:

```

SchemaTable = #table({ "Entity", "SchemaTable"}, {
    {"Airlines", Airlines},
    {"Airports", Airports},
    {"People", People}
})

```

ABC 123 Entity		ABC 123 SchemaTable	
1	Airlines	2	Airports
3	People		
Name			Type
AirlineCode			Type
Name			Type

The SchemaTransformTable Helper Function

The `SchemaTransformTable` helper function described below will be used to enforce schemas on your data. It takes the following parameters:

PARAMETER	TYPE	DESCRIPTION
table	table	The table of data you'll want to enforce your schema on.
schema	table	<p>The schema table to read column info from, with the following type:</p> <pre>type table [Name = text, Type = type]</pre>

PARAMETER	TYPE	DESCRIPTION
enforceSchema	number	<p>(optional) An enum that controls behavior of the function.</p> <p>The default value (<code>EnforceSchema.Strict = 1</code>) ensures that the output table will match the schema table that was provided by adding any missing columns, and removing extra columns.</p> <p>The <code>EnforceSchema.IgnoreExtraColumns = 2</code> option can be used to preserve extra columns in the result.</p> <p>When <code>EnforceSchema.IgnoreMissingColumns = 3</code> is used, both missing columns and extra columns will be ignored.</p>

The logic for this function looks something like this:

1. Determine if there are any missing columns from the source table.
2. Determine if there are any extra columns.
3. Ignore structured columns (of type `list`, `record`, and `table`), and columns set to type `any`.
4. Use `Table.TransformColumnTypes` to set each column type.
5. Reorder columns based on the order they appear in the schema table.
6. Set the type on the table itself using `Value.ReplaceType`.

NOTE

The last step to set the table type will remove the need for the Power Query UI to infer type information when viewing the results in the query editor, which can sometimes result in a double-call to the API.

Putting It All Together

In the greater context of a complete extension, the schema handling will take place when a table is returned from the API. Typically this functionality takes place at the lowest level of the paging function (if one exists), with entity information passed through from a navigation table.

Because so much of the implementation of paging and navigation tables is context-specific, the complete example of implementing a hardcoded schema-handling mechanism won't be shown here. [This TripPin example](#) demonstrates how an end-to-end solution might look.

Sophisticated Approach

The hardcoded implementation discussed above does a good job of making sure that schemas remain consistent for simple JSON responses, but it's limited to parsing the first level of the response. Deeply nested data sets would benefit from the following approach, which takes advantage of M Types.

Here is a quick refresh about types in the M language from the [Language Specification](#):

A **type value** is a value that **classifies** other values. A value that is classified by a type is said to **conform** to that type. The M type system consists of the following kinds of types:

- Primitive types, which classify primitive values (`binary`, `date`, `datetime`, `datetimezone`, `duration`, `list`,

`logical`, `null`, `number`, `record`, `text`, `time`, `type`) and also include a number of abstract types (`function`, `table`, `any`, and `none`).

- Record types, which classify record values based on field names and value types.
- List types, which classify lists using a single item base type.
- Function types, which classify function values based on the types of their parameters and return values.
- Table types, which classify table values based on column names, column types, and keys.
- Nullable types, which classify the value `null` in addition to all the values classified by a base type.
- Type types, which classify values that are types.

Using the raw JSON output you get (and/or by looking up the definitions in the [service's \\$metadata](#)), you can define the following record types to represent OData complex types:

```
LocationType = type [
    Address = text,
    City = CityType,
    Loc = LocType
];

CityType = type [
    CountryRegion = text,
    Name = text,
    Region = text
];

LocType = type [
    #"type" = text,
    coordinates = {number},
    crs = CrsType
];

CrsType = type [
    #"type" = text,
    properties = record
];
```

Notice how `LocationType` references the `CityType` and `LocType` to represent its structured columns.

For the top-level entities that you'll want represented as Tables, you can define *table types*:

```
AirlinesType = type table [
    AirlineCode = text,
    Name = text
];

AirportsType = type table [
    Name = text,
    IataCode = text,
    Location = LocationType
];

PeopleType = type table [
    UserName = text,
    FirstName = text,
    LastName = text,
    Emails = {text},
    AddressInfo = {nullable LocationType},
    Gender = nullable text,
    Concurrency Int64.Type
];
```

You can then update your `schemaTable` variable (which you can use as a lookup table for entity-to-type mappings) to use these new type definitions:

```
SchemaTable = #table({ "Entity", "Type"}, {  
    {"Airlines", AirlinesType},  
    {"Airports", AirportsType},  
    {"People", PeopleType}  
});
```

You can rely on a common function (`Table.ChangeType`) to enforce a schema on your data, much like you used `SchemaTransformTable` in the earlier exercise. Unlike `SchemaTransformTable`, `Table.ChangeType` takes an actual M table type as an argument, and will apply your schema *recursively* for all nested types. Its signature is:

```
Table.ChangeType = (table, tableType as type) as nullable table => ...
```

NOTE

For flexibility, the function can be used on tables as well as lists of records (which is how tables are represented in a JSON document).

You'll then need to update the connector code to change the `schema` parameter from a `table` to a `type`, and add a call to `Table.ChangeType`. Again, the details for doing so are very implementation-specific and thus not worth going into in detail here. [This extended TripPin connector example](#) demonstrates an end-to-end solution implementing this more sophisticated approach to handling schema.

Status Code Handling with `Web.Contents`

2 minutes to read • [Edit Online](#)

The `Web.Contents` function has some built in functionality for dealing with certain HTTP status codes. The default behavior can be overridden in your extension using the `ManualStatusHandling` field in the [options record](#).

Automatic retry

`Web.Contents` will automatically retry requests that fail with one of the following status codes:

CODE	STATUS
408	Request Timeout
429	Too Many Requests
503	Service Unavailable
504	Gateway Timeout
509	Bandwidth Limit Exceeded

Requests will be retried up to 3 times before failing. The engine uses an exponential back-off algorithm to determine how long to wait until the next retry, unless the response contains a `Retry-after` header. When the header is found, the engine will wait the specified number of seconds before the next retry. The minimum supported wait time is 0.5 seconds, and the maximum value is 120 seconds.

NOTE

The `Retry-after` value must be in the `delta-seconds` format. The `HTTP-date` format is currently not supported.

Authentication exceptions

The following status codes will result in a credentials exception, causing an authentication prompt asking the user to provide credentials (or re-login in the case of an expired OAuth token).

CODE	STATUS
401	Unauthorized
403	Forbidden

NOTE

Extensions are able to use the `ManualStatusHandling` option with status codes 401 and 403, which is not something that can be done in `Web.Contents` calls made outside of an extension context (that is, directly from Power Query).

Redirection

The follow status codes will result in an automatic redirect to the URI specified in the `Location` header. A missing `Location` header will result in an error.

CODE	STATUS
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
307	Temporary Redirect

NOTE

Only status code 307 will keep a `POST` request method. All other redirect status codes will result in a switch to `GET`.

Wait-Retry Pattern

2 minutes to read • [Edit Online](#)

In some situations a data source's behavior does not match that expected by Power Query's [default HTTP code handling](#). The examples below show how to work around this situation.

In this scenario you'll be working with a REST API that occasionally returns a 500 status code, indicating an internal server error. In these instances, you could wait a few seconds and retry, potentially a few times before you give up.

ManualStatusHandling

If `Web.Contents` gets a 500 status code response, it throws a `DataSource.Error` by default. You can override this behavior by providing a list of codes as an optional argument to `Web.Contents`:

```
response = Web.Contents(url, [ManualStatusHandling={404, 500}])
```

By specifying the status codes in this way, Power Query will continue to process the web response as normal. However, normal response processing is often not appropriate in these cases. You'll need to understand that an abnormal response code has been received and perform special logic to handle it. To determine the response code that was returned from the web service, you can access it from the `meta` Record that accompanies the response:

```
responseCode = Value.Metadata(response)[Response.Status]
```

Based on whether `responseCode` is 200 or 500, you can either process the result as normal, or follow your wait-retry logic that you'll flesh out in the next section.

NOTE

We recommended that you use `Binary.Buffer` to force Power Query to cache the `Web.Contents` results if you'll be implementing complex logic such as the Wait-Retry pattern shown here. This prevents Power Query's multi-threaded execution from making multiple calls with potentially inconsistent results.

Value.WaitFor

`Value.WaitFor()` is a standard [helper function](#) that can usually be used with no modification. It works by building a List of retry attempts.

producer Argument

This contains the task to be (possibly) retried. It's represented as a function so that the iteration number can be used in the `producer` logic. The expected behavior is that `producer` will return `null` if a retry is determined to be necessary. If anything other than `null` is returned by `producer`, that value is in turn returned by `Value.WaitFor`.

delay Argument

This contains the logic to execute between retries. It's represented as a function so that the iteration number can be used in the `delay` logic. The expected behavior is that `delay` returns a Duration.

count Argument (optional)

A maximum number of retries can be set by providing a number to the `count` argument.

