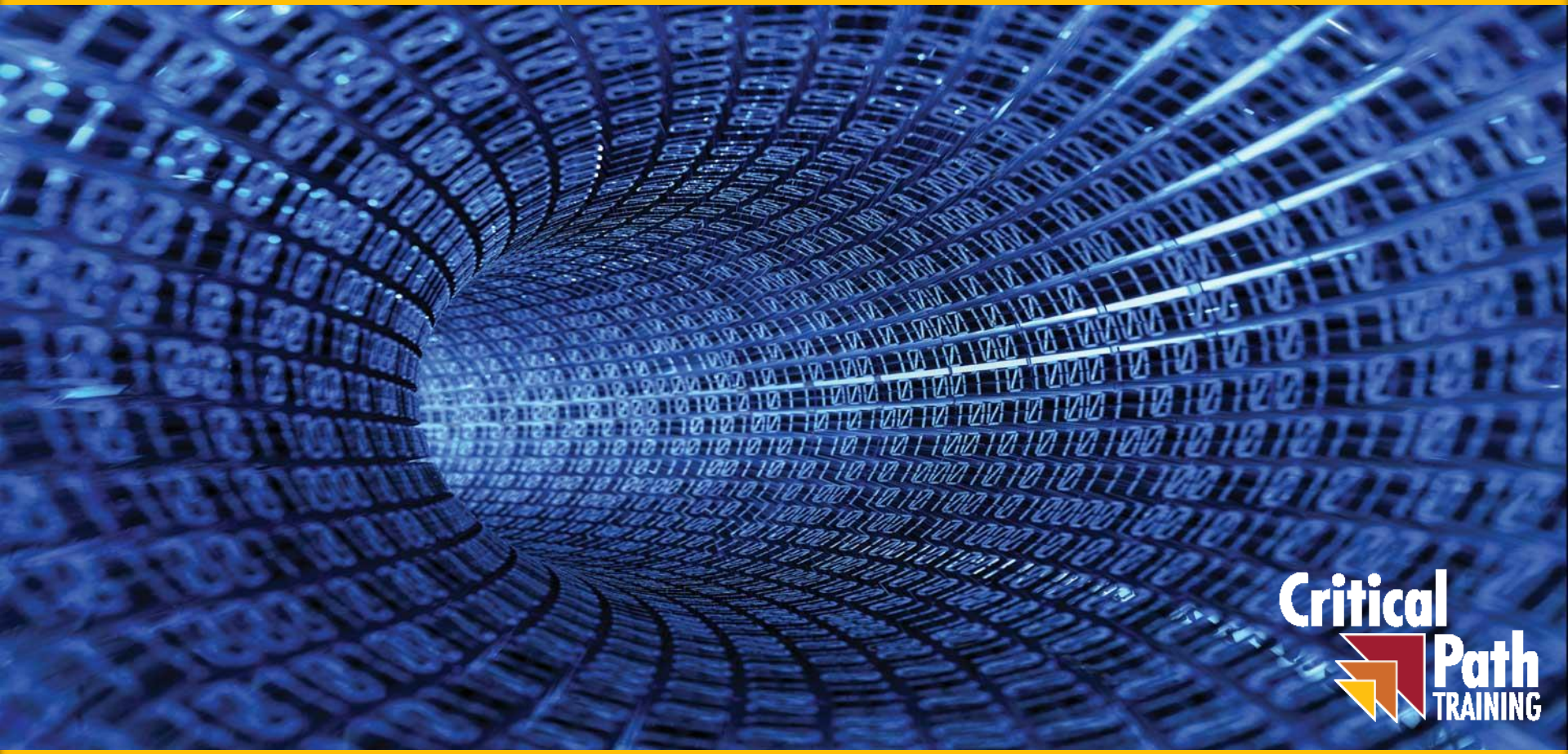
Programming in M and Developing Custom Connectors

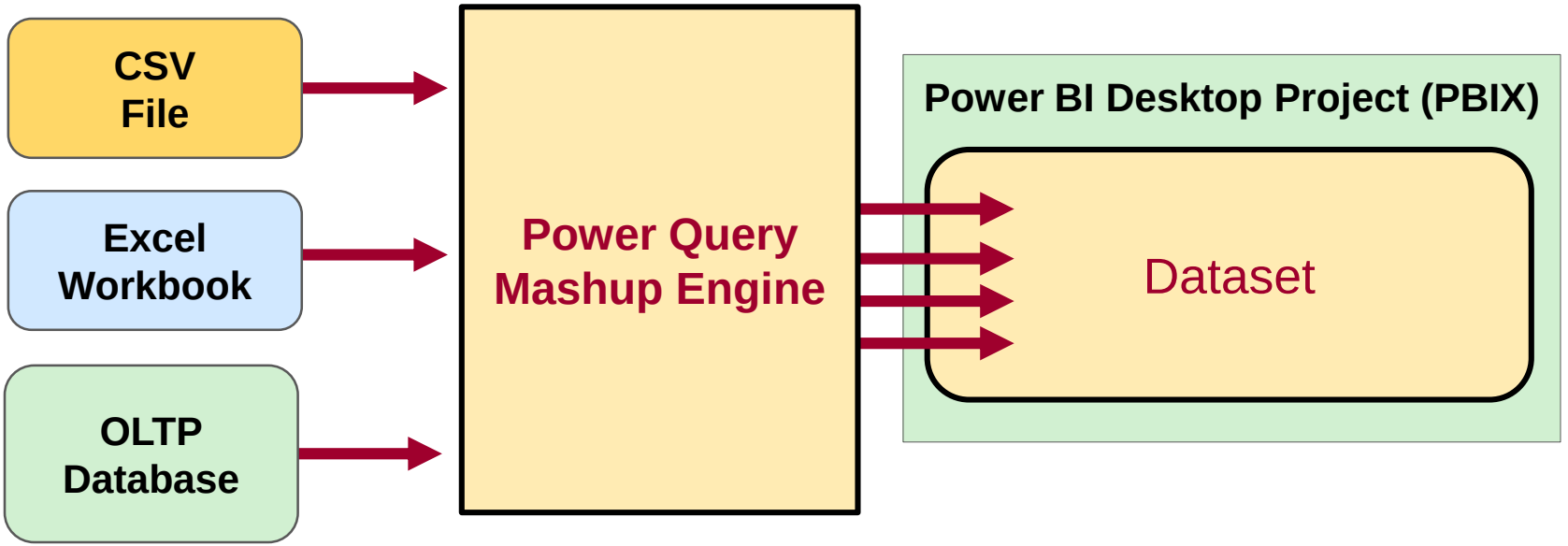


Agenda

- Power Query Mashup Engine
- M Programming Fundamentals
- M Function Library
- Query Functions
- Query Parameters
- Custom Data Connectors

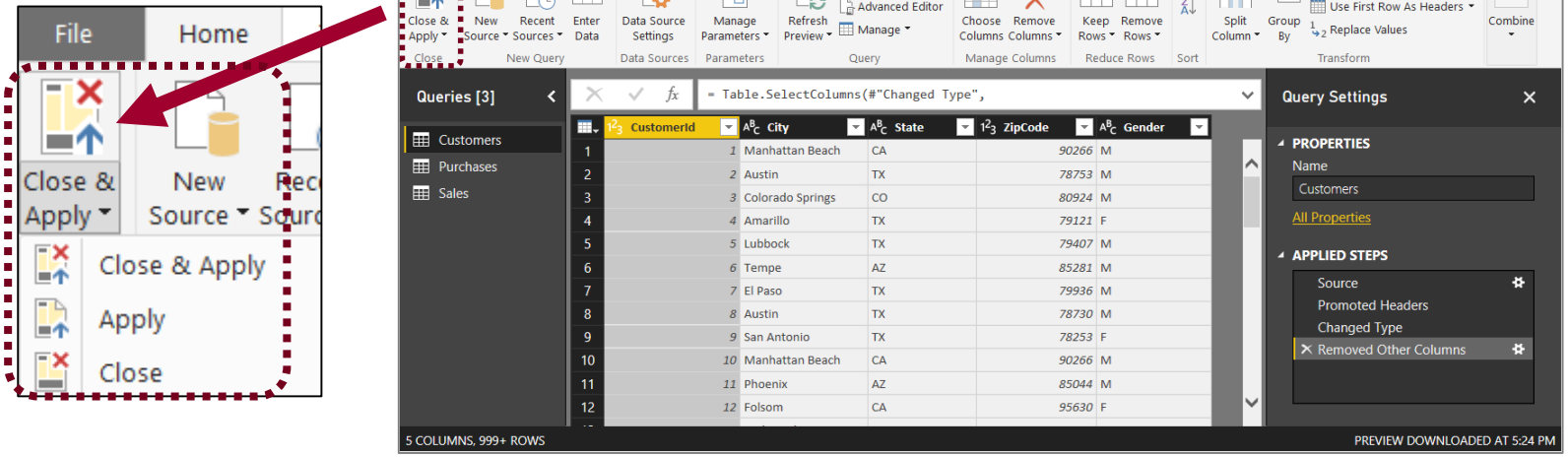
Power BI Desktop is an ETL Tool

- ETL process is essential part of any BI Project
  - **Extract** the data from wherever it lives
  - **Transform** the shape of the data for better analysis
  - **Load** the data into dataset for analysis and reporting



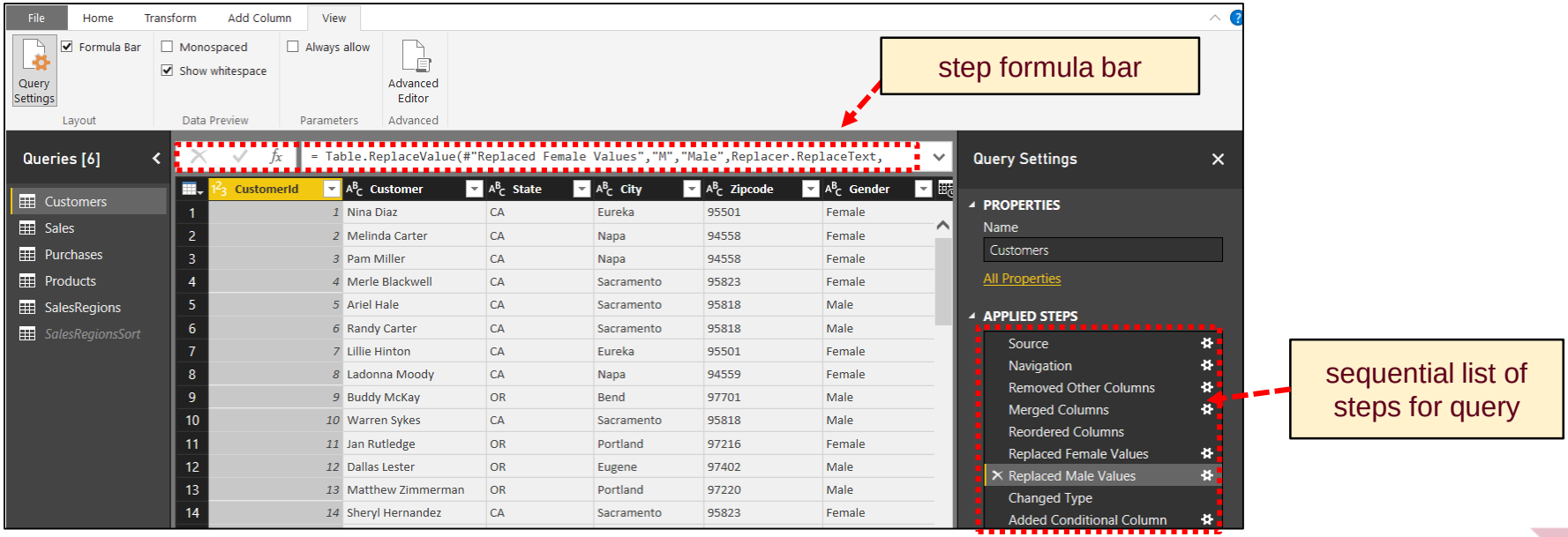
Query Editor Window

- Power BI Desktop provides separate Query Editor window
  - Provides easy-to-use UI experience for designing queries
  - Queries created by creating **Applied Steps**
  - Preview of table generated by query output shown in the middle
  - Query can be executed using **Apply** or **Close & Apply** command



