PowerShell MiniApps: Demo

Project Name:	PowerShell MiniApps
Template Version:	1.0.0
Template Release Date:	5th Feb 2021
Author:	HerbsRy-Cyber (GitHub)
Repository:	https://github.com/HerbsRy-Cyber/PowerShellMiniApps
Project Contact:	HerbsRyGitHub@GMail.com
Author Website:	www.HerbsRy.com
Tested OS's	Windows 10 1809, 1903, 1909.
Release Notes:	Initial Release (V1)

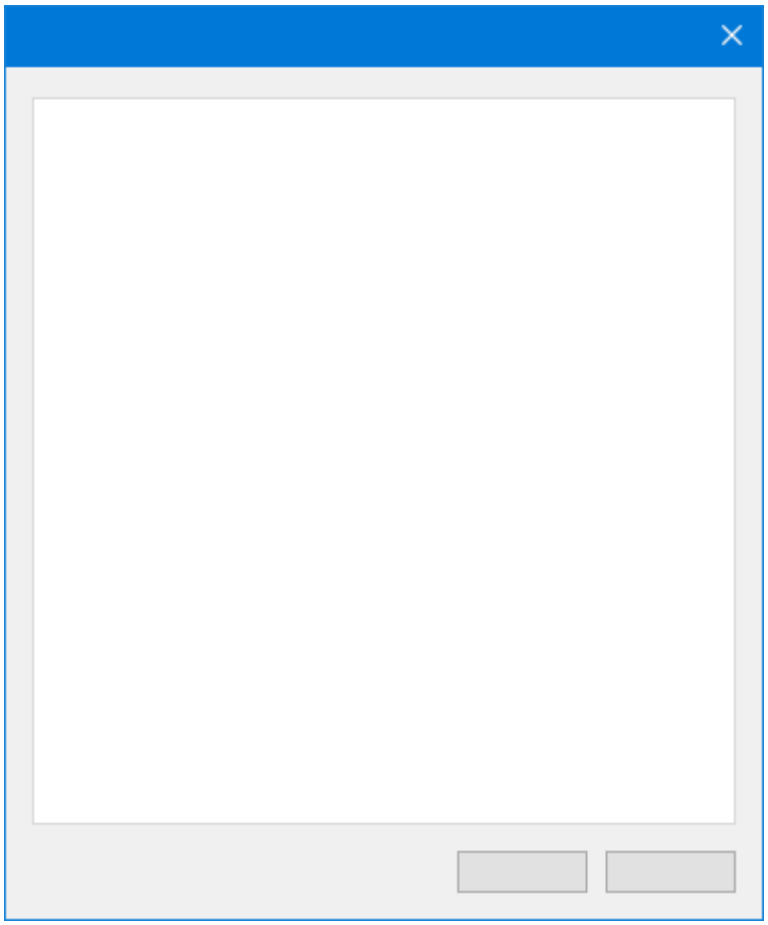
In this demonstration I'll show you how to use the project template file to create a simple GUI application for listing and connecting to Remote Desktop machines. I figured this particular application would be something that we could all relate to and hoped that it wouldn't really need much of an explanation, it's basic but it should give you an idea of what the template can do, the rest my friends… is up to you.

Contents

The Blank Template GUI

The script file, just like any other PowerShell script can be launched with or without the console window visible. The console window can be used to visually see the output and results of the actions performed by the GUI and its related controls. While designing the application I'd recommend you launch the script with a console window present to help you bypass any launch errors such as an execution policy prompt.

When launching the script for the first time with its default values set, you'll be presented with the following Win-Form:



Based on the default values, at minimum we will see the form, no form icon, an empty Caption bar, a basic ToolBox control housing the Close button, a blank client area consisting of a single Panel and two Button controls.

The Close button in the toolbox area will indeed close the form. The two buttons found at the bottom of the form (respectively) are our OK & Cancel Buttons. The button on the left refers to the form's OK button and the button to the right is our form's Cancel button; by default, the Cancel button will close the form.

The form can also be closed by pressing **F4**, or, right clicking the Caption area and selecting the Close option.

From here we will update our *custom* variables (referred to as our AppConfig) one-by-one and slowly start to build our application.

Applying a Form Icon

When applying an Icon (or image) to the form we have a few options, the value that we provide for the image accepts String only, this can be a dynamic string as well as hard coded string.

When the form is loading it will take the values of our icon/image variable(s) and pass them through to a function called `Apply_Image`. The function will first check to see if the value of the image variable is empty, if it is empty then the icon/image will not be applied and no exception will be thrown. If the value is not empty then the function will check to see if the value we have provided matches that of our `$AppConfig_Image_Path_Pattern` RegEx value (default value: `"[A-Z]:\*.*"`), if it does it will attempt to extract the image using the native .NET method called `ExtractAssociatedIcon()`, and if successful from here it will apply the icon. If the value provided does not match our regular expression, the function will presume that you are passing through a Base64 string value and will attempt to decode the string and apply the image to the form, if the value is not valid Base64 an exception will be thrown to the console window – from here you can resolve any issues.

