Ben Gribaudo
Solutions Engineer / Senior Developer/Database Engineer

Power Query M Primer (Part 15): Error Handling

January 15, 2020 • Data Transformation, Errors, Microsoft Excel, Microsoft Power BI, Power Query M

Your Power Query is skipping merrily along its mashup way. Then, bam! Something bad happens! Uh oh! What do you do when an error raises its ugly head? Or, for that matter, what if code you write detects an anomaly and you want to announce this fact in an informative manner?

Thankfully, M has error handling capabilities, allowing you to both raise and handle runtime errors. We'll learn how to do both.

Important: If you're familiar with the idea of an exception from other programming languages, Power Query's error handling is different in at least one significant respect from what you may be familiar with.

Let's get going!

Series Index

Announcing an Error

In Power Query, each expression *must* produce something. Ideally, this is the expected value. However, there's an alternative: an expression can raise an error, which is a special way of indicating that the expression could not produce a value.

The main way to raise an error is by using keyword `error` accompanied with a record describing the problem.

```
error [
  Reason = "Business Rule Violated",
  Message = "Item codes must start with a letter",
  Detail = "Non-conforming Item Code: 456"
]
```

In the *error definition record*, five fields are relevant: `Reason`, `Message`, `Message.Format`, `Message.Parameters` and `Detail`. Technically, all these fields are optional, and any extra fields included in the error definition record will be ignored.

Special behavior applies to field `Reason` and the `Message`* trio of fields:

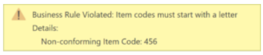
- `Reason`—If this field is missing, the error that's raised will have its reason defaulted to "Expression.Error" (at least, this is true with the version of the mashup engine I'm using—technically, the language specification doesn't mandate this defaulting).
- `Message*` Fields—Two options are available for defining the error's message: Directly specify a `Message`, or use `Message.Format` + `Message.Parameters` to define a structured error message (see *New M Feature: Structured Error Messages* for more details).

As an alternate to creating the error definition record by hand, helper method `Error.Record` can be used to build the record. The function's first argument maps to field `Reason`. The second to either field `Message` or, if a list is passed as `Error.Record`'s forth argument, to `Message.Format`. Arguments three and four map to `Detail` and `Message.Parameters`, respectively. Unlike the above build-your-own-record approach, `Error.Record` requires that you provide a `Reason`; its other arguments are optional.

```
error Error.Record("Business Rule Violated", "Item codes must start with a letter", "Non-conforming Item Code: 456")
```

It's up to you as to whether you prefer to create error definition records using `[...]` syntax or with `Error.Record`. In either case, ultimately, a record is being created which you hand over to `error` when you're ready for the error to be raised.

Both of the above examples produce an equivalent error:

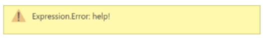


```
Business Rule Violated: Item codes must start with a letter
Details:
    Non-conforming Item Code: 456
```

Looking at the above screenshot, it's easy to see how the three values that were provided map to the error messaging that's displayed.

In lieu of a record, error also accepts a string. The resulting error will have its `Message` set to the provided string and its `Reason` set to "Expression.Error" (at least, that's the default `Reason` with the mashup engine version I'm using—technically, the language specification doesn't mandate this defaulting).

```
error "help!"
```



```
Expression.Error: help!
```

Ellipsis Shortcut

There's also a shortcut operator for raising errors which comes in handy during development.

Let's say you want to test a mashup that's under development where you haven't yet implemented every branch of each expression. Of course, since each branch must either return a value or raise an error, you can't test run your query without putting something as a placeholder in those unimplemented branches, but what should you use?

When you encounter a situation like this, consider the ellipsis operator (`...`). When invoked, `...` raises an error something like "Expression.Error: Not Implemented" or "Expression.Error: Value was not specified" (the exact wording depends on your mashup engine version).

Here's a bit of code where the developer hasn't yet implemented the `if` statement's `else` branch so is using `...` as a placeholder:

```
if Value then DoSomething() else ... // when Value evaluates to false, "..." is called, which raises the placeholder error
```

(Notice how keyword `error` is not used. The ellipsis operator **both** defines *and* raises the error. Very short, sweet and simple to use.)