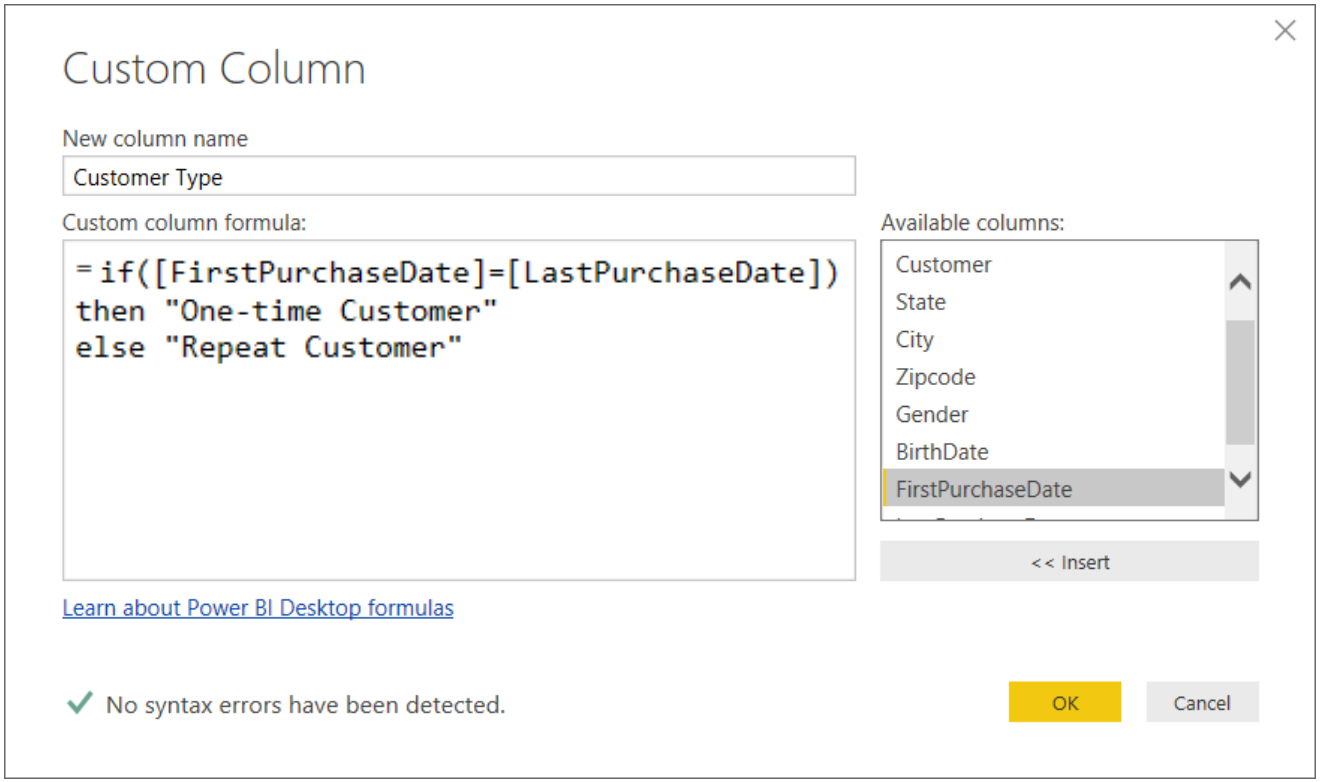
Query Steps

- A query is created as a sequence of steps
  - Each step is a parameterized operation in data processing pipeline
  - Query starts with Source step to extract data from a data source
  - Additional steps added to perform transform operations on data
  - Each step is recorded using M *(aka Power Query Formula Language)*



Custom Column Dialog

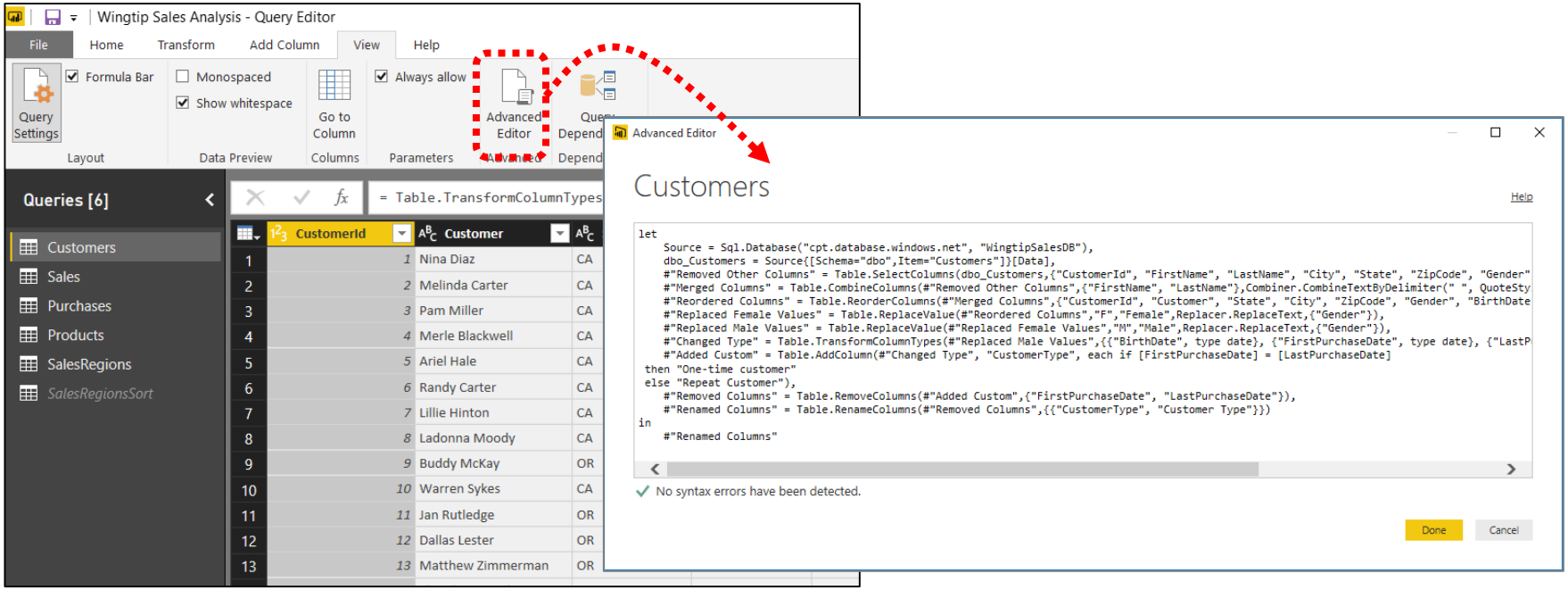
- You can write M code directly for custom column
  - The Custom Column dialog provides a simple M code editor



Advanced Editor

or more correctly - The Simple Editor for Advanced Users

- Power BI Desktop based on "M" functional language
  - Query in Power BI Desktop saved as set of M statements in code
  - Query Editor generates code in M behind the scenes
  - Advanced users can view & modify query code in Advanced Editor



Why Learn M

- Accomplish things that cannot be done in query editor
  - Working with query functions
  - Performing calculations across rows
  - Navigate to SharePoint list by list title instead of GUID with the ID
- Author queries and check them into source control system
  - Add query logic in .m files and store them in GitHub, TFS, etc.
  - Ensure query logic is the same across PBIX projects
- Stay Ahead of the Pack and Win Admiration of Your Peers
  - People will think you are buddies with Chris Webb!

Agenda

- ✓ Power Query Mashup Engine
- ➢ M Programming Fundamentals
- M Function Library
- Query Functions
- Query Parameters
- Custom Data Connectors

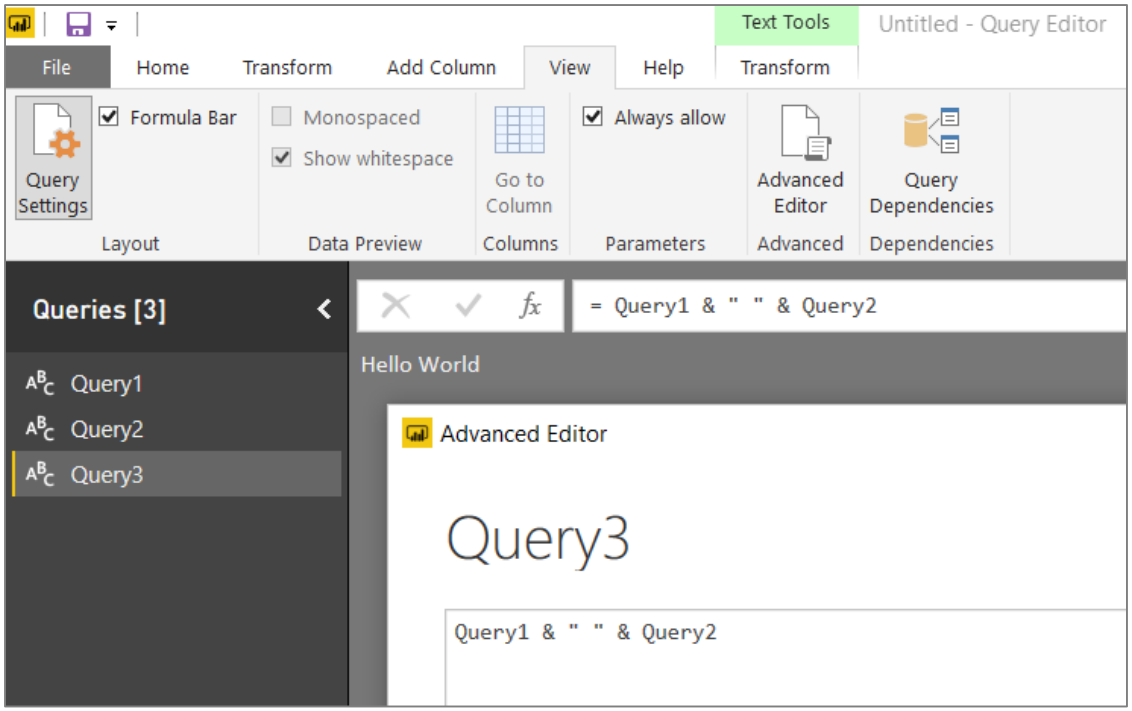
The M Programming Language

- M is a ***functional*** programming language
  - computation through evaluation of mathematical functions
  - Programming involves writing expressions instead of statements
  - M does not support changing-state or mutable data
  - Every query is a single expression that returns a single value
  - Every query has a return type

