Using Redgate SQLDoc with Data Catalog

One of the challenges that face any company offering products that collate and present information is how best their product should display that information to the intended audience. This challenge is particularly acute in a Data Catalog for two main reasons.

* The intended audience is [diverse with equally diverse needs](https://documentation.red-gate.com/sql-data-catalog/taxonomy/further-advice-on-defining-a-taxonomy).
* The corporate data asset can be rich and varied

Three approaches that can be taken to address the presentation challenge are shown below

* Presentation is handled by the product itself
* Data is exported from the product in a form that can be used in other facilities.
* Official documentation exists to support the use of business intelligence tools on the data catalog repository.

These options should not be considered mutually exclusive or even exhaustive. Reject the tyranny of OR in favor of the genius of AND.

For example, [Redgate Data Catalog](https://documentation.red-gate.com/sql-data-catalog) provides both information dashboards and the facility to export the catalog contents in CSV format.

Although an information tool should always provide some visualizations it is not practical or even possible to provide for every need.

For this reason, explicit support for BI and reporting tools using the database underpinning a Data Catalog is necessary. To make this viable that database has to be well documented. For this [Redgate SQLDoc](https://www.red-gate.com/products/sql-development/sql-doc/index) is an excellent complement for the Redgate Data Catalog product.

Descriptions for the Redgate Data Catalog product

Prior to investigating the Redgate Data Catalog I had written two articles for SQLServerCentral.com on techniques to make it easier to attach descriptions to database objects.

1. [Oiling the gears for the data dictionary](https://www.sqlservercentral.com/articles/oiling-the-gears-for-the-data-dictionary) published in April 2014
2. [Re-oiling the gears for the data dictionary](https://www.sqlservercentral.com/articles/re-oiling-the-gears-for-the-data-dictionary-or-catalog) published in January 2020

Using these techniques I captured my interpretation of what the database tables, views and columns in the Redgate Data Catalog product might be. These are held in <https://github.com/AspieArchitect/DataCatalogDoc>

SQLDoc as a metadata authoring tool

In an ideal world the supply of meaningful descriptions for database objects should be part of standard development practice when objects are created or updated. If you cannot explain the object you are creating/updating to the relevant audience then you should ask yourself two questions.

* Is your understanding of why you are creating/updating objects sufficient?
* Are you developing software with a full understanding of requirements?

A software engineer’s enthusiasm for having high quality documentation available seems to be diametrically opposed to their level of enthusiasm for being the person who provides it. SQL Server also has an awkward syntax for attaching descriptions to objects.

Compare SQL Server’s [sp\_addextendedproperty](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-addextendedproperty-transact-sql?view=sql-server-ver15) and [sp\_updateextendedproperty](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-updateextendedproperty-transact-sql?view=sql-server-ver15) to that provided by platforms such as PostGres, AWS Redshift or Vertica that have a simple syntax

COMMENT ON {object} IS ‘{meaningful description}’;

Reluctance combined with complexity means that documentation is less common that it should be.

This is where the use of Redgate SQLDoc as an authoring tool becomes invaluable. Data stewards, business analysts and data subject matter experts can carry out the necessary software archeology to attach meaningful descriptions to database objects. Such descriptions need to backfill into the source code repository so that they are not lost if objects are dropped and recreated.

It is here that we must remember that documentation should be written for the readers of that documentation. It is not a tick box exercise to close a JIRA ticket or an item to be descoped so that development time gains some contingency padding. A system becomes “legacy” when people avoid changing it for fear of consequences that are unpredictable in both scope and nature. Such unpredictability can driven by a lack of easily accessible information about the system, not just by poor coding hygiene.

Documentation is an artefact that benefits from refactoring and continuous improvement every bit as much as software.

Extending the capabilities of Redgate SQLDoc

Redgate SQLDoc can produce documentation in a number of formats

* MS Word
* CHM – Compiled HTML Help file
* PDF
* HTML
* Git Markdown

We are going to focus on HTML and Git Markdown as web pages and git compatible source control systems are almost ubiquitous.

To make documentation more effective there are techniques we can use to emphasize the relevant information.

* The use of bulleted/ numbered lists
* The use of hyperlinks

Redgate SQLDoc already creates hyperlinks between the database objects it is documenting. It would be useful to be able to hyperlink to content external to the database documentation. For example, if we were to use Redgate SQLDoc to document the Redgate Data Catalog it would be useful to hyperlink to the product documentation.

It is here that we begin to find challenges.