Special Behavior

What exactly happens when an error is raised? What special behavior does raising an error entail that sets it apart from simply returning an ordinary value?

Let's start with an expression:

```
SomeFunction(GetValue())
```

When evaluated under normal circumstances, first `GetValue()` is executed. Then, the value it produces is passed into `SomeFunction()`. Lastly, `SomeFunction()`'s result is returned as the expression's output.

Heaven forbid, but suppose instead that `GetValue()` raises an error. Immediately, further execution of the expression stops. `SomeFunction()` is not called. Instead, `GetValue()`'s error becomes the expression's output: it is propagated (a.k.a. raised) to whomever or whatever invoked the expression.

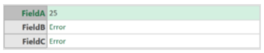
What happens next depends on whether that whomever or whatever can hold a value: the error may be contained or may become the mashup's top-level error. Only in case of the latter does the error cause the mashup as a whole to terminate.

Error Containment

If the error is encountered by an expression that defines *something* holding a value (like the expression for a record field, a table cell or a let variable), **the error is contained** by that *something*—its effects are limited to that *something* and any logic that attempts to access that *something's* value.

Below, the effects of `GetValue()`'s error are contained to the portion of the larger mashup affected by it. **The error does not terminate the entire mashup**; rather, the mashup completes successfully and returns a valid record. **Only *`FieldB`* and *`FieldC`* are errored** because they are the only "somethings" affected by the error.

```
let
  GetValue = () => error "Something bad happened!",
  DoSomething = (input) => input + 1,
  Result = [
    FieldA = 25,
    FieldB = DoSomething(GetValue),
    FieldC = FieldA + FieldB
  ]
in
  Result
```



FieldA	25
FieldB	Error
FieldC	Error

This containment of errors brings with it another special behavior: When an error is contained, **the error is saved into the *something* that contains it**. Throughout the remainder of the mashup's execution, **any attempt to access that something's value causes the saved error to be re-raised**. When an access attempt occurs, the logic that originally caused the error is *not* re-evaluated to see if it now will produce a valid value; that logic is skipped and the previously saved error is simply re-raised.

Below, *`Data`*'s `GetDataFromWebService()` is only evaluated once, even though *`Data`* itself is accessed twice. The second access attempt receives the error saved from the first access.

```
let
  Data = GetDataFromWebService() // raises an error
in
  { List.Sum(Data[Amount]), List.Max(Data[TransactionDate]) }
```

Top-Level Errors

When an error is encountered, if nothing contains it, the error is propagated from the mashup's top-level expression (the mashup's output clause) to the host environment as the mashup's result. Execution of the mashup then stops.

This mashup's top-level expression errors. Nothing is present to contain the error, so the mashup dies, outputting the error as its result:

```
let
  GetValue= () => error "Something bad happened!",
  SomeFunction = (input) => input + 1
in
  SomeFunction(GetValue())
```

The below mashup's error is first contained in *`Result`* but then the top-level expression accesses *`Result`* which results in the error being re-raised to the top-level expression. Since nothing contains the error this time, it becomes the mashup's output—like the preceding, the mashup dies with the error.

```
let
  GetValue= () => error "Something bad happened!",
  SomeFunction = (input) => input + 1,
  Result = SomeFunction(GetValue())
in
  Result
```

Containment vs. Exceptions

Power Query's error containment behavior sets it apart from typical exception-based programming languages. In the world of exceptions, an error automatically propagates all the way to the host environment and so causes the program to die with an error—unless special handling is put in place. With M, an error is automatically contained, so long as something is present to contain it, allowing the mashup as a whole to complete successfully even if select data items could not be computed.

Error containment is a great behavior considering M's target use case: processing data. Suppose the expression defining a table column value errors for one cell out of the entire table. In an exception-based world, this error might cause all processing to terminate. In M's world, the error simply affects that single cell and any code that accesses that cell. Processing continues and the decision of whether the error is significant is left to whatever code consumes the cell's value.

In fact, due to M's laziness, if nothing ever attempts to use that cell's value, its expression may not be evaluated, and so the error never raised. Why should the mashup engine waste effort computing something that will just be thrown away untouched?

```
let
  Data = #table({"Col1"}, {{"SomeValue"}, { error "bad" }})
in
  Table.RowCount(Data)
```

Above, row and column values are not needed to produce the requested output (the count of rows), so the second row's error expression has no effect.