- Get Started with M
  - Language is case-sensitive
  - It's all about writing expressions
  - Query expressions can reference other queries by name

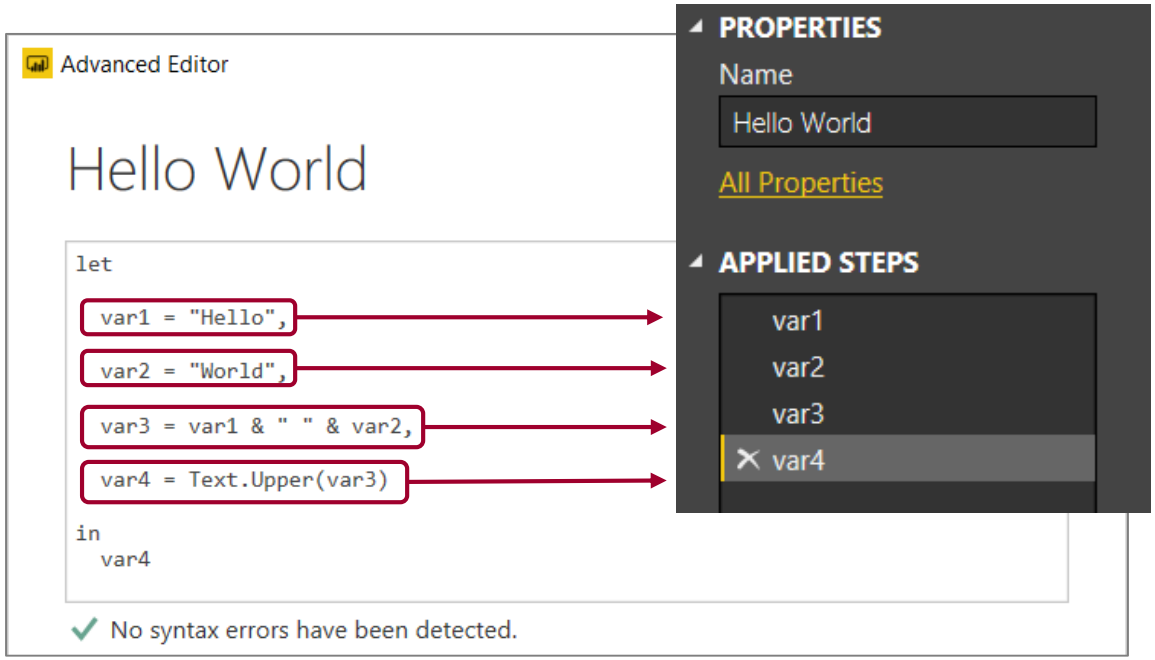
Referencing Other Queries

- Query can reference other queries by name
  - Every query is defined with a return type



Let Statement

- Queries usually created using `let` statement
  - Allows a single expressions to contain inner expressions
  - Each line in `let` block represents a separate expression
  - Each line in `let` block has variable which is named step
  - Each line in `let` block requires comma at end except for last line
  - Expression inside `in` block is returned as `let` statement value



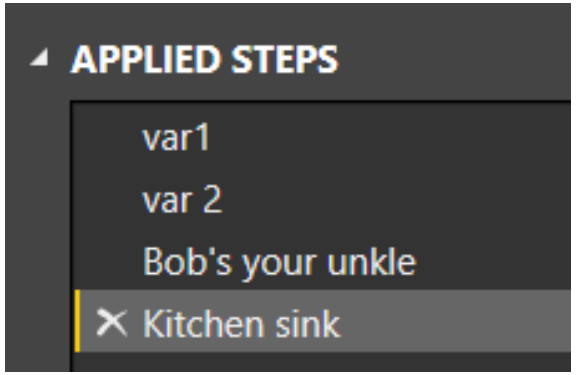
Comments and Variable Names

- M supports using C-style comments
  - Multiline comments created using `/* */`
  - Single line comments created using `//`

```
/*
  This is my most excellent query
*/
let

  var1 = 42, // the secret of life
```

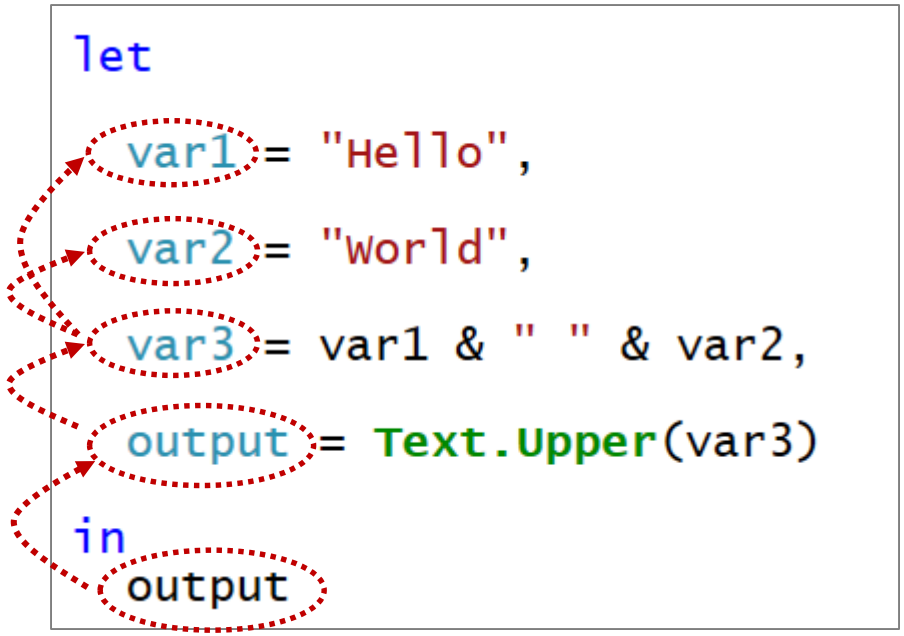
- Variable names with spaces must be enclosed in `#" "`
  - Variable names with spaces created automatically by query designer

```
let

  var1 = "Spaces in ",

  #"var 2" = "variable names ",

  #"Bob's your unkle" = "are evil",

  #"Kitchen sink" = var1 & #"var 2" & #"Bob's your unkle"

in
  #"Kitchen sink"
```



Flow of Statement Evaluation

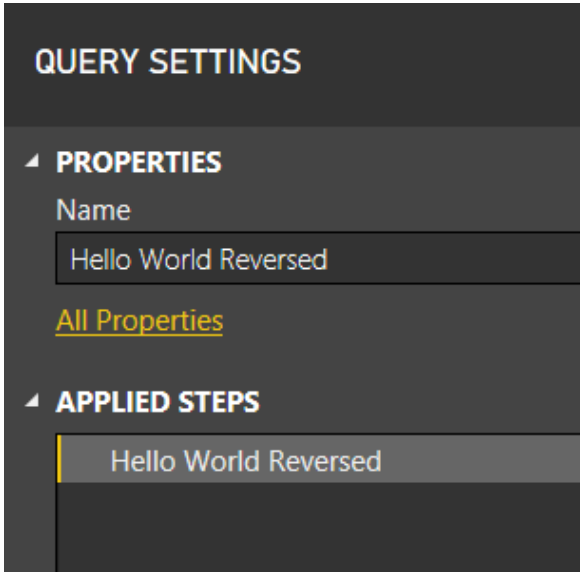
- Evaluation starts with expression inside `in` block
  - Expression evaluation triggers other expression evaluation



Will This M Code Work?

- Yes, the Mashup Engine has no problem with this
  - The order of expressions in `let` block doesn't matter
  - However, the visual designer might get confused

```
let
  var4 = Text.Upper(var3),
  var3 = var1 & " " & var2,
  var2 = "World",
  var1 = "Hello"
in
  var4
```



Query Folding

- Mashup engine pushes work back to datasource when possible
  - Column selection and row filtering
  - Joins, Group By, Aggregate Operations
- Datasource that support folding
  - Relational database
  - Tabular and multidimensional databases
  - OData Web services
- What happens when datasource doesn't support query folding?
  - All work is done locally by the mashup engine
- Things that affect whether query folding occurs
  - The way you structure your M code
  - Privacy level of datasources
  - Native query execution

Query Folding Example

- When you execute this query in Power BI Desktop…

```
let
  Source = Sql.Database("ODYSSEUS", "WingtipSalesDB"),
  CustomersTable = Source{[Item="Customers"]}[Data],

  // select rows
  FilteredRows = Table.SelectRows(CustomersTable, each ([State] = "FL")),

  // select columns
  ColumnsToKeep = {"CustomerId", "FirstName", "LastName"} ,
  RemovedOtherColumns = Table.SelectColumns(FilteredRows, ColumnsToKeep),

  // rename columns
  ColumnRenamingMap = { {"FirstName", "First Name"}, {"LastName", "Last Name"} },
  RenamedColumns = Table.RenameColumns(RemovedOtherColumns, ColumnRenamingMap)

in
  RenamedColumns
```

- Mashup Engine executes the following SQL query

```
execute sp_executesql
N'select [_].[CustomerId] as [CustomerId],
         [_].[FirstName] as [First Name],
         [_].[LastName] as [Last Name]
  from [dbo].[Customers] as [_]
  where [_].[State] = ''FL'' and [_].[State] is not null'
```

Native Queries

- No query folding occurs after native query

```
let

  DatabaseServer = "cpt.database.windows.net",

  DatabaseName = "WingtipSalesDB",

  SQL = "SELECT CustomerId, FirstName, LastName" &
        " FROM Customers" &
        " WHERE CustomerId <= 10" &
        " ORDER BY LastName, FirstName" ,

  Source = Sql.Database( DatabaseServer, DatabaseName , [Query=SQL] ),

  output = Source

in
  output
```

M Type System

- Built-in types

  `any, none`

  `null, logical, number, text, binary`

  `time, data, datetime, datetimezone, duration`

- Complex types

  `list, record, table, function`

- User-defined types
  - You can create custom types for records and tables

M Datatypes

```
let

  // primitives
  var1 = 123,       // number
  var2 = true,      // boolean
  var3 = "hello",   // text
  var4 = null,      // null

  // creating lists
  list1 = {1, 2, 3},           // list of three numbers

  // accessing list elements
  var5 = list1{1},

  // create records
  record1 = [ FirstName="Soupy", LastName="Sales", ID=3 ],

  // accessing records
  var6 = record1[FirstName],

  // table
  table1 = #table( {"A", "B"}, { {1, 2}, {3, 4} } ),

  // creating function
  function1 = (x) => x * 2,

  // calling function
  output = function1(var1)

  in
    output
```

Initializing Dates and Times

```
// time
var1 = #time(09,15,00),

// date
var2 = #date(2013,02,26),

// date and time
var3 = #datetime(2013,02,26, 09,15,00),

// date and time in specific timezone
var4 = #datetimezone(2013,02,26, 09,15,00, 09,00),

// time durection
var5 = #duration(0,1,30,0),
```