Putting It All Together

The following example shows how `ManualStatusHandling` and `Value.WaitFor` can be used to implement a delayed retry in the event of a 500 response. Wait time between retries here is shown as doubling with each try, with a maximum of 5 retries.

```
let
  waitForResult = Value.WaitFor(
    (iteration) =>
      let
        result = Web.Contents(url, [ManualStatusHandling = {500}]),
        buffered = Binary.Buffer(result),
        status = Value.Metadata(result)[Response.Status],
        actualResult = if status = 500 then null else buffered
      in
        actualResult,
    (iteration) => #duration(0, 0, 0, Number.Power(2, iteration)),
    5)
in
  waitForResult,
```

Handling Unit Testing

2 minutes to read • [Edit Online](#)

For both simple and complex connectors, adding unit tests is a best practice and highly recommended.

Unit testing is accomplished in the context of Visual Studio's [Power Query SDK](#). Each test is defined as a `Fact` that has a name, an expected value, and an actual value. In most cases, the "actual value" will be an M expression that tests part of your expression.

Consider a very simple extension that exports three functions:

```
section UnitTesting;

shared UnitTesting.ReturnsABC = () => "ABC";
shared UnitTesting.Returns123 = () => "123";
shared UnitTesting.ReturnTableWithFiveRows = () => Table.Repeat(#table({{"a"}},{{1}}),5);
```

This unit test code is made up of a number of Facts, and a bunch of common code for the unit test framework (`ValueToText`, `Fact`, `Facts`, `Facts.Summarize`). The following code provides an example set of Facts (see [UnitTestng.query.pq](#) for the common code):

```
section UnitTestingTests;

shared MyExtension.UnitTesting =
[
    // Put any common variables here if you only want them to be evaluated once

    // Fact(<Name of the Test>, <Expected Value>, <Actual Value>)
    facts =
    {
        Fact("Check that this function returns 'ABC'", // name of the test
            "ABC", // expected value
            UnitTesting.ReturnsABC() // expression to evaluate (let or single statement)
        ),
        Fact("Check that this function returns '123'",
            "123",
            UnitTesting.Returns123()
        ),
        Fact("Result should contain 5 rows",
            5,
            Table.RowCount(UnitTesting.ReturnTableWithFiveRows())
        ),
        Fact("Values should be equal (using a let statement)",
            "Hello World",
            let
                a = "Hello World"
            in
                a
        ),
        report = Facts.Summarize(facts)
    }[report];
```

Running the sample in Visual Studio will evaluate all of the Facts and give you a visual summary of the pass rates:

Query Result		
Result	Notes	Details
Success	All 4 Passed !!! ✓	100% success rate
Success ✓	Check that this function returns 'ABC'	("ABC" = "ABC")
Success ✓	Check that this function returns '123'	("123" = "123")
Success ✓	Result should contain 5 rows	(5 = 5)
Success ✓	Values should be equal (using a let statement)	("Hello World" = "Hello World")

Implementing unit testing early in the connector development process enables you to follow the principles of test-driven development. Imagine that you need to write a function called `Uri.GetHost` that returns only the host data from a URI. You might start by writing a test case to verify that the function appropriately performs the expected function:

```
Fact("Returns host from URI",
    "https://bing.com",
    Uri.GetHost("https://bing.com/subpath/query?param=1&param2=hello")
),
Fact("Handles port number appropriately",
    "https://bing.com:8080",
    Uri.GetHost("https://bing.com:8080/subpath/query?param=1&param2=hello")
)
```

Additional tests can be written to ensure that the function appropriately handles edge cases.

An early version of the function might pass some but not all tests:

```
Uri.GetHost = (url) =>
    let
        parts = Uri.Parts(url)
    in
        parts[Scheme] & "://" & parts[Host]
```

Query Result		
Result	Notes	Details
⊖	5 Passed ⓘ 1 Failed ⓘ	83% success rate
Success ✓	Check that this function returns 'ABC'	("ABC" = "ABC")
Success ✓	Check that this function returns '123'	("123" = "123")
Success ✓	Result should contain 5 rows	(5 = 5)
Success ✓	Values should be equal (using a let statement)	("Hello World" = "Hello World")
Success ✓	Returns host from URI	("https://bing.com" = "https://bing.com")
Failure ⊖	Handles port number appropriately	("https://bing.com:8080" <> "https://bing.com")

The [final version of the function](#) should pass all unit tests. This also makes it easy to ensure that future updates to the function do not accidentally remove any of its basic functionality.

M Query Output

Output Log Errors Credentials

Query Result

Result	Notes	Details
Success	All 6 Passed !!! ✓	100% success rate
Success ✓	Check that this function returns 'ABC'	("ABC" = "ABC")
Success ✓	Check that this function returns '123'	("123" = "123")
Success ✓	Result should contain 5 rows	(5 = 5)
Success ✓	Values should be equal (using a let statement)	("Hello World" = "Hello World")
Success ✓	Returns host from URI	("https://bing.com" = "https://bing.com")
Success ✓	Handles port number appropriately	("https://bing.com:8080" = "https://bing.com:8080")

Helper Functions

10 minutes to read • [Edit Online](#)

This topic contains a number of helper functions commonly used in M extensions. These functions may eventually be moved to the official M library, but for now can be copied into your extension file code. You shouldn't mark any of these functions as `shared` within your extension code.

Navigation Tables

Table.ToTable

This function adds the table type metadata needed for your extension to return a table value that Power Query can recognize as a Navigation Tree. See [Navigation Tables](#) for more information.

```
Table.ToTable = (
    table as table,
    keyColumns as list,
    nameColumn as text,
    dataColumn as text,
    itemKindColumn as text,
    itemNameColumn as text,
    isLeafColumn as text
) as table =>
    let
        tableType = Value.Type(table),
        newTableType = Type.AddTableKey(tableType, keyColumns, true) meta
        [
            NavigationTable.NameColumn = nameColumn,
            NavigationTable.DataColumn = dataColumn,
            NavigationTable.ItemKindColumn = itemKindColumn,
            Preview.DelayColumn = itemNameColumn,
            NavigationTable.IsLeafColumn = isLeafColumn
        ],
        navigationTable = Value.ReplaceType(table, newTableType)
    in
        navigationTable;
```

PARAMETER	DETAILS
table	Your navigation table.
keyColumns	List of column names that act as the primary key for your navigation table.
nameColumn	The name of the column that should be used as the display name in the navigator.
dataColumn	The name of the column that contains the Table or Function to display.
itemKindColumn	The name of the column to use to determine the type of icon to display. Valid values for the column are <code>Table</code> and <code>Function</code> .

PARAMETER	DETAILS
itemNameColumn	The name of the column to use to determine the type of tooltip to display. Valid values for the column are <code>Table</code> and <code>Function</code> .
isLeafColumn	The name of the column used to determine if this is a leaf node, or if the node can be expanded to contain another navigation table.

Example usage:

```
shared MyExtension.Contents = () =>
    let
        objects = #table(
            {"Name", "Key", "Data", "ItemKind", "ItemName", "IsLeaf"}, {
                {"Item1", "item1", #table({ "Column1"}, { "Item1"}), "Table", "Table", true},
                {"Item2", "item2", #table({ "Column1"}, { "Item2"}), "Table", "Table", true},
                {"Item3", "item3", FunctionCallThatReturnsATable(), "Table", "Table", true},
                {"MyFunction", "myfunction", AnotherFunction.Contents(), "Function", "Function", true}
            })
        NavTable = Table.ToNavigationTable(objects, {"Key"}, "Name", "Data", "ItemKind", "ItemName", "IsLeaf")
    in
        NavTable;
```

URI Manipulation

Uri.FromParts

This function constructs a full URL based on individual fields in the record. It acts as the reverse of [Uri.Parts](#).

```
Uri.FromParts = (parts) =>
    let
        port = if (parts[Scheme] = "https" and parts[Port] = 443) or (parts[Scheme] = "http" and parts[Port] = 80) then "" else ":" & Text.From(parts[Port]),
        div1 = if Record.FieldCount(parts[Query]) > 0 then "?" else "",
        div2 = if Text.Length(parts[Fragment]) > 0 then "#" else "",
        uri = Text.Combine({parts[Scheme], "://", parts[Host], port, parts[Path], div1,
        Uri.BuildQueryString(parts[Query]), div2, parts[Fragment]})
```

in

uri;

Uri.GetHost

This function returns the scheme, host, and default port (for HTTP/HTTPS) for a given URL. For example,

`https://bing.com/subpath/query?param=1¶m2=hello` would become `https://bing.com:443`.