While error containment is a great default behavior, what if it doesn't suit your needs? In particular, with tables, what if it's important to differentiate between rows with errors and those without? Perhaps you're not accessing row contents directly, so aren't doing anything that would trigger error propagation, but still want to know which rows have an error somewhere in them and which do not. `Table.SelectRowsWithErrors` and `Table.RemoveRowsWithErrors` are likely just what you need.

```
let
  Data = #table({"Col1"}, {{"SomeValue"}, { error "bad" }})
in
  [
    RowsWithErrors = Table.RowCount(Table.SelectRowsWithErrors(Data)),
    RowsWithoutErrors = Table.RowCount(Table.RemoveRowsWithErrors(Data))
  ]
```

Handling Errors

With an understanding of raising errors tucked away, what do you do if you're handed an error? Surely there's a graceful way to handle it—some way to try to resolve it!

That's it—that's the keyword: `try`. `try` allows you to attempt to *handle* an error by taking remedial action.

`try` comes in three main variants:

```
// try otherwise
try ExpressionToTry otherwise FallbackExpression

// try catch
try ExpressionToTry catch (e) => FunctionBody
try ExpressionToTry catch () => FunctionBody

// plain try
try ExpressionToTry
```

try otherwise

The first version, *try otherwise*, tries to execute the *expression to try*. If that expression returns a value, `try` simply returns that value. If, instead, the expression errors, that error is ignored, the *otherwise expression* is evaluated and whatever that expression produces becomes the output of the *try otherwise* expression. In essence, if the first expression (the "to try" expression) errors, fallback to the second expression (the "otherwise" expression).

```
try Number.FromText(input) otherwise 0
```

If `Number.FromText` returns a value, then that value is returned from `try`. Instead, if `Number.FromText` raises an error, `try` handles that error, replacing it with the output produced by the otherwise expression (in this case, the value 0). So, if *`input`* can be parsed to a number, that number is returned; otherwise, a default value of 0 is returned.

Keep in mind that only the expression directly to the right of `try` will have its errors caught and replaced. If the otherwise expression returns an error, that error won't be handled by the `try` coming before it. Of course, since the otherwise expression is itself just an expression, you could put a `try` inside *that* expression to handle errors raised from it.

```
try GetFromPrimary()
  otherwise try GetFromSecondary()
    otherwise "Having problems with both servers. Take the rest of the day off."
```

Try otherwise works well in a situations like text-to-number parsing but it can leave something to be desired in more complex scenarios. Why? The catch is that the otherwise is indiscriminate: it replaces *any* error by evaluating the fallback expression. Sometimes, the desired remedial action differs based on the specifics of the error encountered.

try catch

try catch allows us to handle this possibility. If the tried expression completes successfully (i.e. it returns a value), the value it produces is output. If, instead, the expression being tried raises an error, the catch function is invoked. This sounds very much like *try otherwise*, and it is—except for **one very significant difference**.

The catch function can be defined as accepting zero arguments or one argument. If a zero-argument function is used, then *try catch* is identical in behavior to *try otherwise*.

```
// both are equivalent in behavior
try Number.FromText(input) catch () => 0
try Number.FromText(input) otherwise 0
```

On the other hand, if the catch function is defined as accepting an argument, then when that function is invoked, it will be passed a record with details about the error that just occurred. This presents **the possibility to dynamically adapt how the error is handled based on its specifics**—a significant ability not possible with *try otherwise*.

```
let
  Source =
    try GetDataFromPrimary()
    catch (e) =>
      // if the error is because primary is unreachable, fall back to secondary
      if e[Reason] = "External Source Error" and e[Message] = "Server is unreachable"
      then GetDataFromSecondary()

     // if primary had a different problem, reraise the error instead of falling back to secondary
      else error e
in
  Source
```

An *error details records* (passed into the one-parameter catch function above, and also included in plain *try*'s output, as we'll learn shortly) will contain the five fields that make up an error in M: `Reason`, `Message`, `Message.Format`, `Message.Parameters` and `Details`. This is true even if the record originally used to define the error left off one or more of these fields (remember: they're optional when defining the error) or if that record included extra fields.

try catch has some interesting syntax rules for the *catch* function:

- It must be **defined inline**. Defining the function elsewhere and then simply referencing it by name isn't allowed.
- Its parameter list must be defined using **parenthesis syntax**. The `each` shortcut isn't allowed.
- **Type assertions** may not be used in the definition.