Lists

- List is a single dimension array
  - Literal list can be created using { } operators
  - List elements accessed using { } operator and zero-based index

```
let

  RatPack = { "Frank", "Dean", "Sammy" } ,

  FirstRat  = RatPack{0} ,
  SecondRat = RatPack{1} ,
  ThirdRat  = RatPack{2} ,

  output = FirstRat & ", " & SecondRat & " and " & ThirdRat

in
  output
```

  - Use { }? to avoid error when index range is out-of-bounds

```
Rat4 = RatPack{4},   // error - index range out of bounds
Rat5 = RatPack{5}? , // no error - Rat5 equals null
```

Text.Select

- Text.Select can be used to clean up text value
  - You create a list of characters to include

```
// take a text value with unwanted charactors
input = "!!My text has some @bad things !&^",

// get upper and lower case letters
set1 = {"A".."Z"},
set2 = {"a".."z"},

// get digits 0-9 and convert to text
set3 = List.Transform({0..9}, each Number.ToText(_)),

// add any other allowed charactors
set4 = {" ", "-", "_", "."},

// combine all allowed charactors in single list
allowedChars = set1 & set2 & set3 & set4,

// call Text.Select to strip out unwanted charactors
output = Text.Select(input, allowedChars)
```

Records

- Record contains fields for single instance of entity

```
// create records by using [] and defining fields
Person1 = [FirstName="Chris", LastName="Webb"],
Person2 = [FirstName="Reza", LastName="Rad"],
Person3 = [FirstName="Matt", LastName="Masson"],


// access field inside a record using [] operator
FirstName1 = Person1[FirstName],
LastName2 =  Person2[LastName],
```

- You must often create records to call M library functions

```
// create a record to define HTTP request headers
RequestHeaders = [ Accept="application/json",
                   #"OData-MaxVersion"="4.0"],

// create a second record which contains the first record
OptionsRecord = [ Headers=RequestHeaders ],

// pass the second record as parameter to Web.Contents
Response = Web.Contents(url, OptionsRecord),
```

Combination Operator (&)

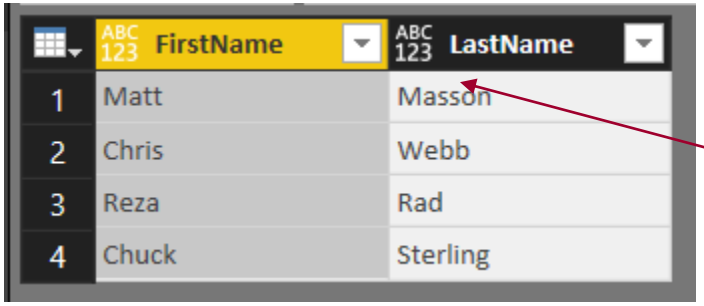
- Used to combine strings, arrays and records

```
// text concatenation: "ABC"
var1  = "A" & "BC",

// list concatenation: {1, 2, 3}
var2 = {1} & {2, 3},

// record merge: [ a = 1, b = 2 ]
var3 = [ a = 1 ] & [ b = 2 ],
```

Table.FromRecords

- Table.FromRecords can be used to create table
  - Table columns are not strongly typed

```
let

  CustomersTable = Table.FromRecords({
    [ FirstName="Matt", LastName="Masson"],
    [ FirstName="Chris", LastName="Webb"],
    [ FirstName="Reza", LastName="Rad"],
    [ FirstName="Chuck", LastName="Sterling"]
  })

in
  CustomersTable
```



	FirstName	LastName
1	Matt	Masson
2	Chris	Webb
3	Reza	Rad
4	Chuck	Sterling

Bad, Bad, Bad ☹

Creating User-defined Types

- M allows you to create user-defined types
  - Here is a user-defined type for a record and a table

```
CustomerRecordType = type [FirstName = text, LastName = text],

CustomerTableType = type table CustomerRecordType,
```

  - User-defined table used to create table with strongly typed columns

```
let

  CustomerRecordType = type [FirstName = text, LastName = text],

  CustomerTableType = type table CustomerRecordType,

  CustomersTable =
    #table(CustomerTableType, {
      { "Matt", "Masson" },
      { "Chris", "Webb" },
      { "Reza", "Rad" },
      { "Chuck", "Sterilicious"}
    }
  )
in
  CustomersTable
```

	FirstName	LastName
1	Matt	Masson
2	Chris	Webb
3	Reza	Rad
4	Chuck	Sterilicious

Using Each with Unary Functions

- Many library functions take function as parameters
  - Function parameters are often unary *(e.g. they accept 1 parameter)*

```
FilteredRows = Table.SelectRows(CustomersTable, (row) => row[CustomerId]<=10  ),
```

- M provides `each` syntax to make code easier to read/write
  - Unary parameter passed implicitly using `_` variable

```
FilteredRows = Table.SelectRows(CustomersTable, each _[CustomerId]<=10  ),
```

  - You can omit `_` variable when accessing fields inside record

```
FilteredRows = Table.SelectRows(CustomersTable, each [CustomerId]<=10  ),
```

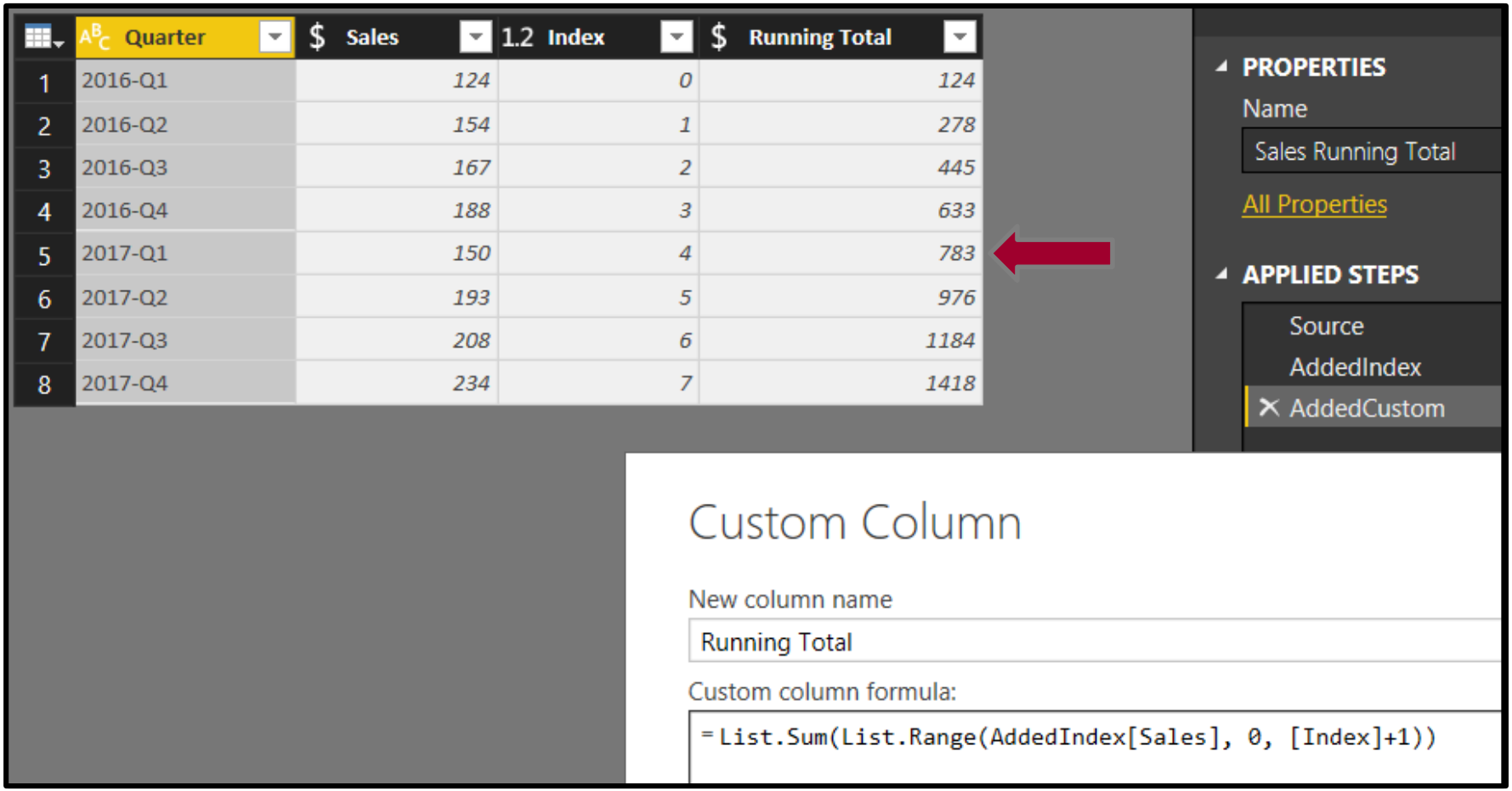
```
AddedColumn =Table.AddColumn(FilteredRows, "Display Name", each [FirstName] & " " & [LastName])
```

  - You must use `_` variable when using **`each`** with a list

```
MyList = { "Item 1", "Item 2", "Item 3" },

MyUpperCaseList = List.Transform(MyList, each Text.Upper(_) )
```