This is particularly useful for building `ResourcePath`.

```
Uri.GetHost = (url) =>
    let
        parts = Uri.Parts(url),
        port = if (parts[Scheme] = "https" and parts[Port] = 443) or (parts[Scheme] = "http" and parts[Port] = 80) then "" else ":" & Text.From(parts[Port])
    in
        parts[Scheme] & "://" & parts[Host] & port;
```

ValidateUrlScheme

This function checks if the user entered an HTTPS URL and raises an error if they don't. This is required for user entered URLs for certified connectors.

```
ValidateUrlScheme = (url as text) as text => if (Uri.Parts(url)[Scheme] <> "https") then error "Url scheme must be HTTPS" else url;
```

To apply it, just wrap your `url` parameter in your data access function.

```
DataAccessFunction = (url as text) as table =>
let
    _url = ValidateUrlScheme(url),
    source = Web.Contents(_url)
in
    source;
```

Retrieving Data

Value.WaitFor

This function is useful when making an asynchronous HTTP request and you need to poll the server until the request is complete.

```
Value.WaitFor = (producer as function, interval as function, optional count as number) as any =>
let
    list = List.Generate(
        () => {0, null},
        (state) => state{0} <> null and (count = null or state{0} < count),
        (state) => if state{1} <> null then {null, state{1}} else {1 + state{0}, Function.InvokeAfter(() => producer(state{0}), interval(state{0}))},
        (state) => state{1})
in
    List.Last(list);
```

Table.GenerateByPage

This function is used when an API returns data in an incremental/paged format, which is common for many REST APIs. The `getNextPage` argument is a function that takes in a single parameter, which will be the result of the previous call to `getNextPage`, and should return a `nullable table`.

```
getNextPage = (lastPage) as nullable table => ...`
```

`getNextPage` is called repeatedly until it returns `null`. The function will collate all pages into a single table. When the result of the first call to `getNextPage` is null, an empty table is returned.

```

// The getNextPage function takes a single argument and is expected to return a nullable table
Table.GenerateByPage = (getNextPage as function) as table =>
    let
        listOfPages = List.Generate(
            () => getNextPage(null),           // get the first page of data
            (lastPage) => lastPage <> null,    // stop when the function returns null
            (lastPage) => getNextPage(lastPage) // pass the previous page to the next function call
        ),
        // concatenate the pages together
        tableOfPages = Table.FromList(listOfPages, Splitter.SplitByNothing(), {"Column1"}),
        firstRow = tableOfPages{0}?
    in
        // if we didn't get back any pages of data, return an empty table
        // otherwise set the table type based on the columns of the first page
        if (firstRow = null) then
            Table.FromRows({})
        else
            Value.ReplaceType(
                Table.ExpandTableColumn(tableOfPages, "Column1", Table.ColumnNames(firstRow[Column1])),
                Value.Type(firstRow[Column1])
            );

```

Additional notes:

- The `getNextPage` function will need to retrieve the next page URL (or page number, or whatever other values are used to implement the paging logic). This is generally done by adding `meta` values to the page before returning it.
- The columns and table type of the combined table (that is, all pages together) are derived from the first page of data. The `getNextPage` function should normalize each page of data.
- The first call to `getNextPage` receives a null parameter.
- `getNextPage` must return null when there are no pages left.

An example of using this function can be found in the [Github sample](#), and the [TripPin paging sample](#).

```

Github.PagedTable = (url as text) => Table.GenerateByPage((previous) =>
    let
        // If we have a previous page, get its Next link from metadata on the page.
        next = if (previous <> null) then Value.Metadata(previous)[Next] else null,
        // If we have a next link, use it, otherwise use the original URL that was passed in.
        urlToUse = if (next <> null) then next else url,
        // If we have a previous page, but don't have a next link, then we're done paging.
        // Otherwise retrieve the next page.
        current = if (previous <> null and next = null) then null else Github.Contents(urlToUse),
        // If we got data back from the current page, get the link for the next page
        link = if (current <> null) then Value.Metadata(current)[Next] else null
    in
        current meta [Next=link];

```

SchemaTransformTable

```

EnforceSchema.Strict = 1;           // Add any missing columns, remove extra columns, set table type
EnforceSchema.IgnoreExtraColumns = 2; // Add missing columns, do not remove extra columns
EnforceSchema.IgnoreMissingColumns = 3; // Do not add or remove columns

SchemaTransformTable = (table as table, schema as table, optional enforceSchema as number) as table =>
    let
        // Default to EnforceSchema.Strict
        _enforceSchema = if (enforceSchema <> null) then enforceSchema else EnforceSchema.Strict,

        // Applies type transforms to a given table
        EnforceTypes = (table as table, schema as table) as table =>
            let
                map = (t) => if Type.Is(t, type list) or Type.Is(t, type record) or t = type any then null
            else t,
                mapped = Table.TransformColumns(schema, {"Type", map}),
                omitted = Table.SelectRows(mapped, each [Type] <> null),
                existingColumns = Table.ColumnNames(table),
                removeMissing = Table.SelectRows(omitted, each List.Contains(existingColumns, [Name])),
                primitiveTransforms = Table.ToRows(removeMissing),
                changedPrimitives = Table.TransformColumnTypes(table, primitiveTransforms)
            in
                changedPrimitives,

            // Returns the table type for a given schema
            SchemaToTableType = (schema as table) as type =>
                let
                    toList = List.Transform(schema[Type], (t) => [Type=t, Optional=false]),
                    toRecord = Record.FromList(toList, schema[Name]),
                    toType = Type.ForRecord(toRecord, false)
                in
                    type table (toType),

                // Determine if we have extra/missing columns.
                // The enforceSchema parameter determines what we do about them.
                schemaNames = schema[Name],
                foundNames = Table.ColumnNames(table),
                addNames = List.RemoveItems(schemaNames, foundNames),
                extraNames = List.RemoveItems(foundNames, schemaNames),
                tmp = Text.NewGuid(),
                added = Table.AddColumn(table, tmp, each []),
                expanded = Table.ExpandRecordColumn(added, tmp, addNames),
                result = if List.IsEmpty(addNames) then table else expanded,
                fullList =
                    if (_enforceSchema = EnforceSchema.Strict) then
                        schemaNames
                    else if (_enforceSchema = EnforceSchema.IgnoreMissingColumns) then
                        foundNames
                    else
                        schemaNames & extraNames,

                // Select the final list of columns.
                // These will be ordered according to the schema table.
                reordered = Table.SelectColumns(result, fullList, MissingField.Ignore),
                enforcedTypes = EnforceTypes(reordered, schema),
                withType = if (_enforceSchema = EnforceSchema.Strict) then Value.ReplaceType(enforcedTypes,
SchemaToTableType(schema)) else enforcedTypes
            in
                withType;

```

Table.ChangeType

```

let
    // table should be an actual Table.Type, or a List.Type of Records
    Table.ChangeType = (table, tableType as type) as nullable table =>
        // we only operate on table types
        if (not Type.Is(tableType, type table)) then error "type argument should be a table type" else