These rules make all of the following illegal:

```
// Not allowed -- catch needs to be defined inline; it cannot be a reference to a function defined elsewhere
let
  ErrorHandler = (e) => ...some code...
in
  try SomeFunction() catch ErrorHandler
```

```
// Not allowed -- each cannot be used here
try SomeFunction() catch each ...some code...
```

```
// Not allowed - type assertions may not be used here
try SomeFunction() catch (e as record) as any => ...some code...
```

try

Last but not lease, plain vanilla *try* evaluates the provided expression, then returns a record with details about the expression's result.

```
try SomeExpression
```

If the tried expression completed successfully, the record try outputs is in the form of:

```
[
  HasError = false,
  Value = (whatever value the tried expression returned)
]
```

For example:

```
let
  DoSomething = () => 45,
  Result = try DoSomething()
in
  Result // [HasError = false, Value = 45]
```

If the tried expression raised an error, the returned record looks like:

```
[
  HasError = true,
  Error = (error details record)
]
```

Example:

```
let
  DoSomething = () => error "bad",
  Result = try DoSomething()
in
  Result
// [
//    HasError = true,
//    Error = [
//       Reason = "Expression.Error",
//       Message = "bad",
//       Detail = null,
//       Message.Format = "bad,
//       Message.Parameters = null
//    ]]
//  ]
```

Prior to try catch being added to M, implementing conditional remediation logic required using try with some boilerplate code, resulting in verbose expression like:

```
let
  Primary = try GetDataFromPrimary(),
  Source =
    // if primary is good, use what it returns
    if Primary[HasError] = false
    then Primary[Value]

    // if the error is because primary is unreachable, fall back to secondary
    else if Primary[Error][Reason] = "External Source Error" and Primary[Error][Message] = "Server is unreachable"
    then GetDataFromSecondary()

    // if primary had a different problem, reraise the error instead of falling back to secondary
    else error Primary[Error]
in
  Source
```

try catch achieves the same effect with less code, as this example (repeated from earlier) demonstrates:

```
let
  Source =
    try GetDataFromPrimary()
    catch (e) =>
      // if the error is because primary is unreachable, fall back to secondary
      if e[Reason] = "External Source Error" and e[Message] = "Server is unreachable"
      then GetDataFromSecondary()

     // if primary had a different problem, reraise the error instead of falling back to secondary
      else error e
in
  Source
```

Again, *try catch* didn't used to be part of M. Today, you may still find plain *try* used to implement adaptive error handling in legacy cases, but hopefully new adaptive handling logic will use the more succinct *try catch* syntax instead.

Moving forward, you (likely) will see plain *try* used much less.

Scope

In order have an effect, error handing must occur at a level where the error is encountered. Error handling has no effect on errors that are contained at a different level.

```
let
  Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}})
in
  try Data otherwise 0
```

`Result`'s `try` doesn't do anything for this mashup. Apparently, the developer hoped it would replace any column errors with zero, but that's not how it was applied. The way things were wired up, if the expression defining `Data` raises an error, `try` will replace that error with zero. However, in this case, `Data` returns a valid table. True, there are cells in that table with errors, but those errors are *contained* at the cell level. Since they do not affect `Data`'s *table-level* expression, the `try` at the table expression level has no practical effect.

`try` does help with the following, but its effect may not be what the developer intended.

```
let
  Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}})
in
  try List.Sum(Data[Amount]) otherwise 0
```

Above, `List.Sum` iterates through the values in `Data[Amount]`, adding them up. If an expression defining an item value raises an error, that error is propagated out of `List.Sum`, causing the summation as a whole to abort. `try` handles this error, returning 0 in place of the total `List.Sum` would have output in error-free circumstances.