If we attempt to put HTML into object descriptions the product will html encode it. A hyperlink such as the following

<a href=”https://www.red-gate.com/products/dba/sql-data-catalog/”>Data catalog</a>

Will be converted to

&lt;a href=<https://www.red-gate.com/products/dba/sql-data-catalog/>&lt;Data catalog&lt;/a&gt;

If we were to use markdown format instead then we could use the following.

[Data catalog]( <https://www.red-gate.com/products/dba/sql-data-catalog/>)

This would render correctly when Redgate SQLDoc publishes markdown files but not if we asked for html files.

Similarly if we wished to use bullet points then having a description for the dbo.AssetTypes table of the Redgate Data Catalog DB as follows would render in a barely acceptable form in HTML but would break the markdown table structure completely.

Describes the type of object within the data catalog that can be categorized  
\* SQL Server Instance  
\* Database  
\* Schema  
\* Table  
\* Column

A final challenge is the default filenames used when you access the documentation for a directory differs depending on whether you use a web servers or git repository.

* Webservers use index.html
* Git repositories use readme.md

All these problems are solvable with a little code.

Disclaimer for the code below

There is immense latitude in what can be put in an MS\_DESCRIPTION property. The code in this article will work well on column comments where some discipline is observed however, the amount of work ensure that all variations are catered for is prohibitive.

STEP ONE: Generating the initial documentation set

The foundations for the first step were laid in the article [Getting started with SQLDoc and Powershell](https://www.red-gate.com/hub/product-learning/sql-doc/getting-started-with-sql-doc-and-powershell) by Phil Factor.

The techniques in the article were used to create **RunSQLDoc.ps1** file as shown below

Set-Alias sqldoc 'C:\Program Files (x86)\Red Gate\SQL Doc 5\SQLDoc.exe' -Scope Script

# base\_path is the folder containing this script

$base\_path = Split-Path -Parent $PSCommandPath

$documentation\_path = "$base\_path\SQLDoc\DADSPCJUNE2014\_documentation"

# Make sure we start with a clean folder

Get-ChildItem -Path $documentation\_path -Recurse|Remove-Item -Force -Recurse

# Generate both md and html documentation

sqldoc /server:DADSPCJUNE2014 /database:Redgate\_SqlDataCatalog /project:RedgateDataCatalog.sqldoc /fileType:md

sqldoc /server:DADSPCJUNE2014 /database:Redgate\_SqlDataCatalog /project:RedgateDataCatalog.sqldoc /fileType:html

The Get-ChildItem|Remove-Item construct is used instead of the SQLDoc /force option because we are going to rename the index.md files to readme.md. Doing so ensures that the documentation generation has the same starting state every time.

The sqldoc alias is called twice so that both a markdown and html documentation set is generated.

STEP TWO: index.md to readme.md

If the index.md files are renamed to readme.md then any links in the markdown documentation set that point to the original index.md files must also be updated. The approach taken was

* Create a ScriptHelpers subfolder
* Create an index\_to\_readme.ps1 file within the ScriptHelpers folder as follows

[cmdletbinding()]

param([string]$Foldername)

# Git repos automatically render readme.md files so change the index.md files to readme.md files

Get-ChildItem -Path $Foldername -Recurse -File -Filter "index.md"|Rename-Item -Force -NewName "readme.md"

# As all references point to index.md change ALL references across all md files to them to point to readme.md

Get-ChildItem -Path $Foldername -Recurse -File -Filter "\*.md"|ForEach-Object{

(Get-Content -Path $\_.FullName).Replace("index.md","readme.md")|Set-Content -Path $\_.FullName

}

A call to this script can be added to the original RunSQLDoc.ps1 file described in STEP ONE.

& ".\ScriptHelpers\index\_to\_readme.ps1" "$documentation\_path"

When RunSQLDoc.ps1 is run the output will be as follows

* HTML documentation set
* Markdown documentation set
* Default readme.md files
* Documentation breadcrumb trails and links correctly using readme.md as targets

STEP THREE: Rendering links embedded in descriptions

Let us compare an HTML link with the Git markdown equivalent

| **Format** | **Link** |
| --- | --- |
| HTML | <a href=”https://www.red-gate.com/products/dba/sql-data-catalog/” title=”Getting Started with the Redgate SQL Data Catalog”>SQL Data Catalog</a> |
| MD | [SQL Data Catalog]( <https://www.red-gate.com/products/dba/sql-data-catalog/> “Getting Started with the Redgate SQL Data Catalog”>SQL Data Catalog”) |

Or to look at it slightly differently

|  |  |  |  |
| --- | --- | --- | --- |
| **Format** | **1st Param** | **2nd Param** | **3rd Param** |
| HTML | Hyperlink | Tooltip text | Link text |
| MD | Link text | Hyperlink | Tooltip text |

The tooltip text may be optional in either case. The hyperlink and link text must both be present.