```

```

// if we have a null value, just return it
if (table = null) then table else
let
    columnsForType = Type.RecordFields(Type.TableRow(tableType)),
    columnsAsTable = Record.ToTable(columnsForType),
    schema = Table.ExpandRecordColumn(columnsAsTable, "Value", {"Type"}, {"Type"}),
    previousMeta = Value.Metadata(tableType),

    // make sure we have a table
    parameterType = Value.Type(table),
    _table =
        if (Type.Is(parameterType, type table)) then table
        else if (Type.Is(parameterType, type list)) then
            let
                asTable = Table.FromList(table, Splitter.SplitByNothing(), {"Column1"}),
                firstValueType = Value.Type(Table.FirstValue(asTable, null)),
                result =
                    // if the member is a record (as expected), then expand it.
                    if (Type.Is(firstValueType, type record)) then
                        Table.ExpandRecordColumn(asTable, "Column1", schema[Name])
                    else
                        error Error.Record("Error.Parameter", "table argument is a list, but not a
list of records", [ValueType = firstValueType ])
            in
                if (List.IsEmpty(table)) then
                    #table({"a"}, {})
                else result
        else
            error Error.Record("Error.Parameter", "table argument should be a table or list of
records", [ValueType = parameterType]),

    reordered = Table.SelectColumns(_table, schema[Name], MissingField.UseNull),

    // process primitive values - this will call Table.TransformColumnTypes
    map = (t) => if Type.Is(t, type table) or Type.Is(t, type list) or Type.Is(t, type record) or t =
type any then null else t,
    mapped = Table.TransformColumns(schema, {"Type", map}),
    omitted = Table.SelectRows(mapped, each [Type] <> null),
    existingColumns = Table.ColumnNames(reordered),
    removeMissing = Table.SelectRows(omitted, each List.Contains(existingColumns, [Name])),
    primativeTransforms = Table.ToRows(removeMissing),
    changedPrimatives = Table.TransformColumnTypes(reordered, primativeTransforms),

    // Get the list of transforms we'll use for Record types
    recordColumns = Table.SelectRows(schema, each Type.Is([Type], type record)),
    recordTypeTransformations = Table.AddColumn(recordColumns, "RecordTransformations", each (r) =>
Record.ChangeType(r, [Type]), type function),
    recordChanges = Table.ToRows(Table.SelectColumns(recordTypeTransformations, {"Name",
"RecordTransformations"})),

    // Get the list of transforms we'll use for List types
    listColumns = Table.SelectRows(schema, each Type.Is([Type], type list)),
    listTransforms = Table.AddColumn(listColumns, "ListTransformations", each (t) =>
List.ChangeType(t, [Type]), Function.Type),
    listChanges = Table.ToRows(Table.SelectColumns(listTransforms, {"Name", "ListTransformations"})),

    // Get the list of transforms we'll use for Table types
    tableColumns = Table.SelectRows(schema, each Type.Is([Type], type table)),
    tableTransforms = Table.AddColumn(tableColumns, "TableTransformations", each (t) =>
@Table.ChangeType(t, [Type]), Function.Type),
    tableChanges = Table.ToRows(Table.SelectColumns(tableTransforms, {"Name",
"TableTransformations"})),

    // Perform all of our transformations
    allColumnTransforms = recordChanges & listChanges & tableChanges,
    changedRecordTypes = if (List.IsEmpty(allColumnTransforms)) then changedPrimatives else
Table.TransformColumns(changedPrimatives, allColumnTransforms, null, MissingField.Ignore),

    // set final type

```

```

        withType = Value.ReplaceType(changedRecordTypes, tableType)
    in
        if (List.IsEmpty(Record.FieldNames(columnsForType))) then table else withType meta previousMeta,
    end

    // If given a generic record type (no predefined fields), the original record is returned
    Record.ChangeType = (record as record, recordType as type) =>
        let
            // record field format is [ fieldName = [ Type = type, Optional = logical], ... ]
            fields = try Type.RecordFields(recordType) otherwise error "Record.ChangeType: failed to get
            record fields. Is this a record type?",
            fieldNames = Record.FieldNames(fields),
            fieldTable = Record.ToTable(fields),
            optionalFields = Table.SelectRows(fieldTable, each [Value][Optional])[Name],
            requiredFields = List.Difference(fieldNames, optionalFields),
            // make sure all required fields exist
            withRequired = Record.SelectFields(record, requiredFields, MissingField.UseNull),
            // append optional fields
            withOptional = withRequired & Record.SelectFields(record, optionalFields, MissingField.Ignore),
            // set types
            transforms = GetTransformsForType(recordType),
            withTypes = Record.TransformFields(withOptional, transforms, MissingField.Ignore),
            // order the same as the record type
            reorder = Record.ReorderFields(withTypes, fieldNames, MissingField.Ignore)
        in
            if (List.IsEmpty(fieldNames)) then record else reorder,
        end

    List.ChangeType = (list as list, listType as type) =>
        if (not Type.Is(listType, type list)) then error "type argument should be a list type" else
        let
            listItemType = Type.ListItem(listType),
            transform = GetTransformByType(listItemType),
            modifiedValues = List.Transform(list, transform),
            typed = Value.ReplaceType(modifiedValues, listType)
        in
            typed,
        end

    // Returns a table type for the provided schema table
    Schema.ToTableType = (schema as table) as type =>
        let
            toList = List.Transform(schema[Type], (t) => [Type=t, Optional=false]),
            toRecord = Record.FromList(toList, schema[Name]),
            toType = Type.ForRecord(toRecord, false),
            previousMeta = Value.Metadata(schema)
        in
            type table (toType) meta previousMeta,
        end

    // Returns a list of transformations that can be passed to Table.TransformColumns, or
    Record.TransformFields
    // Format: {"Column", (f) => ...} .... ex: {"A", Number.From}
    GetTransformsForType = (_type as type) as list =>
        let
            fieldsOrColumns = if (Type.Is(_type, type record)) then Type.RecordFields(_type)
                else if (Type.Is(_type, type table)) then Type.RecordFields(Type.TableRow(_type))
                else error "GetTransformsForType: record or table type expected",
            toTable = Record.ToTable(fieldsOrColumns),
            transformColumn = Table.AddColumn(toTable, "Transform", each GetTransformByType([Value][Type]),
            Function.Type),
            transformMap = Table.ToRows(Table.SelectColumns(transformColumn, {"Name", "Transform"}))
        in
            transformMap,
        end

    GetTransformByType = (_type as type) as function =>
        if (Type.Is(_type, type number)) then Number.From
        else if (Type.Is(_type, type text)) then Text.From
        else if (Type.Is(_type, type date)) then Date.From
        else if (Type.Is(_type, type datetime)) then DateTime.From
        else if (Type.Is(_type, type duration)) then Duration.From
        else if (Type.Is(_type, type datetimezone)) then DateTimeZone.From
        else if (Type.Is(_type, type logical)) then Logical.From
    end

```

```
        else if (Type.Is(_type, type time)) then Time.From
        else if (Type.Is(_type, type record)) then (t) => if (t <> null) then @Record.ChangeType(t, _type)
else t
    else if (Type.Is(_type, type table)) then (t) => if (t <> null) then @Table.ChangeType(t, _type) else
t
    else if (Type.Is(_type, type list)) then (t) => if (t <> null) then @List.ChangeType(t, _type) else t
else (t) => t
in
Table.ChangeType
```

Adding Function Documentation

3 minutes to read • [Edit Online](#)

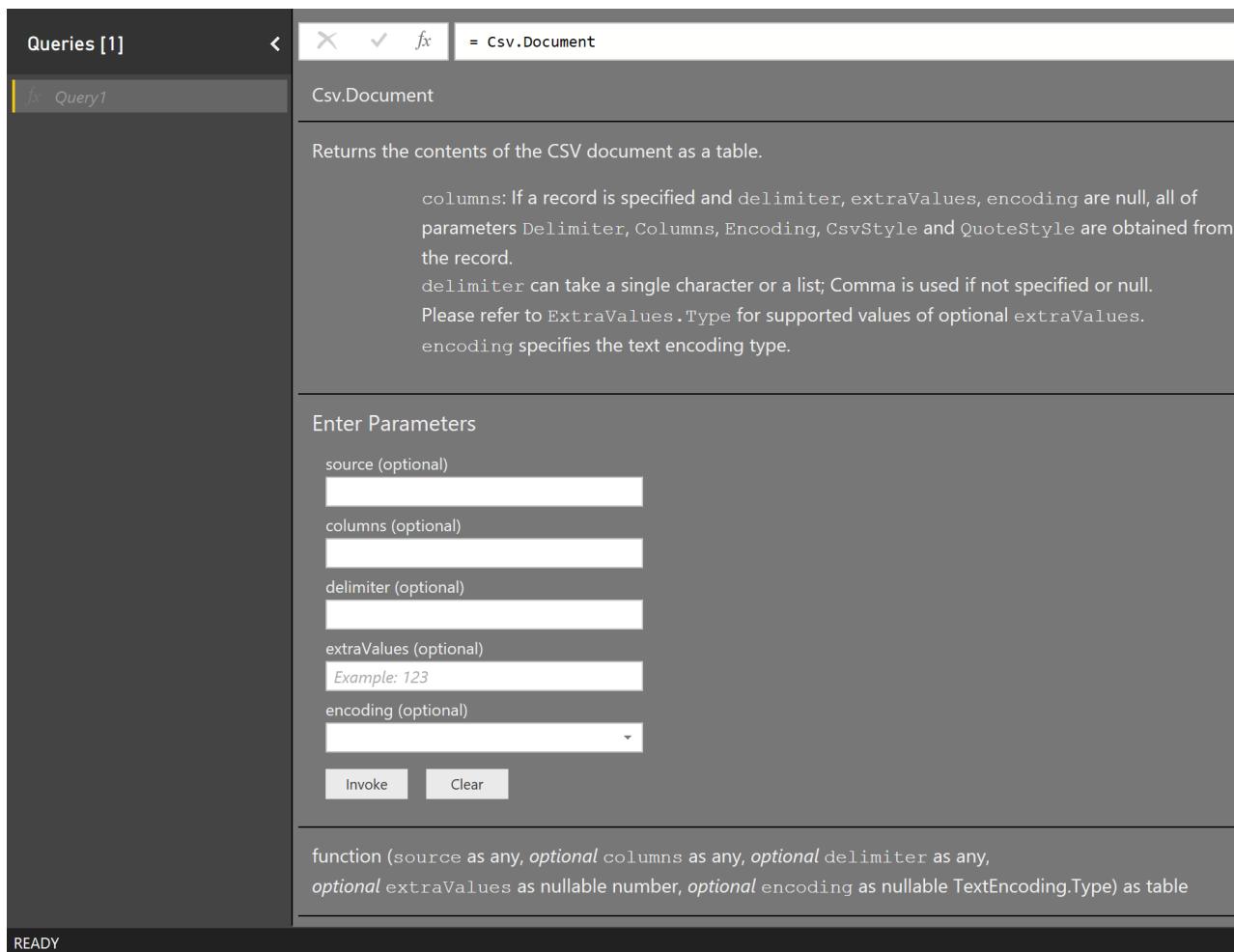
Power Query will automatically generate an invocation UI for you based on the arguments for your function. By default, this UI will contain the name of your function, and an input for each of your parameters.