Performing Calculations Across Rows

- Requires adding an index column

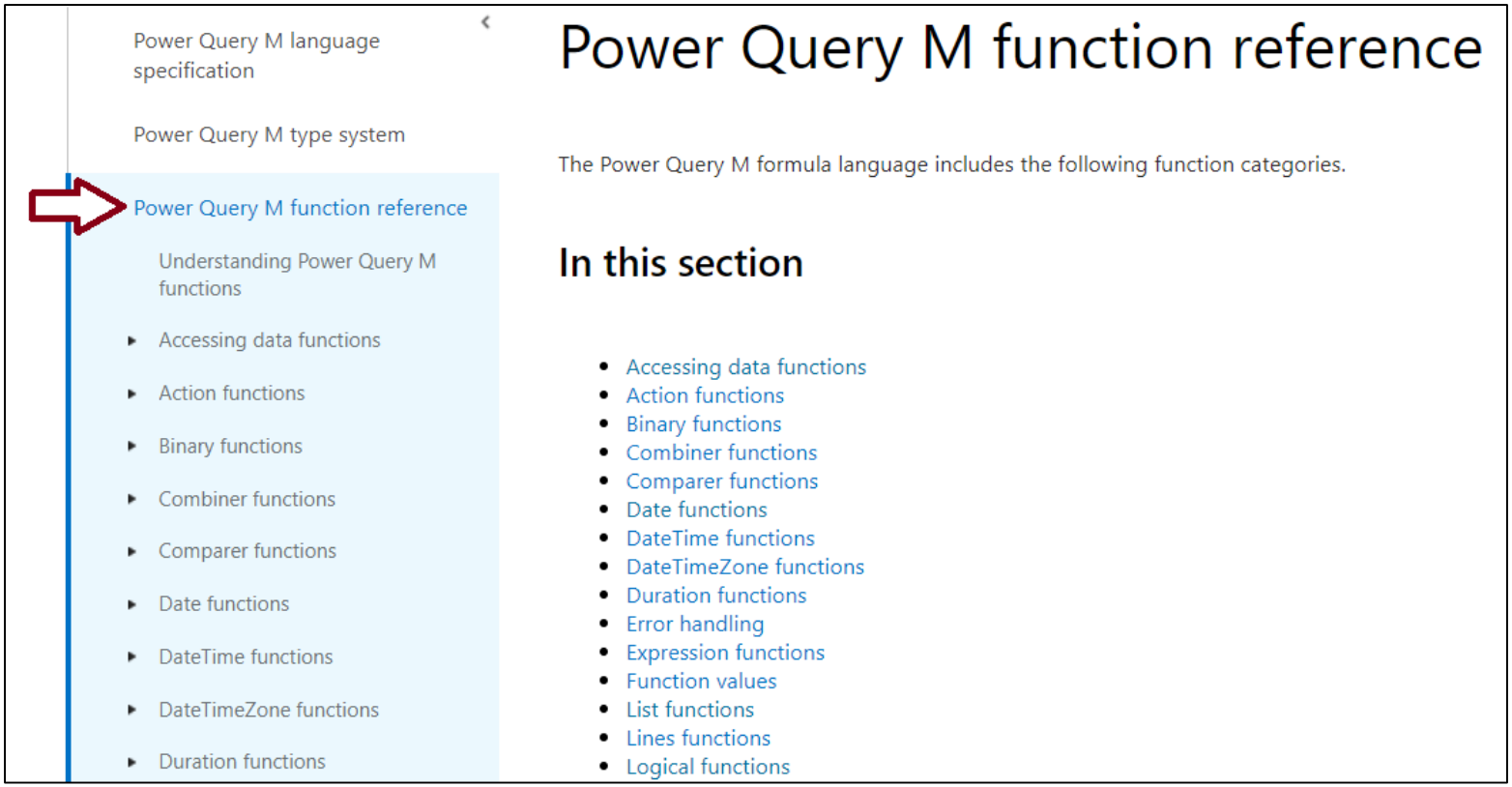


Agenda

- ✓ Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ➢ M Function Library
- Query Functions
- Query Parameters
- Custom Data Connectors

M Function Library

- Check out the Power Query M function reference
  - https://msdn.microsoft.com/en-us/library/mt779182.aspx



Accessing Data using OData.Feed

- OData.Feed can pull data from OData web service
  - OData connector assists with navigation through entities
  - OData connector support query folding

```
let
    Source = OData.Feed("http://subliminalsystems.com/api/"),

    // get Customers table
    CustomersTable = Source{[Name="Customers",Signature="table"]}[Data],

    // select columns
    ColumnsToKeep = {"CustomerId", "FirstName", "LastName", "City", "State", "Zipcode", "Gender", "BirthDate"},
    RemovedOtherColumns = Table.SelectColumns(CustomersTable, ColumnsToKeep),

    // select rows
    FilteredRows = Table.SelectRows(RemovedOtherColumns, each [CustomerId] <= 10),

    // perform other transforms
    ReplacedValue = Table.ReplaceValue(FilteredRows,"F","Female",Replacer.ReplaceText,{"Gender"}),
    ReplacedValue1 = Table.ReplaceValue(ReplacedValue,"M","Male",Replacer.ReplaceText,{"Gender"}),
    ChangedType = Table.TransformColumnTypes(ReplacedValue1,{{"BirthDate", type date}}),
    MergedColumns = Table.CombineColumns(ChangedType,{"FirstName", "LastName"},
                                         Combiner.CombineTextByDelimiter(" ", QuoteStyle.None),
                                         "Customer")

in
    MergedColumns
```

- OData makes extra calls to acquire metadata
  - Let's look at the execution of this query using Fiddler

Web.Contents

- Can be more efficient than OData.Feed
  - You can pass OData query string parameters (e.g. $select)

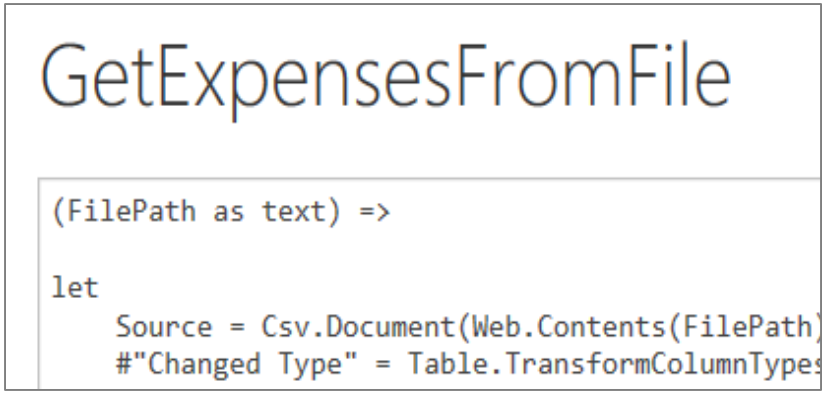
```
let
    // create REST URI for OData source
    Source = "http://subliminalsystems.com/api/Customers?" &
             "?$select=CustomerId,FirstName,LastName,City,State,Zipcode,Gender,BirthDate" &
             "&filter=(CustomerId+le+10)",

    // create options record for calling Web.Contents
    OptionsRecord =  [Headers=[Accept="application/json;odata=nometadata",
                               #"OData-MaxVersion"="4.0"]],

    // call Web.Content to make call across network
    WebContents = Web.Contents(Source, OptionsRecord),

    // deal with JSON dataset return by Web.Contents
    JsonDocument = Json.Document(WebContents),
    RecordList = Record.ToTable(JsonDocument){1}[Value],
    Table = Table.FromList(RecordList, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    ColumnsToExpand = {"CustomerId", "FirstName", "LastName", "City", "State", "Zipcode", "Gender", "BirthDate"},
    ExpandedColumns = Table.ExpandRecordColumn(Table, "Column1", ColumnsToExpand, ColumnsToExpand),
```

Agenda

- ✓ Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ M Function Library
- ➢ Query Functions
- Query Parameters
- Custom Data Connectors

Understanding Function Queries

- Query can be converted into reusable function
  - Requires editing query M code in Advanced Editor
  - Function query defined with one or more parameters



  - Function query can be called from other queries
  - Function query can be called using Invoke Custom Function
  - Function query can't be edited with visual designer

List.Generate

- `List.Generate` accepts 3 function parameters

```
MyList = List.Generate( ()=>1, (item)=>(item<=10), (item)=>(item+1) )
```

	List
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

- You can use `each` syntax for 2nd and 3rd parameter

```
MyList = List.Generate( ()=>1, each _<=10, each _+1 )
```

- You can optionally split functions out into separate expressions

```
let

  StartFunction =  ()=>10,
  TestFunction = each (_ <= 70),
  IncrementFunction = each (_ + 10),

  MyList = List.Generate( StartFunction, TestFunction, IncrementFunction)

in
  MyList
```

Agenda

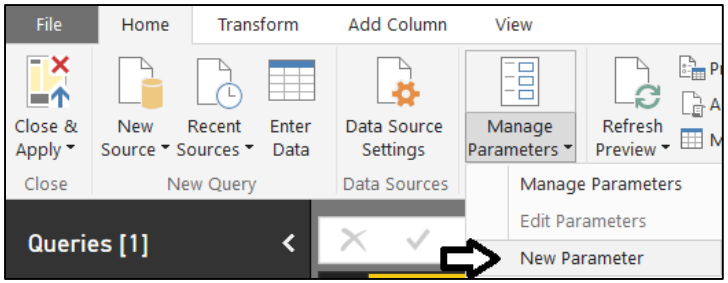
- ✓ Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ M Function Library
- ✓ Query Functions
- ➢ Query Parameters
- Custom Data Connectors

Query Parameters

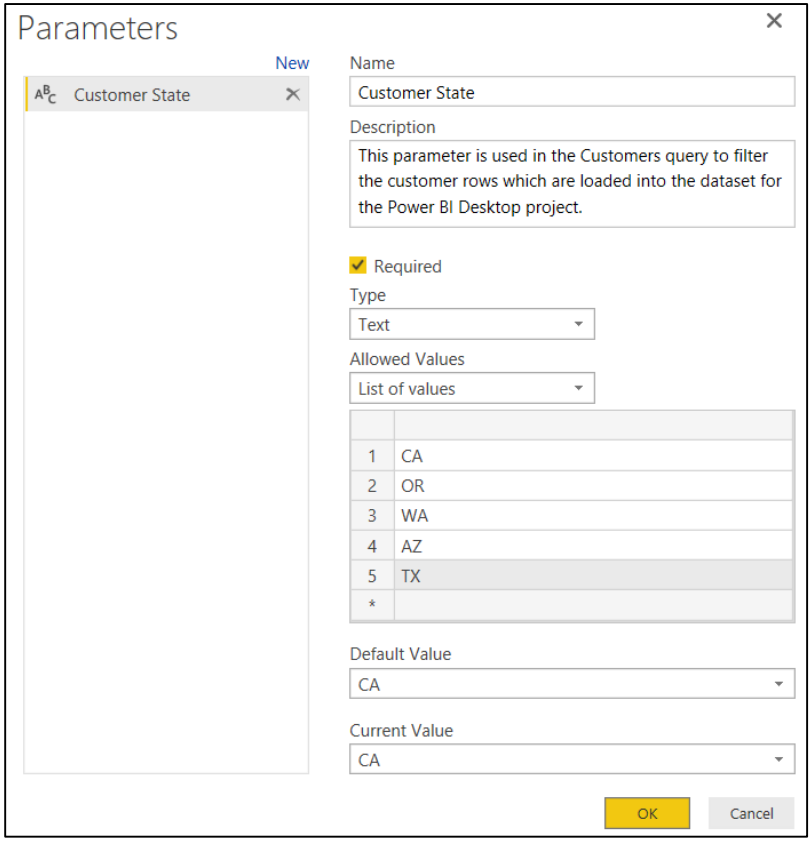
- What is a Query Parameter?
  - Configurable setting with project scope
  - Strongly-typed value to which you can apply restrictions
  - Can be referenced from a query
  - Selected values can be populated using list
- Where are Parameters commonly used
  - To parameterize data source connection details
  - To filter rows when importing data