We need a Powershell script to extract the three parameters from a Git markdown link and render the equivalent HTML.

For a given HTML file created by Redgate SQLDoc we will want to do the following.

1. Read the content of the file
2. Find all strings that match the Git markdown hyperlink pattern
3. Put each match into an array
4. For each unique instance of a link replace the MD link with the HTML equivalent
5. Write the modified content back into the file

In our ScriptHelpers folder we create an **html\_link.ps1** file as follows.

# The purpose of this script is to replace a git markdown link with an html link.

[cmdletbinding()]

param([string]$inputFileName)

$fileContent = Get-Content $inputFileName

$md\_markdown\_pattern='((?:\[)[\w+\s\*]+(?:\]))\(([a-zA-Z0-9:\/\_\.\?=-]+)(\s)?(\"[\w\s]\*\")? \*\)'

$results=$fileContent|Select-String -Pattern $md\_markdown\_pattern -AllMatches

# We only want unique strings because the replace command takes care of multiple instances

foreach ($item in $results.Matches.Value|Sort-Object|Get-Unique) {

$newItem=$item|Select-String -Pattern $md\_markdown\_pattern -AllMatches

# This is the full markdown link to be replaced

$markdown\_link = $newItem.Matches.Groups[0]

# Build the HTML hyperlink

$html\_link ='<a href="{0}" title="{2}">{1}</a>' -f $newItem.Matches.Groups[2], $newItem.Matches.Groups[1].ToString().Replace("[","").Replace("]",""), $newItem.Matches.Groups[4].ToString().Replace('"','')

# Replace the MD hyperlink with the HTML hyperlink

$fileContent=$fileContent.Replace($markdown\_link,$html\_link)

}

Set-Content -Path $inputFileName -Value $fileContent

The key to this script is to understand the RegEx expression held in the $md\_markdown\_pattern variable. This does assume that the URL for the hyperlink does not contain parameters.

The best way to gain an understanding of the RegEx is to post the expression into <https://regex101.com/>. This online tester provides a good explanation for this RegEx expression and any other RegEx expression you may wish to use.

In short the expression is set to dissect a Git Markdown hyperlink into 4 parts known as groups

* **Group 0** = The entire matched string
* **Group 1** = Everything between a pair of square brackets including the brackets themselves
* **Group 2** = Everything after a left round bracket
* **Group 3** = The space between the Git markdown hyperlink and the end bracket
* **Group 4** = Everything between two double-quote characters

We also need a script file to feed in the HTML files into **html\_link.ps1**

For this we create **md\_to\_html\_link.ps1** which is as follows.

[cmdletbinding()]

param([string]$Foldername)

# Iterate over every HTMNL file in the folder.

Get-ChildItem -Path $Foldername -Recurse -File -Filter "\*.html"|ForEach-Object{

./ScriptHelpers/html\_link.ps1 $\_.FullName

}

Finally, we add the call to our **md\_to\_html\_link.ps1** script to the bottom of the **RunSQLDoc.ps1** file created in STEP ONE.

& ".\ScriptHelpers\md\_to\_html\_link.ps1" "$documentation\_path"

STEP FOUR: Rendering bullet lists in Git markdown files

The Git markdown for tables is fragile. It does not handle multi-line table cells but will render HTML represented on a single line.

For us to render a bulleted list of values from a description from the Redgate Data Catalog dbo.AssetType tables we would need to change the markdown in a description shell from the description below in its unprocessed form.

Describes the type of object within the data catalog that can be categorized  
\* SQL Server Instance  
\* Database  
\* Schema  
\* Table  
\* Column

To get this to render correctly as Git markdown we would have to change it into the following

Describes the type of object within the data catalog that can be categorized<ul><li>SQL Server Instance</li><li>Database</li><li>Schema</li><li>Table</li><li>Column</li></ul>

The content in the previous steps could be handled using a small ruleset and simple replace statements. This meant that editing files could be done using Get-Content/Set-Content.

The rules and handling for bulleted lists is more complex and need to be executed in stages. This dictates that we should output to a 2nd file rather than overwrite the original.