Similarly, evaluating the name of your function, without specifying parameters, will display information about it.



You might notice that built-in functions typically provide a better user experience, with descriptions, tooltips, and even sample values. You can take advantage of this same mechanism by defining specific meta values on your function type. This topic describes the meta fields that are used by Power Query, and how you can make use of them in your extensions.



Function Types

You can provide documentation for your function by defining custom *type* values. The process looks like this:

1. Define a type for each parameter.
2. Define a type for your function.
3. Add various `Documentation.*` fields to your types metadata record.
4. Call `Value.ReplaceType` to ascribe the type to your shared function.

You can find more information about types and metadata values in the [M Language Specification](#).

Using this approach allows you to supply descriptions and display names for your function, as well as individual parameters. You can also supply sample values for parameters, as well as defining a preset list of values (turning the default text box control into a drop down).

The Power Query experience retrieves documentation from meta values on the type of your function, using a combination of calls to `Value.Type`, `Type.FunctionParameters`, and `Value.Metadata`.

Function Documentation

The following table lists the Documentation fields that can be set in the metadata for your *function*. All fields are optional.

FIELD	TYPE	DETAILS
-------	------	---------

FIELD	TYPE	DETAILS
Documentation.Examples	list	List of record objects with example usage of the function. Only displayed as part of the function info. Each record should contain the following optional text fields: <code>Description</code> , <code>Code</code> , and <code>Result</code> .
Documentation.LongDescription	text	Full description of what the function does, displayed in the function info.
Documentation.Name	text	Text to display across the top of the function invocation dialog.

Parameter Documentation

The following table lists the Documentation fields that can be set in the metadata for your *function parameters*. All fields are optional.

FIELD	TYPE	DETAILS
Documentation.AllowedValues	list	List of valid values for this parameter. Providing this field will change the input from a textbox to a drop down list. Note, this doesn't prevent a user from manually editing the query to supply alternative values.
Documentation.FieldCaption	text	Friendly display name to use for the parameter.
Documentation.FieldDescription	text	Description to show next to the display name.
Documentation.SampleValues	list	List of sample values to be displayed (as faded text) inside of the text box.

Example

The following code snippet (and resulting dialogs) are from the [HelloWorldWithDocs](#) sample.

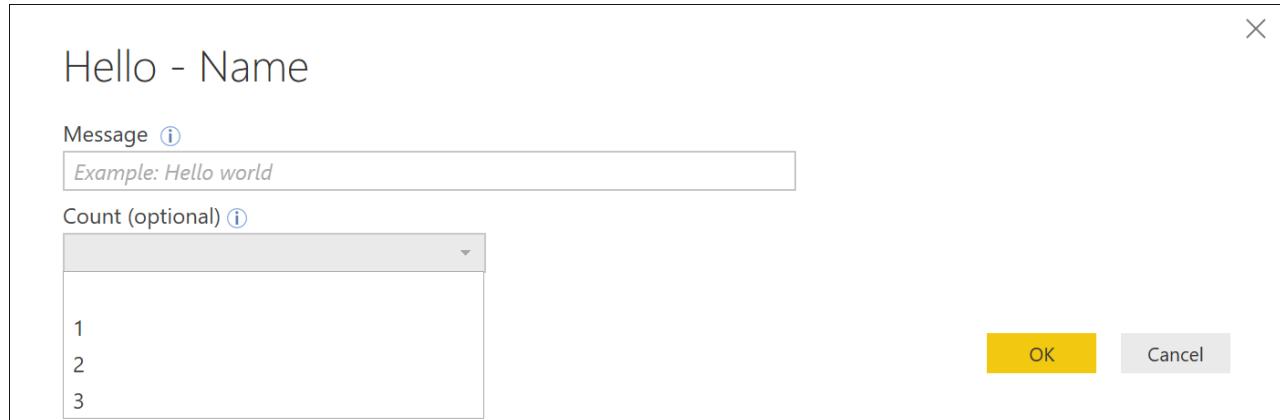
```
[DataSource.Kind="HelloWorldWithDocs", Publish="HelloWorldWithDocs.Publish"]
shared HelloWorldWithDocs.Contents = Value.ReplaceType(HelloWorldImpl, HelloWorldType);

HelloWorldType = type function (
    message as (type text meta [
        Documentation.FieldCaption = "Message",
        Documentation.FieldDescription = "Text to display",
        Documentation.SampleValues = {"Hello world", "Hola mundo"}
    ]),
    optional count as (type number meta [
        Documentation.FieldCaption = "Count",
        Documentation.FieldDescription = "Number of times to repeat the message",
        Documentation.AllowedValues = { 1, 2, 3 }
    ])
)
as table meta [
    Documentation.Name = "Hello - Name",
    Documentation.LongDescription = "Hello - Long Description",
    Documentation.Examples = {[[
        Description = "Returns a table with 'Hello world' repeated 2 times",
        Code = "HelloWorldWithDocs.Contents(\"Hello world\", 2)",
        Result = "#table({\"Column1\"}, {\"Hello world\"}, {\"Hello world\"})"
    ], [
        Description = "Another example, new message, new count!",
        Code = "HelloWorldWithDocs.Contents(\"Goodbye\", 1)",
        Result = "#table({\"Column1\"}, {\"Goodbye\"})"
    ]}]
];
];

HelloWorldImpl = (message as text, optional count as number) as table =>
let
    _count = if (count <> null) then count else 5,
    listOfMessages = List.Repeat({message}, _count),
    table = Table.FromList(listOfMessages, Splitter.SplitByNothing())
in
    table;
```

This code results in the following dialogs in Power BI.

Function invocation



Function info

= HelloWorldWithDocs.Contents

Hello - Name

Hello - Long Description

Enter Parameters

Message
Example: Hello world

Count (optional)

Invoke Clear

function (message as text, *optional* count as nullable number) as table

Example: Returns a table with 'Hello world' repeated 2 times

Usage:
HelloWorldWithDocs.Contents("Hello world", 2)

Output:
`#table({ "Column1" }, { { "Hello world" }, { "Hello world" } })`

Example: Another example, new message, new count!