Creating Query Parameters

- Parameters can be created using **Manager Parameters** menu

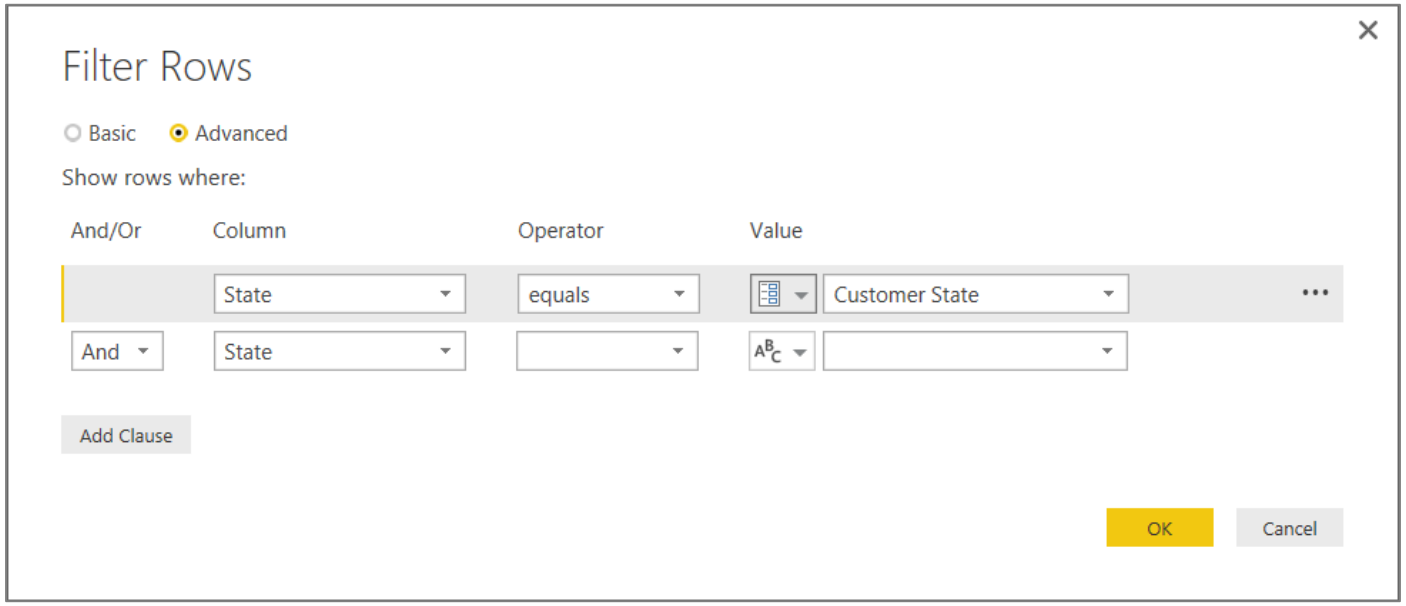


- Parameter properties
  - Name
  - Description
  - Required
  - Allowed Values
  - Default Value
  - Current Value

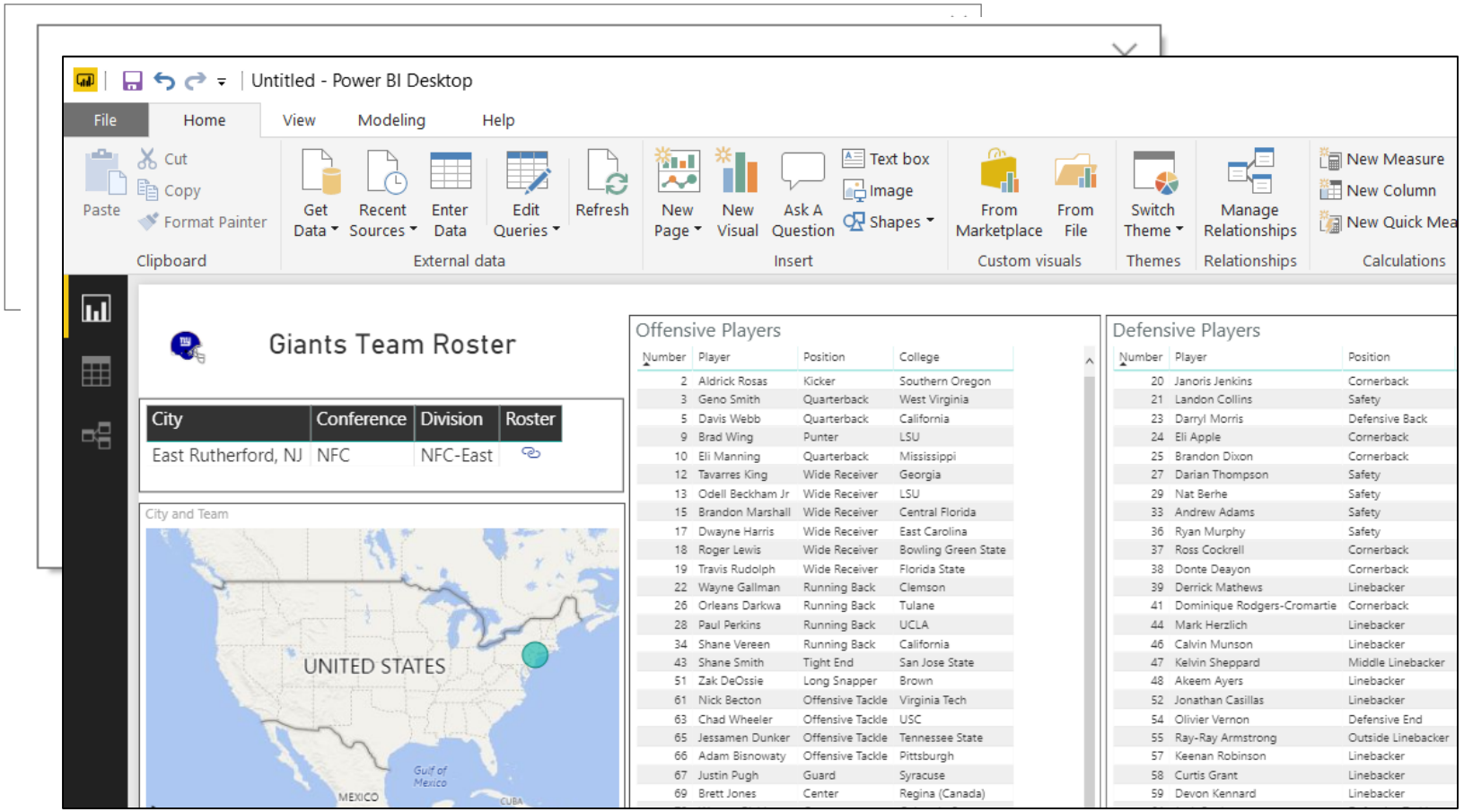


Referencing Parameters in a Query

- Parameters can be referenced inside query
  - Next query execution uses current parameter value

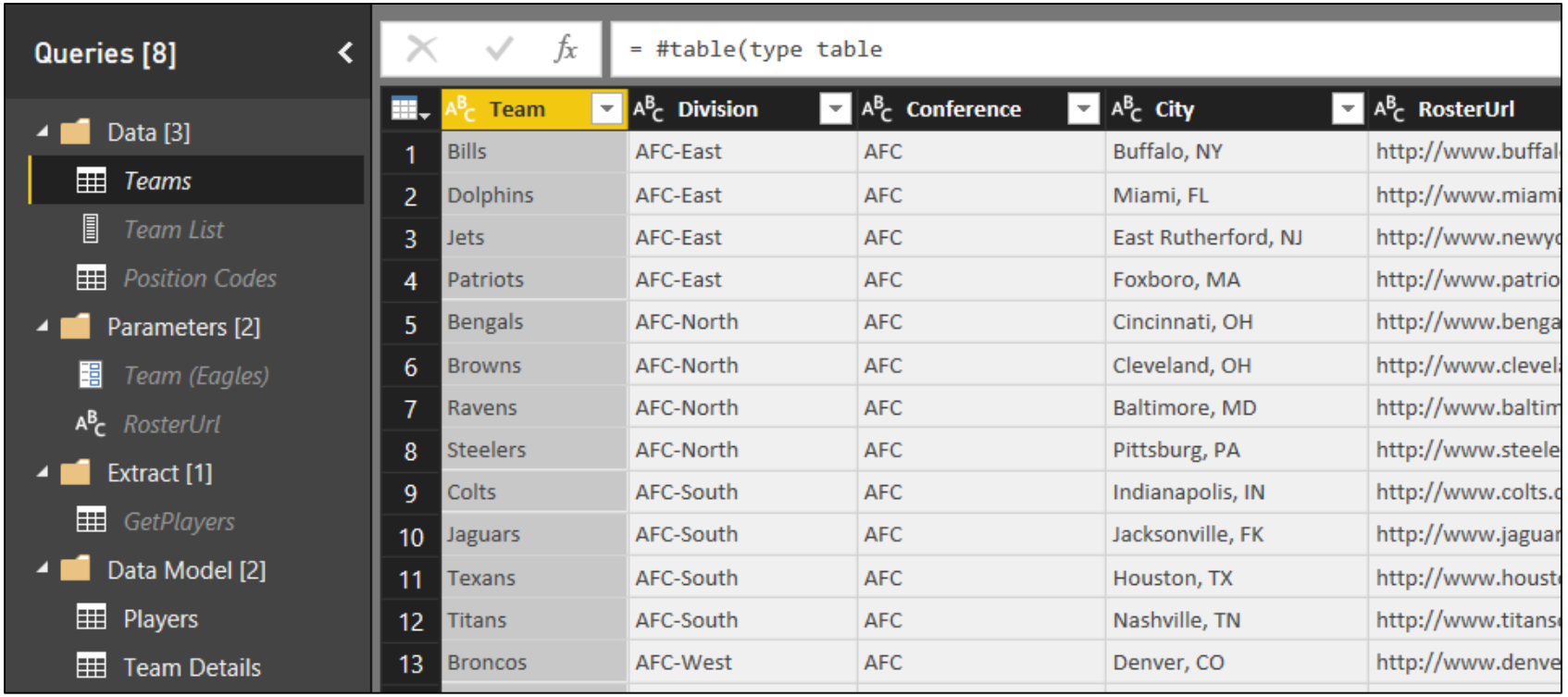


Creating a Project Template File



The Template File Implementation

- Solution required advanced query design



	Team	Division	Conference	City	RosterUrl
1	Bills	AFC-East	AFC	Buffalo, NY	http://www.buffal
2	Dolphins	AFC-East	AFC	Miami, FL	http://www.miami
3	Jets	AFC-East	AFC	East Rutherford, NJ	http://www.newyo
4	Patriots	AFC-East	AFC	Foxboro, MA	http://www.patrio
5	Bengals	AFC-North	AFC	Cincinnati, OH	http://www.benga
6	Browns	AFC-North	AFC	Cleveland, OH	http://www.clevel
7	Ravens	AFC-North	AFC	Baltimore, MD	http://www.baltim
8	Steelers	AFC-North	AFC	Pittsburg, PA	http://www.steele
9	Colts	AFC-South	AFC	Indianapolis, IN	http://www.colts.c
10	Jaguars	AFC-South	AFC	Jacksonville, FK	http://www.jaguar
11	Texans	AFC-South	AFC	Houston, TX	http://www.houst
12	Titans	AFC-South	AFC	Nashville, TN	http://www.titans
13	Broncos	AFC-West	AFC	Denver, CO	http://www.denve