The rules we would use are as follows

* Only Git markdown files will be affected
* Only bulleted lists inside markdown tables will be affected. The affected column is assumed to be the last (description) column
* Any line that is not in a table will be written out unaltered
* Any blank line in an unclosed table will be interpreted as a closing that table.
* Any line that begins as a table cell with contents “Key” or “Name” will signify the opening of a table. This will be written out unaltered and a count of the number of columns will be recorded.
* A count of the number of columns will be the array length when a line is split on the pipe “|” symbol.
* Any table line that has the same number of columns as the count of columns in the first table line will be written out unaltered
* A table line that ends in a table closing pipe symbol will close the table
* A description cell can hold multi-line content other than bulleted lists.

As a design principle we want the code to detect the cases where it should simply yield output and move to considering the next line to execute first.

The code to implement these rules is as shown below.

[cmdletbinding()]

param([string]$inputFileName)

$fileContent = Get-Content $inputFileName

$fileContentLines=$fileContent.Split([Environment]::NewLine)

$tableColumnCount=0

$currentColumnCount=0

$concatenatedHTMLContent=""

$listOpened = $false

ForEach ($line in $fileContentLines){

# Signifies the start of a markdown table

If (($line.StartsWith("| Key |")) -or ($line.StartsWith("| Name |"))){

$tableColumnCount=($line.Split("|")).count

Write-Output $line

continue

}

# Empty line in an existing table means time to close the table

if(($line.Length -eq 0) -and ($tableColumnCount -gt 0)){

$tableColumnCount=0

$currentColumnCount=0

$concatenatedHTMLContent=""

}

if($tableColumnCount -gt 0){

$lineArray=$line.Split("|")

$currentColumnCount=($lineArray).count

}

# If you are not in a table or the table line contains the same number of columns

# As the title line then just print it out.

if(($currentColumnCount -eq $tableColumnCount) -or ($tableColumnCount -eq 0)){

Write-Output $line

continue

}

# Anything beyond this point must be in an open table

if($tableColumnCount -eq $currentColumnCount +1){

$lineStart=$lineArray[0..($lineArray.Length -2)] -join "|"

}

# As the line is split on the "|" the description line

# 1. For a table closing line will be the last but one cell

# 2. for a non-table closing line will be the last cell.

if($line.Trim().EndsWith("|")){

$lastItem=$lineArray[-2].Trim()

}else{

$lastItem=$lineArray[-1].Trim()

}

# Allow for detecting list starts

if (($lastItem.StartsWith("\*")) -or ($lastItem.StartsWith("\_\*"))){

if ($listOpened -eq $false){

$listOpened = $True

$concatenatedHTMLContent+= "{0}</li>" -f $lastItem.Replace("\_\*","<ul><li>").Replace("\*","<ul><li>")

}else{

$concatenatedHTMLContent+= "<li>{0}</li>" -f $lastItem.TrimStart("\*")

}

}else{

# Test that isn't in a list

if($lastItem.Length -gt 0){

if($listOpened -eq $True){

$listOpened = $false

$concatenatedHTMLContent +="</ul>"

}

$concatenatedHTMLContent+="<p>{0}</p>" -f $lastItem

}

}

if($line.EndsWith("|")){

if($listOpened -eq $True){

$listOpened = $false

$concatenatedHTMLContent+="</ul>|"

}else{

$concatenatedHTMLContent+="|"

}

Write-output ("{0}|{1}" -f $lineStart,$concatenatedHTMLContent)

$concatenatedHTMLContent=""

}

}

As the code above works on a single file we need a script that will iterate over the markdown files feeding each one into this script. For this we create a **md\_to\_bullet\_list.ps1** list as follows.

[cmdletbinding()]

param([string]$Foldername)

# Iterate over every HTML file in the folder

Get-ChildItem -Path $Foldername -Recurse -File -Filter "\*.md" -Exclude "tmp\_output.md"|ForEach-Object{

$temporary\_file = Join-Path -Path $\_.DirectoryName -ChildPath "tmp\_output.md"

./ScriptHelpers/MarkDownList.ps1 $\_.FullName > $temporary\_file

# Copy the tmp\_output.md file back over the original file.

Copy-Item -Path $temporary\_file -Destination $\_.FullName -Force

# Clean up the temporary file

Remove-Item -Path $temporary\_file

}

Finally, we add the call to our **md\_to\_bullet\_list.ps1** script to the bottom of the **RunSQLDoc.ps1** file created in STEP ONE.

& ".\ScriptHelpers\md\_to\_bullet\_list.ps1" "$documentation\_path"