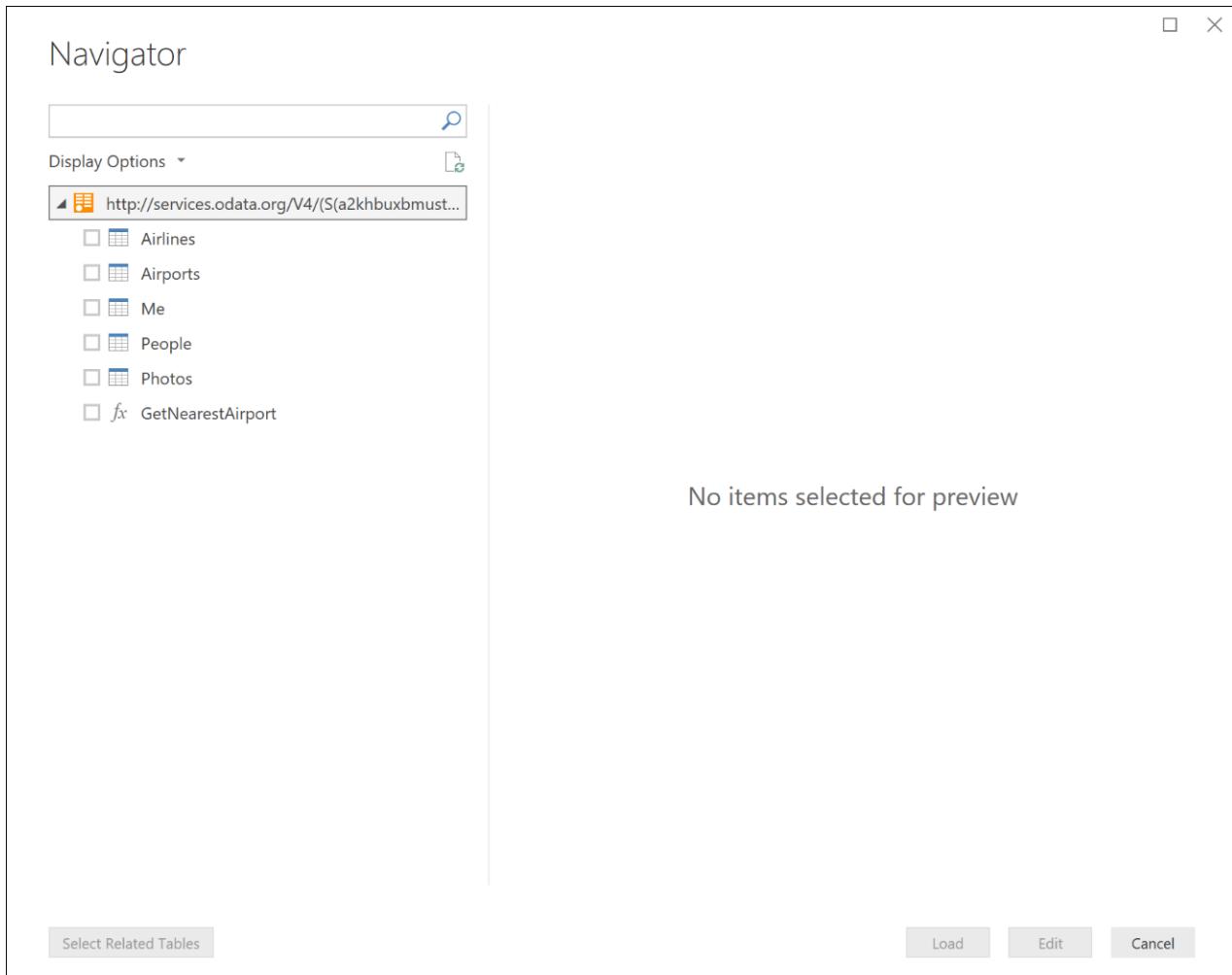
Usage:
HelloWorldWithDocs.Contents("Goodbye", 1)

Output:
`#table({ "Column1" }, { { "Goodbye" } })`

Handling Navigation

3 minutes to read • [Edit Online](#)

Navigation Tables (or nav tables) are a core part of providing a user-friendly experience for your connector. The Power Query experience displays them to the user after they've entered any required parameters for your data source function, and have authenticated with the data source.



Behind the scenes, a nav table is just a regular M Table value with specific metadata fields defined on its Type. When your data source function returns a table with these fields defined, Power Query will display the navigator dialog. You can actually see the underlying data as a Table value by right-clicking on the root node and selecting **Edit**.

Table.ToNavigationTable

You can use the `Table.ToNavigationTable` function to add the table type metadata needed to create a nav table.

NOTE

You currently need to copy and paste this function into your M extension. In the future it will likely be moved into the M standard library.

The following table describes the parameters for this function:

PARAMETER	DETAILS
table	Your navigation table.
keyColumns	List of column names that act as the primary key for your navigation table.
nameColumn	The name of the column that should be used as the display name in the navigator.
dataColumn	The name of the column that contains the Table or Function to display.
itemKindColumn	The name of the column to use to determine the type of icon to display. See below for the list of valid values for the column.
itemNameColumn	The name of the column to use to determine the preview behavior. This is typically set to the same value as itemKind.
isLeafColumn	The name of the column used to determine if this is a leaf node, or if the node can be expanded to contain another navigation table.

The function adds the following metadata to the table type:

FIELD	PARAMETER
NavigationTable.NameColumn	nameColumn
NavigationTable.DataColumn	dataColumn
NavigationTable.ItemKindColumn	itemKindColumn
NavigationTable.IsLeafColumn	isLeafColumn
Preview.DelayColumn	itemNameColumn

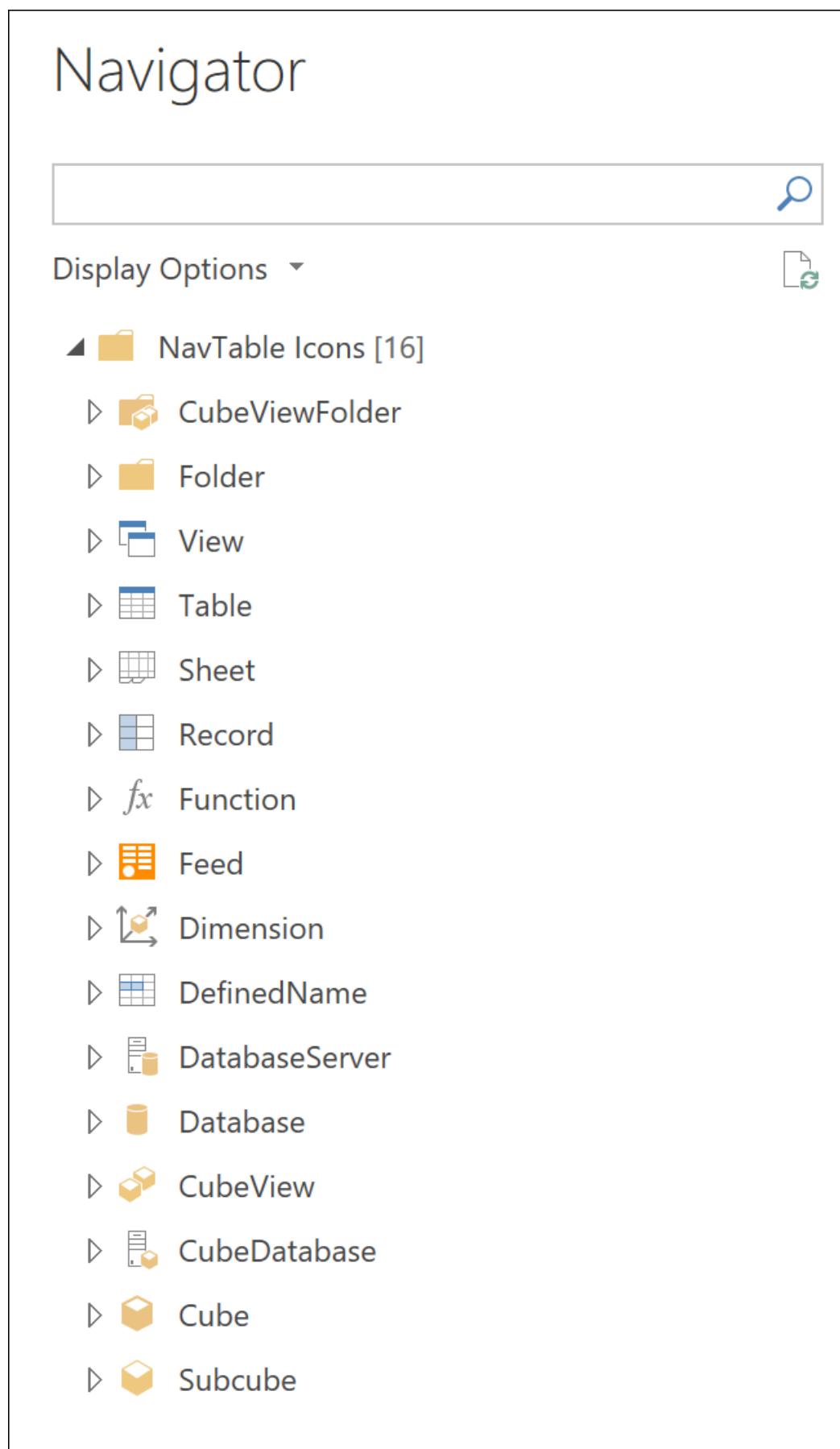
Values for ItemKind

Each of the following item kind values provide a different icon in the navigation table.

- Feed
- Cube
- CubeDatabase
- CubeView
- CubeViewFolder
- Database
- DatabaseServer
- Dimension
- Table
- Folder
- Function
- View

- Sheet
- Subcube
- DefinedName
- Record

The image below shows the icons for item kinds in Power BI Desktop.