Agenda

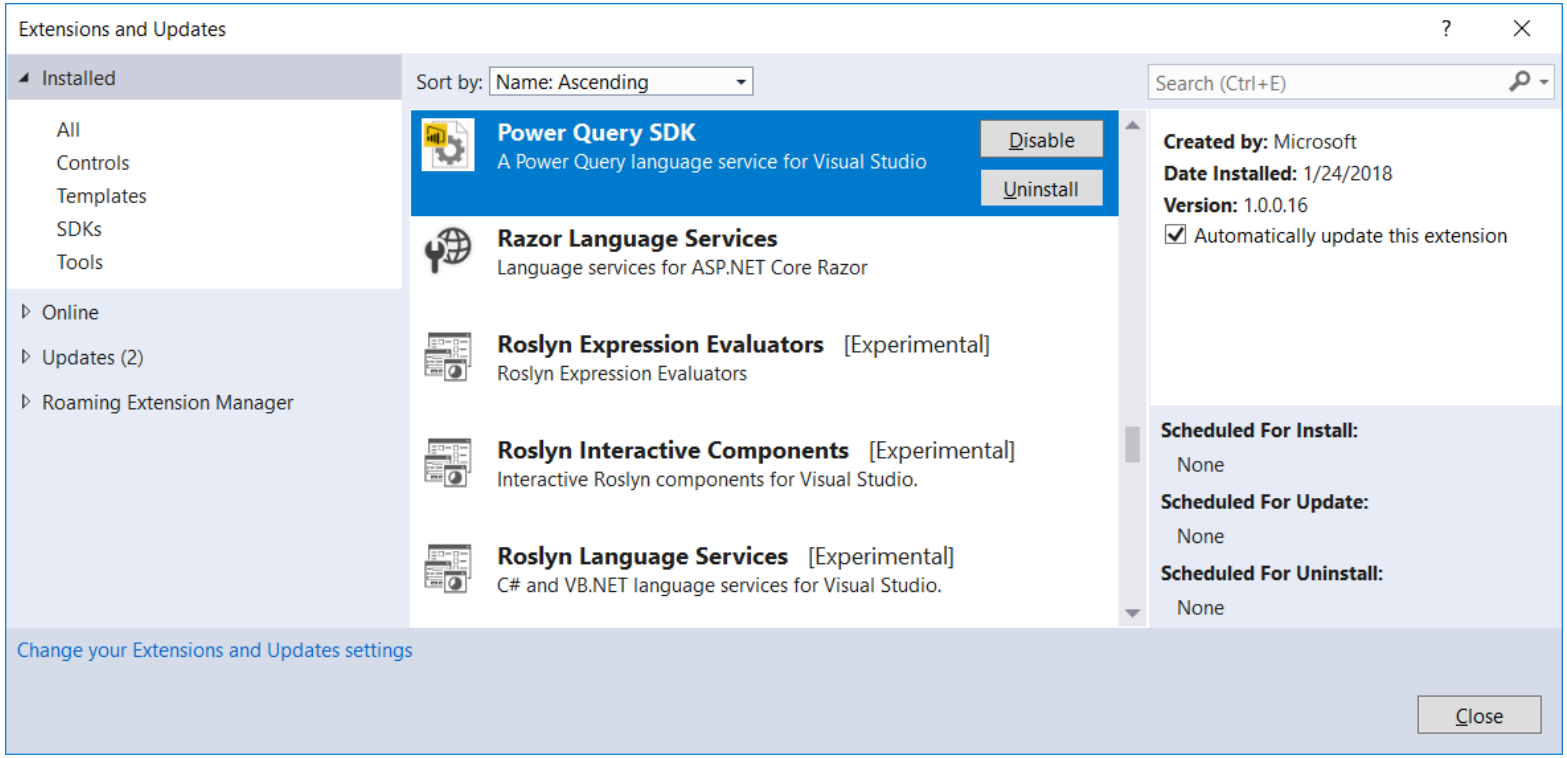
- ✓ Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ M Function Library
- ✓ Query Functions
- ✓ Query Parameters
- ➢ Custom Data Connectors

Developing Custom Data Connectors

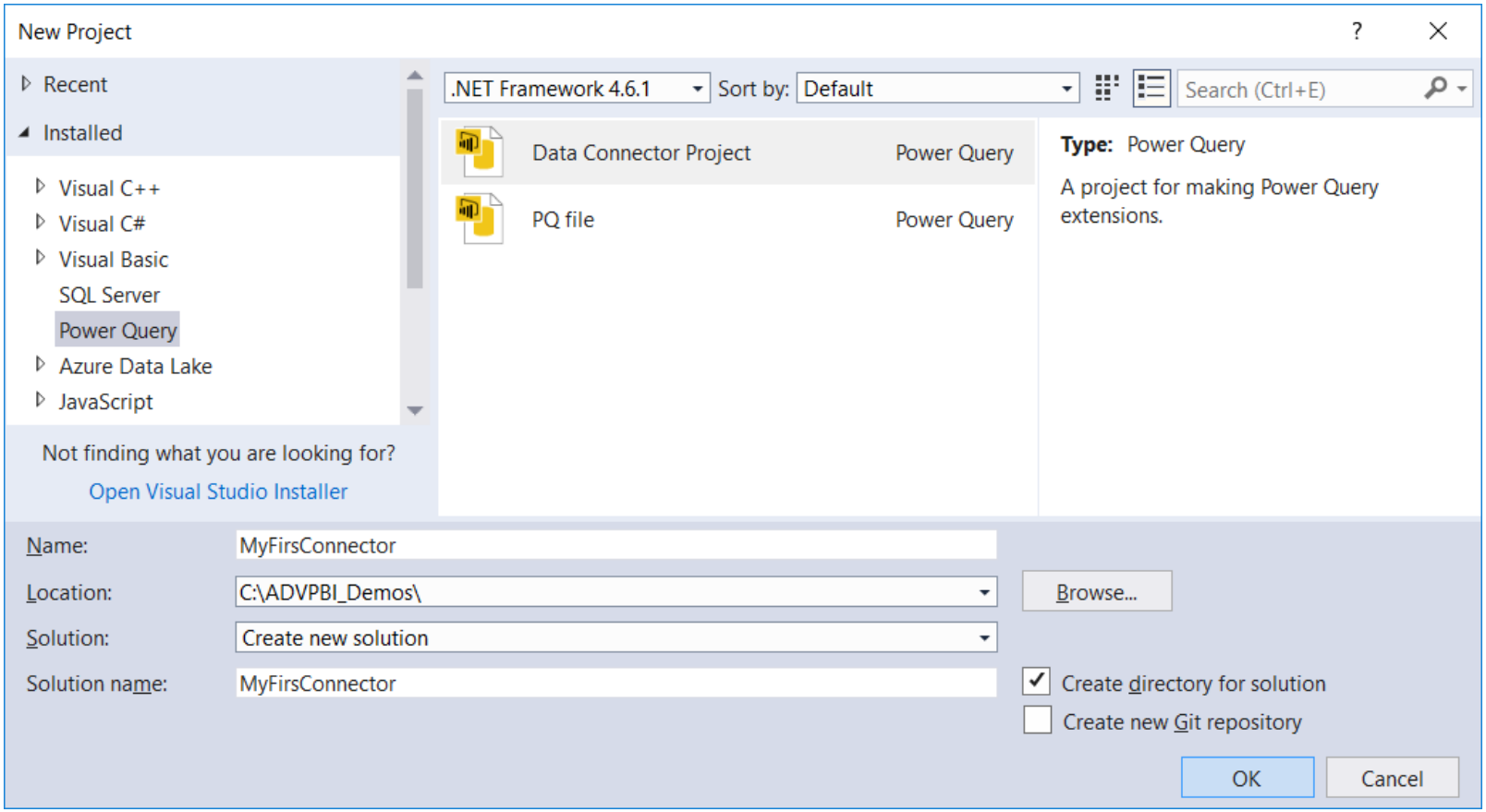
- Custom Connectors let you write reusable query logic
  - Custom connector is written using M programming language
  - Custom connector can be used across PBIX project files

- Common motivation for developing a custom connector
  - Creating a friendly view of a REST API for business analyst
  - Providing branding on top of existing connector
  - Exposing a limited/filtered view over your data source
  - Control how mashup engine authenticates against datasource
  - Implementing OAuth v2 authentication flow for a SaaS offering
  - Enabling Direct Query for a data source via an ODBC driver`