If that was the intention, great! However, if the intention was to replace any erroring items with 0 while allowing the summation as a whole to complete, `try` must be applied so that it handles errors at the table cell level—it needs to be wired in to receive errors from column value expressions.

At first glance, `Table.TransformColumns(Data, {"Col1", (input) => try input otherwise 0})` might seem like an option. Perhaps surprisingly, this logic does **not** catch errors raised by column value expressions. Why not? A function's arguments are eagerly evaluated *before* their values are passed into the function. If that evaluation results in an error, the function is not invoked so never sees the error; instead, the error is propagated out to the caller. In the case of `Table.TransformColumns`, if a column value expression raises an error, the transformation function (e.g. `(input) => ...`) is not called, so its try cannot handle the error; instead, the error is propagated back to `Table.TransformColumns`.

The problem is that the column value expression needs to be evaluated inside the `try`. To achieve this, try stepping back to the row level. Wire in a function that receives a reference to the entire row. Then, inside your function, use the row reference to access the column's value, wrapped in a `try` expression. Now, any errors raised as a result of that evaluation will be propagated to your `try` expression which can then handle them appropriately.

It's not simple, but one of the simplest ways to get a column's value via a row reference, work with it, then save the resulting output back to the table is to replace the column of interest by using `Table.AddColumn` followed by `Table.RemoveColumns` + `Table.RenameColumns`:

```
let
  Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}}),
  ErrorsReplacedWithZero = Table.AddColumn(Data, "NewAmount", (row) => try row[Amount] otherwise 0),
  RemoveOldAmount = Table.RemoveColumns(ErrorsReplacedWithZero, {"Amount"}),
  RenameNewAmount = Table.RenameColumns(RemoveOldAmount, {"NewAmount", "Amount"})
in
  List.Sum(RenameNewAmount[Amount]) // returns 10
```

I agree with you—the above is a complex solution to achieve something that seems like it should be straightforward. If you want to use an elaborate `try`, unfortunately, some form of working with the table at the row level is required. However, if all you need is to simply replace any error in a particular column with a default value (which is all the above example's `try` does), `Table.ReplaceErrorValues` is your friend.

```
let
  Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}}),
  ErrorsReplacedWithZero = Table.ReplaceErrorValues(Data, {{"Amount", 0}}) // replaces any errors in column Amount with 0
in
  List.Sum(ErrorsReplacedWithZero[Amount]) // returns 10
```

Applying similar behavior to items in a list is more complex. There's no `List.ReplaceErrorValues` library function and `List.Transform(Data, (input) => ...)` doesn't help for the same reason that `Table.TransformColumns` doesn't help with tables. Instead, the simplest solution may be to turn the list into a table, handle the error appropriately, then convert the table back to a list.

```
let
  Data = {10, error "help!", error "save me!"},
  DataAsTable = Table.FromValue(Data),
  ErrorsReplacedWithZero = Table.ReplaceErrorValues(DataAsTable, {{"Value", 0}}),
  BackToList = ErrorsReplacedWithZero[Value]
in
  List.Sum(BackToList) // returns 10
```

Rule Violations

You may not find yourself raising errors that often. Typically, the errors you encounter may come from data connectors and library functions. Don't forget, though, that you can use errors to announce violations of expectations, such as to signify that a particular data item failed to conform to a business rule.

Say you're processing a CSV file where values in the *ItemCode* column should always start with an "A". Early in your mashup, you could check values for conformance to this rule, replacing abnormal values with errors. Later processing steps which access the column will be alerted if they attempt to work with rule-violating values (because of the errors that will be raised).

```
let
  Data = GetData(), // for testing use: #table({"ItemCode"}, {{"1"}, {"A2"}})
  Validated = Table.TransformColumns(Data, {"ItemCode", each if Text.StartsWith(_, "A") then _ else error Error.Record("Invalid Data", "ItemCode does not start with expected letter", _) })
in
  Validated
```

This approach may be of particular interest when writing a base query that several other queries will pull from, as it allows you to centralize your validation (think: the DRY principle) while ensuring that attempted users of erroneous data are forcibly alerted to the presence of the anomalies.