Examples

Flat navigation table

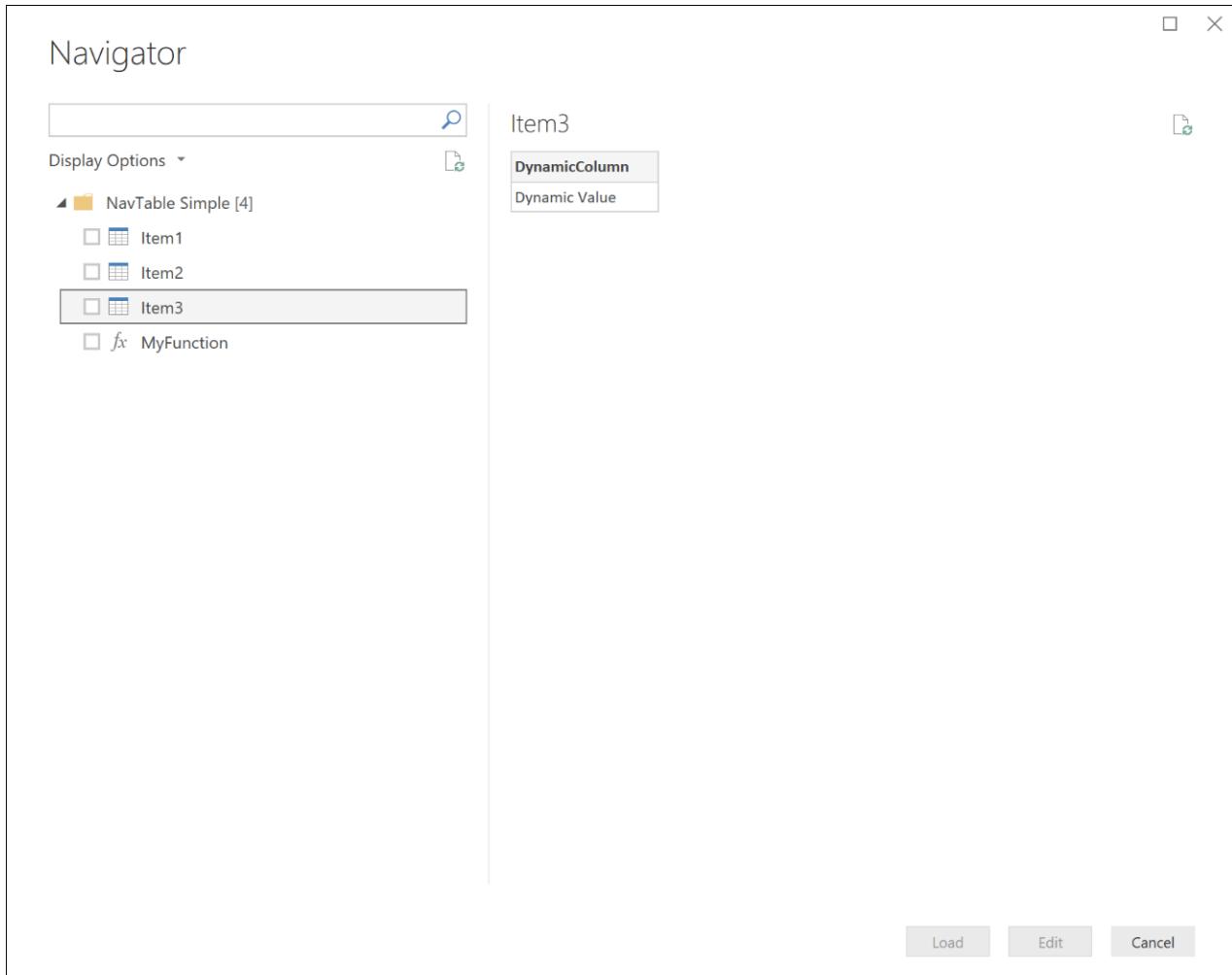
The following code sample displays a flat nav table with three tables and a function.

```
shared NavigationTable.Simple = () =>
let
    objects = #table(
        {"Name", "Key", "Data", "ItemKind", "ItemName", "IsLeaf"}, {
            {"Item1", "item1", #table({ "Column1"}, {{"Item1"}}, "Table", "Table", true),
            {"Item2", "item2", #table({ "Column1"}, {{"Item2"}}, "Table", "Table", true),
            {"Item3", "item3", FunctionCallThatReturnsATable(), "Table", "Table", true},
            {"MyFunction", "myfunction", AnotherFunction.Contents(), "Function", "Function", true}
        })
    NavTable = Table.ToNavigationTable(objects, {"Key"}, "Name", "Data", "ItemKind", "ItemName", "IsLeaf")
in
    NavTable;

shared FunctionCallThatReturnsATable = () =>
    #table({ "DynamicColumn"}, {{"Dynamic Value"}});

```

This code will result in the following Navigator display in Power BI Desktop:



Multi-level navigation table

It is possible to use nested navigation tables to create a hierarchical view over your data set. You do this by setting the `IsLeaf` value for that row to `false` (which marks it as a node that can be expanded), and format the `Data` column to also be another nav table.

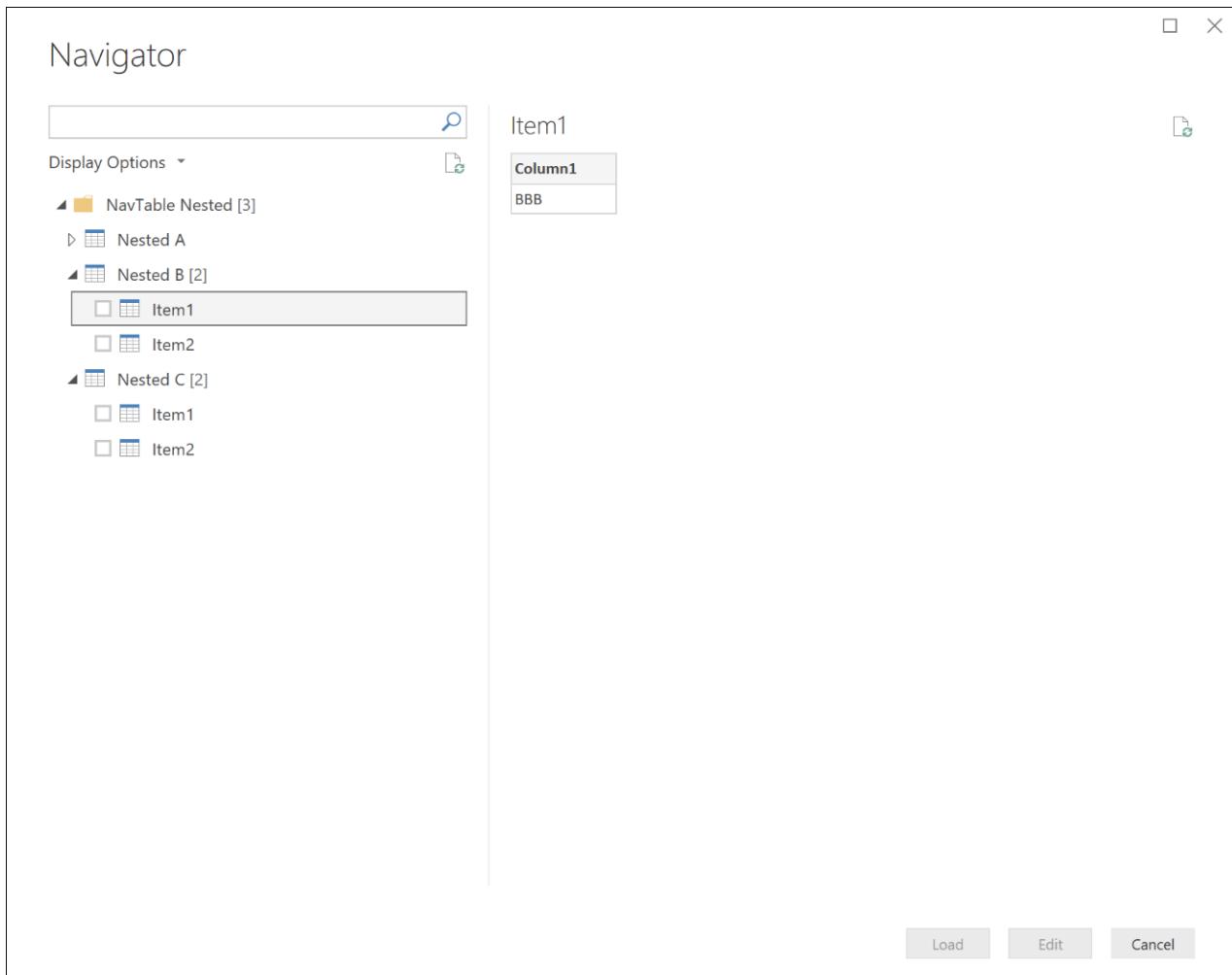
```

shared NavigationTable.Nested = () as table =>
let
    objects = #table(
        {"Name", "Key", "Data", "ItemKind", "ItemName", "IsLeaf"},{
            {"Nested A", "n1", CreateNavTable("AAA"), "Table", "Table", false},
            {"Nested B", "n2", CreateNavTable("BBB"), "Table", "Table", false},
            {"Nested C", "n3", CreateNavTable("CCC"), "Table", "Table", false}
        })
    NavTable = Table.ToNavigationTable(objects, {"Key"}, "Name", "Data", "ItemKind", "ItemName", "IsLeaf")
in
    NavTable;

CreateNavTable = (message as text) as table =>
let
    objects = #table(
        {"Name", "Key", "Data", "ItemKind", "ItemName", "IsLeaf"},{
            {"Item1", "item1", #table({{"Column1"}}, {{message}}), "Table", "Table", true},
            {"Item2", "item2", #table({{"Column1"}}, {{message}}), "Table", "Table", true}
        })
    NavTable = Table.ToNavigationTable(objects, {"Key"}, "Name", "Data", "ItemKind", "ItemName", "IsLeaf")
in
    NavTable;