Power Query SDK

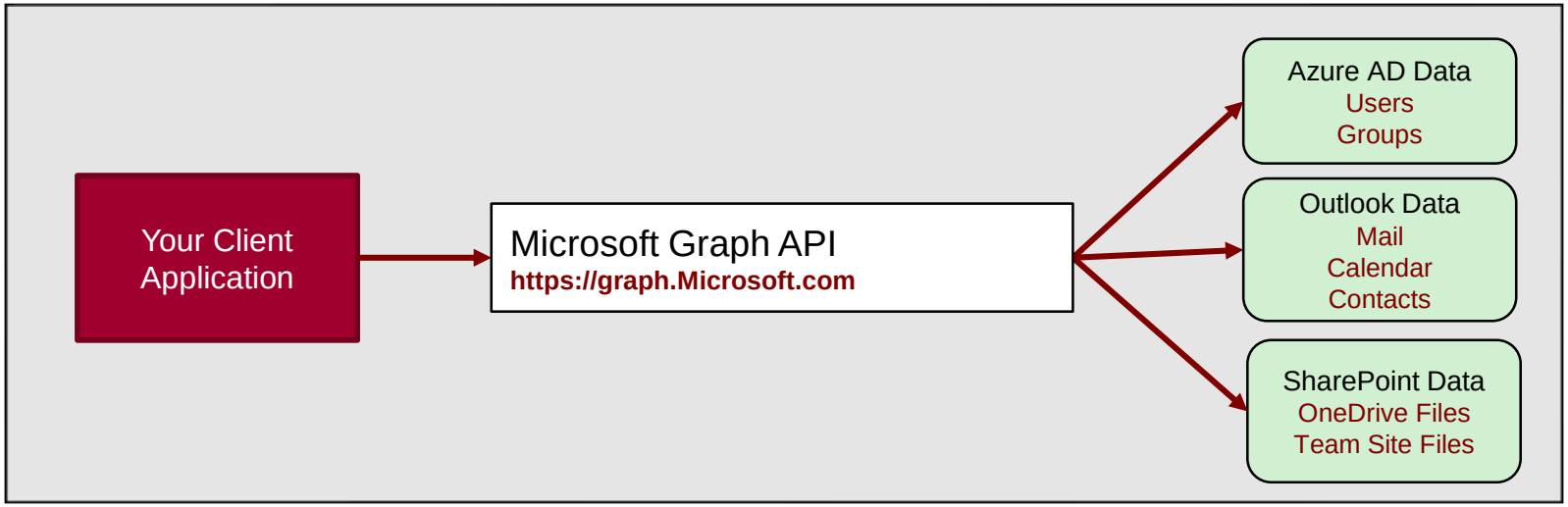


Creating a New Data Connector Project



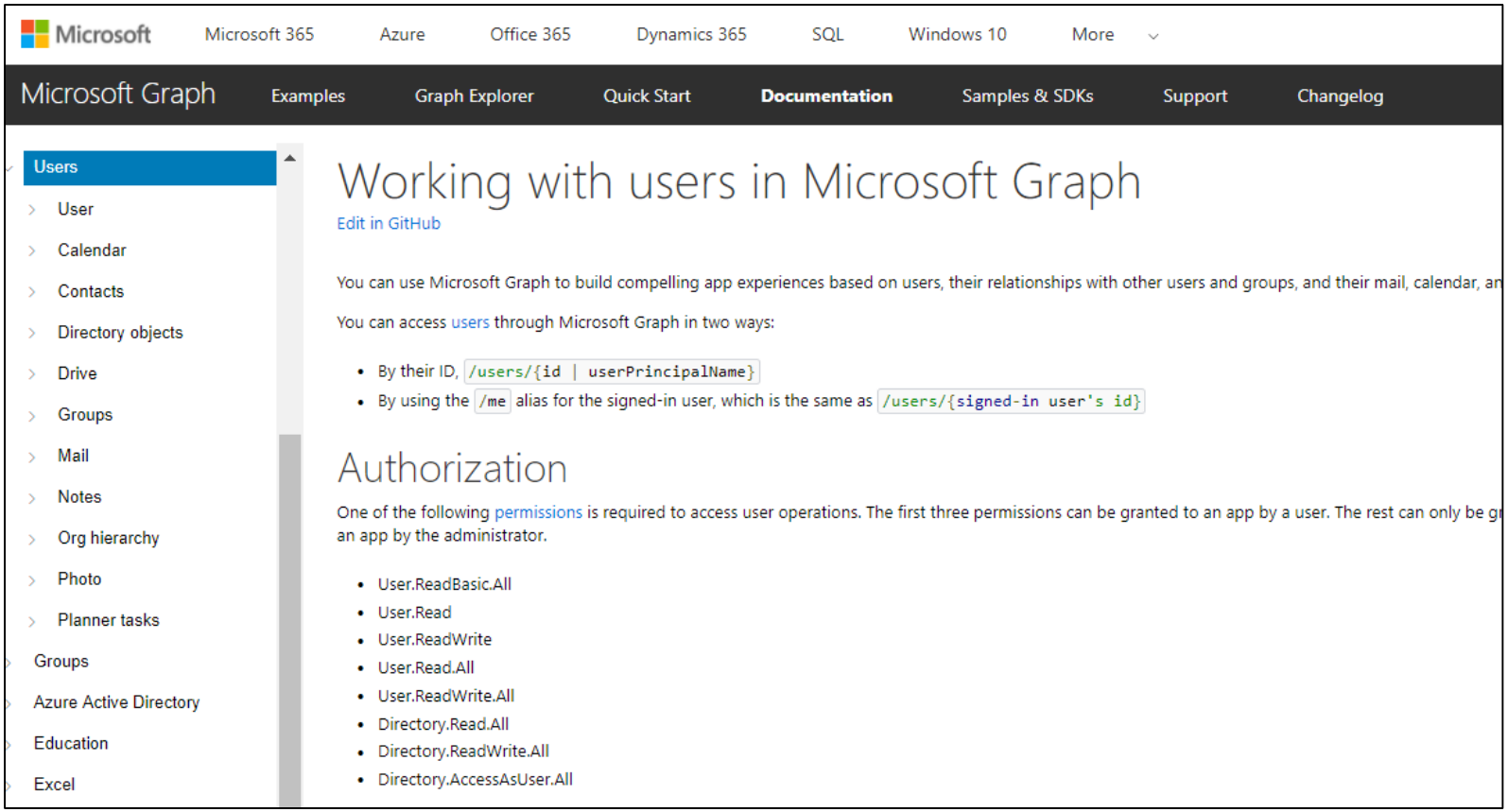
The Microsoft Graph API

- Designed as a one-stop-shopping kind of service
  - Abstracts away divisions between AD, Exchange and SharePoint
  - No need to discover endpoints using the Discovery Service
  - You can acquire and cache a single access token per user



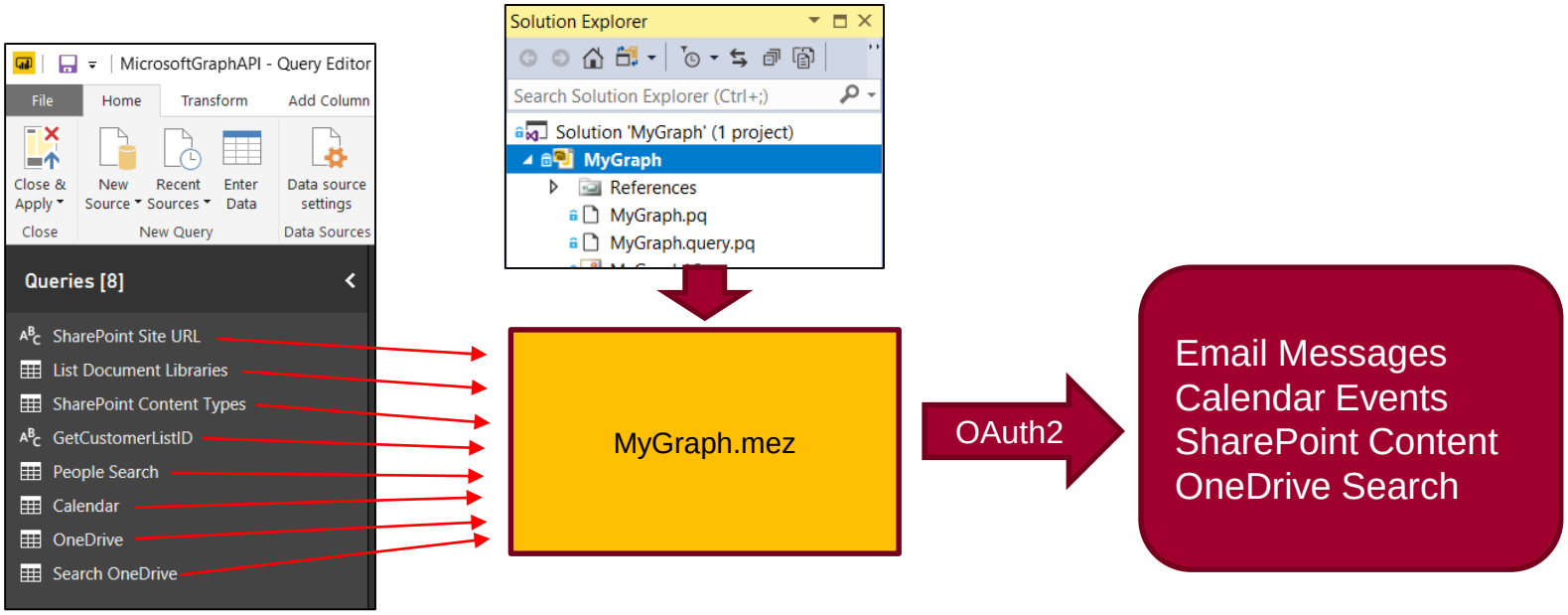
More Info on the Microsoft Graph API

- https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0



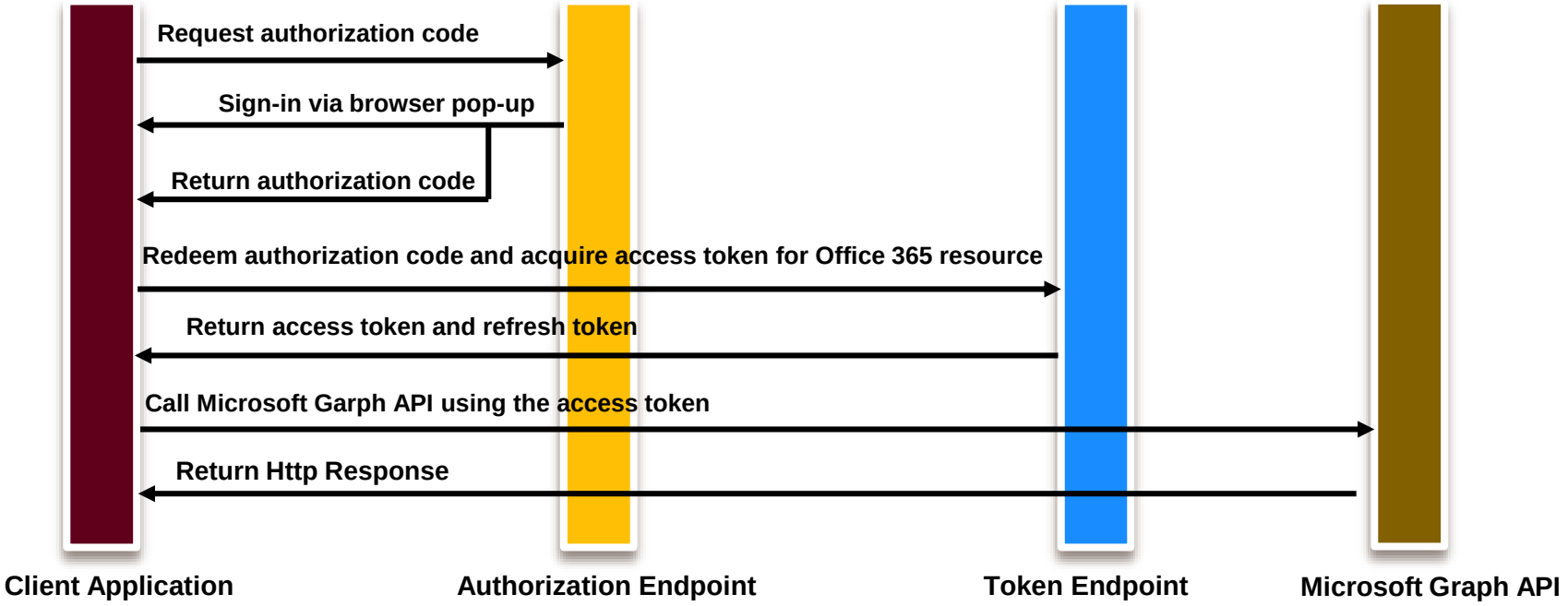
MyGraph Demo

- Project originally created by Matt Masson
  - Connector designed to query Microsoft Graph API
  - Connector provides code to authenticate with OAuth2



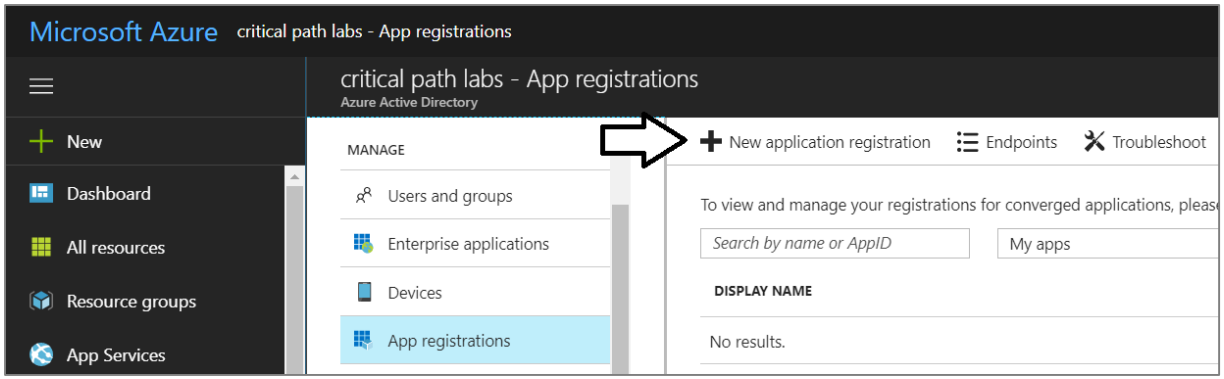
Authorization Code Grant Flow

- Sequence of Requests in Authorization Code Grant Flow
  - Application redirects to AAD authorization endpoint
  - User prompted to log on at Windows logon page
  - User prompted to consent to permissions (first access)
  - AAD redirects to application with authorization code
  - Application redirects to AAD access token endpoint



Registering an Azure Application

- Can be done using Azure portal



- Details you need for the custom data connector
  - Client ID
  - Client Secret
  - Redirect URL

Summary

- ✓ Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ M Function Library
- ✓ Query Functions
- ✓ Query Parameters
- ✓ Custom Data Connectors