By no means is this the only means of centralizing validation logic. Another option is simply to define an extra column for the rule, set to `true` or `false`, based on whether the rule is complied with:

```
let
  Data = GetData(), // for testing use: #table({"ItemCode"}, {{"1"}, {"A2"}})
  Validated = Table.AddColumn(Data, "ValidItemCode", each Text.StartsWith(_[ItemCode], "A"), type logical)
in
  Validated
```

With this option, logic that cares whether *ItemCode* is valid is responsible to check *ValidItemCode*. If the developer forgets to perform this check, invalid data may be treated as valid. In contrast, the *replace invalid data with errors* approach ensures that logic attempting to access an invalid value is forced to recon with its nonconformance (because the access attempt raises an error).

Whether either of these options is appropriate will depend on your context.

Next Time

That's it! There you have M's error handling. Hopefully you don't encounter too many errors in your Power Query work, but when you do, you know know how to handle them. You also know how to raise your own errors, for cases where you detect an anomaly or need a development placeholder.

Soon the plan is to talk about the behind the scenes way Power Query organizes things (sections) and the faculty M provides to annotate values with additional information (metadata). Before then, though, it's time to look at how the type system works in Power Query.

Until then, happy coding!

Revision History

- **2020-02-10**: Updated section *Scope*, explaining how to use a `try` expressions to handle errors in table cells.
- **2022-06-09**: Revised to reflect M's new `try catch` syntax, as well as to incorporate rudimentary mentions of M's new structured error message functionality.

Related Posts:

1. **The Elusive, Uncatchable Error?**
2. **Adding an Error Details Column**
3. **New M Feature: Structured Error Messages**
4. **Enhancing an Error's Detail**

12 thoughts on "Power Query M Primer (Part 15): Error Handling"

Gavin
September 1, 2020 at 7:57 am

Another really helpful article, helping me to build my understanding of M. However I can't quite get my head around how I can best use this technique in my situation. I simply want to know if one ofd my queries does not complete due to an error.
Currently I use 32Bit Excel 2013 with pq and pp as add-ons. So I have a set of queries that manipulate and clean my source data into a results dataset that will be used by other queries. My colleagues and I run the query using Data>>Refresh>>All or some VBA code.
But if something fails to refresh (lets say a field name has changed) there is no alert. The only way I know of telling is by viewing the queries in the PowerQuery Pane and looking for the orange warning triangles. I would like to either return an error to a table or to detect the error in VBA.
The next step might be to explore how I can use the info in your blog to get a more sophisticated message or handle the error.

Ben Gribaudo Post author
September 29, 2020 at 3:45 pm

Hi Gavin,

Interesting question. I'm not sure on the VBA side. On the idea of returning "an error to a table", are you hoping to have a separate table that lists errors from your various queries (so all errors in one place) or instead to have the table whose query encountered the error return say only one row stating that an error occurred?

Ben

Alex
November 26, 2020 at 2:43 pm

Hi!
When I meet errors in the middle of my multi query calculation, speed drops significantly. I want to move from queries like this

```
let .... in result
to this:
let .... in ThrowErrorIfTableContainsAnyErrors(result)
```

Because in my calculation I need to stop on error. Is there any way to transform errors from the table (or at least first of them) to string and then to raise a new exception? I can't find a way to access error message.
I tried doing like this: Table.AddColumn(xx, "Msg", each try _), it doesnt work. It results in Containment error again.

Ben Gribaudo Post author
December 7, 2020 at 2:51 pm

Does something like this help as a starting place? For example, you could modify it to throw the error from the first non-null Errors column value.

```
let
    Source = #table(null, { {1, error "help"}, { 2, "B"}, { error "bad", "C"} }),
    RowErrorsToTable =
        (row as record) as nullable table => let
            Table = Record.ToTable(row),
            AddedErrorsColumn = Table.AddColumn(Table, "Errors", each let Test = try [Value] in if Test[HasError] = true then Test[Error] else null),
            RemovedDataColumn = Table.RemoveColumns(AddedErrorsColumn, {"Value"}),
            FilteredToErrors = Table.SelectRows(RemovedDataColumn, each [Errors] <> null)
        in
            if Table.IsEmpty(FilteredToErrors) then null else FilteredToErrors,
    AddedErrorsColumn = Table.AddColumn(Source, "Errors", RowErrorsToTable)
in
    AddedErrorsColumn
```

Alex Groberman
March 23, 2021 at 3:54 pm

Hi, I've created a couple of functions for dealing with errors in lists (along with an examples), hope these help!