```

This code would result in the following Navigator display in Power BI Desktop:



Dynamic Navigation Tables

More complex functionality can be built from these basics. While all of the above examples show hard-coded entities in the nav table, it's easy to see how a nav table could be generated dynamically based on entities that are available to a given user. A few key considerations for dynamic navigation tables include:

- [Error handling](#) to ensure a good experience for users that don't have access to certain endpoints.
- Node evaluation is lazy by default; leaf nodes are not evaluated until the parent node is expanded. Certain implementations of multi-level dynamic nav tables may result in eager evaluation of the entire tree. Be sure to monitor the number of calls that Power Query is making as it initially renders the navigation table. For example, `Table.InsertRows` is 'lazier' than `Table.FromRecords`, as it does not need to evaluate its arguments.

Handling Gateway Support

2 minutes to read • [Edit Online](#)

Test Connection

Custom Connector support is available in both Personal and Standard modes of the [on-premises data gateway](#). Both gateway modes support **Import. Direct Query** is only supported in Standard mode.

The method for implementing TestConnection functionality is likely to change while the Power BI Custom Data Connector functionality is in preview.

To support scheduled refresh through the on-premises data gateway, your connector **must** implement a TestConnection handler. The function is called when the user is configuring credentials for your source, and used to ensure they are valid. The TestConnection handler is set in the [Data Source Kind](#) record, and has the following signature:

```
(dataSourcePath) as list => ...
```

Where `dataSourcePath` is the [Data Source Path](#) value for your function, and the return value is a list composed of:

- The name of the function to call (this function must be marked as `#shared`, and is usually your primary data source function).
- One or more arguments to pass to your function.

If the invocation of the function results in an error, TestConnection is considered to have failed, and the credential won't be persisted.

NOTE

As stated above, the function name provided by TestConnection must be a `shared` member.

Example: Connector with no required arguments

The code snippet below implements TestConnection for a data source with no required parameters (such as the one found in the [TripPin tutorial](#)). Connectors with no required parameters (referred to as 'Singletons') do not need any user provided input to test a connection (other than credentials). In this case, the `dataSourcePath` value would be equal to the name of the Data Source Kind, and can be ignored. The `TripPin.Contents` function is invoked with no additional parameters.

```
TripPin = [
    TestConnection = (dataSourcePath) => { "TripPin.Contents" },
    Authentication = [
        Anonymous = []
    ],
    Label = "TripPin"
];
```

Example: Connector with a URL parameter

If your data source function has a single required parameter of the type `Uri.Type`, its `dataSourcePath` will be equal to the URL provided by the user. The snippet below shows the TestConnection implementation from the [Github](#)

Sample.

```
GithubSample = [
    TestConnection = (dataSourcePath) => {"GithubSample.Contents", dataSourcePath},
    Authentication = [
        OAuth = [
            StartLogin = StartLogin,
            FinishLogin = FinishLogin,
            Label = Extension.LoadString("AuthenticationLabel")
        ]
    ]
];
```

Example: Connector with required parameters

If your data source function has multiple parameters, or a single non-URL parameter, then the `dataSourcePath` value will be a JSON string containing the parameters. The snippet below comes from the [DirectQueryForSQL](#) sample.

```
DirectSQL = [
    TestConnection = (dataSourcePath) =>
        let
            json = Json.Document(dataSourcePath),
            server = json[server],
            database = json[database]
        in
            { "DirectSQL.Database", server, database },
    Authentication = [
        Windows = [],
        UsernamePassword = []
    ],
    Label = "Direct Query for SQL"
];
```

Handling Power Query Connector Signing

3 minutes to read • [Edit Online](#)

In Power BI, the loading of custom connectors is limited by your choice of security setting. As a general rule, when the security for loading custom connectors is set to 'Recommended', the custom connectors won't load at all, and you have to lower it to make them load.

The exception to this is trusted, 'signed connectors'. Signed connectors are a special format of custom connector, a .pqx instead of .mez file, which have been signed with a certificate. The signer can provide the user or the user's IT department with a thumbprint of the signature, which can be put into the registry to securely indicate trusting a given connector.

The following steps enable you to use a certificate (with explanation on how to generate one if you don't have one available) and sign a custom connector with the 'MakePQX' tool.

NOTE

If you need help creating a self-signed certificate to test these instructions, see the Microsoft documentation on [New-SelfSignedCertificate in PowerShell](#).

NOTE

If you need help exporting your certificate as a pfx, see [How to create a PKCS#12 \(PFX\) file on a Windows server](#).

1. Download [MakePQX](#).
2. Extract the MakePQX folder in the included zip to your desired target.
3. To run it, call MakePQX in the command-line. It requires the other libraries in the folder, so you can't copy just the one executable. Running without any parameters will return the help information.

Usage: **MakePQX [options] [command]**

Options:

OPTIONS	DESCRIPTION
-? -h --help	Show help information

Commands:

COMMAND	DESCRIPTION
pack	Create a .pqx file.
sign	Signs an unsigned pqx, or countersigns if pqx is already signed. Use the --replace option to replace the existing signature.
verify	Verify the signature status on a .pqx file. Return value will be non-zero if the signature is invalid.

There are three commands in MakePQX. Use **MakePQX [command] --help** for more information about a command.

Pack

The **Pack** command takes a .mez file and packs it into a .pinq file, which is able to be signed. The .pinq file is also able to support a number of capabilities that will be added in the future.

Usage: **MakePQX pack [options]**

Options:

OPTION	DESCRIPTION
-? -h --help	Show help information.
-mz --mez	Input extension file.
-c --certificate	Certificate (.pfx) used to sign the extension file.
-p --password	Password for the certificate file.
-t --target	Output file name. Defaults to the same name as the input file.

Example

```
C:\Users\cotope\Downloads\MakePQX>MakePQX.exe pack -mz  
"C:\Users\cotope\OneDrive\Documents\Power BI Desktop\Custom Connectors\HelloWorld.mez" -t  
"C:\Users\cotope\OneDrive\Documents\Power BI Desktop\Custom Connectors\HelloWorldSigned.pinq"
```

Sign

The **Sign** command signs your .pinq file with a certificate, giving it a thumbprint that can be checked for trust by Power BI clients with the higher security setting. This takes a pinq file and returns the same pinq file, signed.

Usage: **MakePQX sign [arguments] [options]**

Arguments:

ARGUMENT	DESCRIPTION
<pinq file>	The path to the .pinq file.

Options:

OPTION	DESCRIPTION
-c --certificate	Certificate (.pfx) used to sign the extension file.
-p --password	Password for the certificate file.
-r --replace	Replace existing signature instead of countersigning.
-? -h --help	Show help information.

Example

```
C:\Users\cotope\Downloads\MakePQX>MakePQX sign "C:\Users\cotope\OneDrive\Documents\Power BI Desktop\Custom Connectors\HelloWorldSigned.pqx" --certificate ColinPopellTestCertificate.pfx --password password
```

Verify

The **Verify** command verifies that your module has been properly signed, as well as showing the Certificate status.

Usage: **MakePQX verify [arguments] [options]**

Arguments:

ARGUMENT	DESCRIPTION
<pqx file>	The path to the .pqi file.

Options:

OPTION	DESCRIPTION
-q --quiet	Hides signature verification output.
-? -h --help	Show help information.

Example

```
C:\Users\cotope\Downloads\MakePQX>MakePQX verify "C:\Users\cotope\OneDrive\Documents\Power BI Desktop\Custom Connectors\HelloWorldSigned.pqx"
```

```
{
  "SignatureStatus": "Success",
  "CertificateStatus": [
    {
      "Issuer": "CN=Colin Popell",
      "Thumbprint": "16AF59E4BE5384CD860E230ED4AED474C2A3BC69",
      "Subject": "CN=Colin Popell",
      "NotBefore": "2019-02-14T22:47:42-08:00",
      "NotAfter": "2020-02-14T23:07:42-08:00",
      "Valid": false,
      "Parent": null,
      "Status": "UntrustedRoot"
    }
  ]
}
```

Trusting signed connectors in Power BI Desktop

Once you've verified your signature, you can provide the thumbprint to the end-user to list as trusted. You can read about how to do this in the [Power BI Documentation](#).