-Alex

```
let
    List.CustomRemoveErrors = (List as list) as list =>
        let
            FirstItem = List{0},
            FirstItemHasError = (try FirstItem)[HasError],
            FirstItemAsList = if FirstItemHasError then {} else {FirstItem}
        in
            if List.IsEmpty(List) then {} else FirstItemAsList & @List.CustomRemoveErrors(List.RemoveFirstN(List, 1)),

    List.CustomReplaceErrors = (List as list, Replace as any) as list =>
        let
            FirstItemAsList = {try List{0} otherwise Replace}
        in
            if List.IsEmpty(List) then {} else FirstItemAsList & @List.CustomReplaceErrors(List.RemoveFirstN(List, 1), Replace),

    ExampleList = {1, error "a", 2},
    ExampleRemove = List.CustomRemoveErrors(ExampleList),
    ExampleReplace = List.CustomReplaceErrors(ExampleList, "error")
in
    ExampleReplace
```

Ben Gribaudo Post author
March 24, 2021 at 7:49 am

Thanks for sharing, Alex.

Peter Morris
May 4, 2021 at 12:52 pm

Hi,

I'm using PQ to collate a number of files from a folder. I'm quite happy with converting errors to something understandable and that seems to work. But on occasion an error occurs which is not being captured and so the whole import fails – which is very annoying. I'm afraid I cant even think where to start – the overall process has many steps and I cant imagine covering each line with a 'try/otherwise' structure.

Any ideas gratefully received.

Ben Gribaudo Post author
May 28, 2021 at 7:09 am

Are the errors that are not getting captured at the table cell level?

GWP
April 29, 2022 at 3:23 pm

Hello sir,

First, thank you very much! Your primers are invaluable.

Second, I'm running into difficulties applying your error containment technique to errors emanating from `Folder.Files()`. This function throws an error when it encounters a path of length greater than 260 characters. It seems to occur at such a root level that I can't discern how to get 'underneath' or 'before' the error.

Admittedly, super-long paths maybe shouldn't exist ideally, but they do in reality, and I'm struggling to handle them.

Any advice would be greatly appreciated! Thank you 😀

Ben Gribaudo Post author
May 4, 2022 at 11:39 am

Does the hard to catch error scenario happen only if the **first** file has a really long path? If you change the first file to have a shorter path, does the error situation change?

Ben Gribaudo Post author
May 20, 2022 at 9:34 am

A new post on *The Elusive, Uncatchable Error?* just when live here on the blog. Does it help with your situation?

GWP
May 23, 2022 at 7:36 am

The error does not necessarily occur on the first file. It occurs on the first file encountered by `Folder.Files` that exceeds the folder path >248 char or full path >260 char constraints.

The use case is as follows:

I pass `Folder.Files` a string such as `C:\Users\MyName\OneDrive - MyOrg`.

`Folder.Files` chugs along (streaming, presumably, as you suggest in your "Elusive" post), until it hits either a folder path >248 char or a full path >260 char. Then it errors on that particular row.

Since the error doesn't occur on row 1, the "Elusive" fix doesn't trap it.

I can trap the error using `try Table.Buffer()`. This returns `HasError = true` and the Error Record Message "The file name 'C:\Users\blah…*' is longer than the system-defined maximum length."
So I can trap it, but I'm not sure what to do at that point, since all I have is an error record.

What I'd like to do is get a table of all the files at a given location, skipping files/folders that throw path-too-long errors.

I've worked around this issue using `Folder.Contents` recursively. This is a little more forgiving because you start on safe territory (path < 248 char), then recursively explore subfolders, taking care to only fetch the contents of a subfolder if the total path length will be < 248 char. This allows us to programmatically avoid the error situation altogether.

But it still leaves me curious if there is a way to trap and handle errors that occur when `Folder.Files` (or `Folder.Contents`, for that matter) encounters a path-too-long situation.

Thank you for your "Elusive" post 😀