Once we have decided that we want to display an icon we must first locate `$AppConfig_Show_Form_Main_Icon` and set its value to `$True`, this will enable the icon. If we choose to show the form icon but we don't explicitly provide a value then the native .NET form icon will be shown, as below:

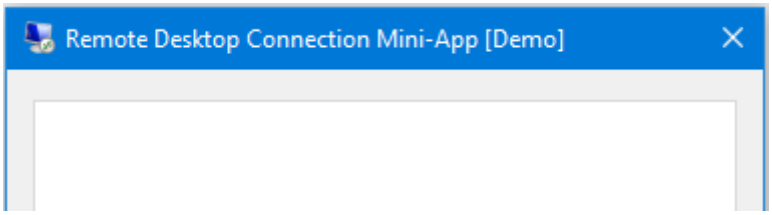
The default template value (albeit initially commented out) for the form icon is `(Get-Command PowerShell).Path`, this resolves to an absolute path such as C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe, therefore is an accepted string path and will apply the form icon as shown below (assuming you uncomment the value):

Although the application we are building is created using PowerShell, I prefer to add a more representable image based on the applications use. To do this we can simply change the value of `$AppConfig_Image_Form_Main_Icon` to `(Get-Command MSTSC).Path`.
PowerShell will extract the absolute path of mstsc.exe and pass it through to the `ExtractAssociatedIcon()` method, the result will be as below:

Applying a Form Caption

To apply a form caption we need to update the value of `$AppConfig_Application_Name`, the GUI will now display the application name:

```
[String]$AppConfig_Application_Name = "Remote Desktop Connection Mini-App [Demo]"
```

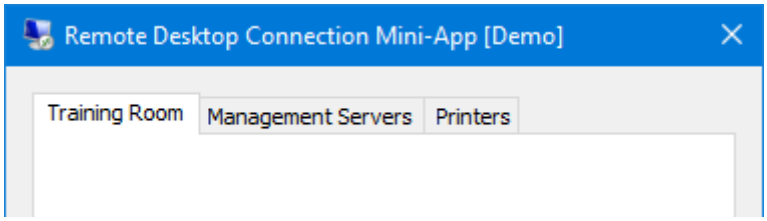


Configure the Tab System

There are 3 tabs in which we can enable on our Tab Control, for this demo we will enable all 3 but only configure one of them fully (Tab 1).

```
[Boolean] $AppConfig_Show_Tab_1          = $True
[Boolean] $AppConfig_Show_Tab_2          = $True
[Boolean] $AppConfig_Show_Tab_3          = $True

[String]  $AppConfig_Tab_Page_1_Tab_Text = "Training Room"
[String]  $AppConfig_Tab_Page_2_Tab_Text = "Management Servers"
[String]  $AppConfig_Tab_Page_3_Tab_Text = "Printers"
```

We first need to enable each tab by modifying the value of the above `Boolean` variables and setting the values to `$True`.
For each enabled tab we must provide a text value that represents the tab, to do this we must update the related Tab `String` values, as above.
Our application interface should now have applied the changes, as below:



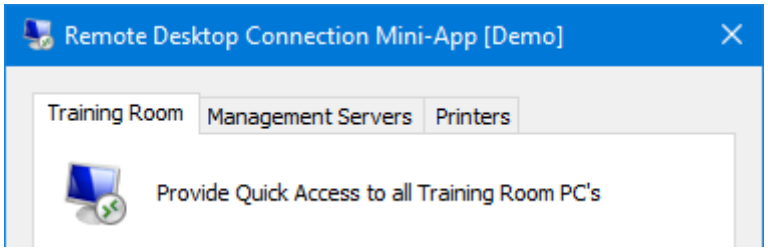
For each tab that is enabled you will see a CheckedListBox control and a GroupBox control inside the body of the tab, we'll cover these controls shortly.

Tab Icons and Label Headings

To add more meaning to an individual tab we can apply an image to help represent it, as with the form icon we apply the same process, we can either use a string to extract an icon or apply our own Base64 image. For this demo we'll stick with the simple extract method as we did previously.

Locate the `$AppConfig_Image_Tab_Page_1_Picture_Box` variable and update the value to `(Get-Command MSTSC).Path`.

Next, we can add a string description that will display next to the image to give the tab even more meaning. If we update the string value of `$AppConfig_Tab_Page_1_Label_Heading_Text` to `"Provide Quick Access to all Training Room PC's"` our tab should now display our image and text, as below:



Configuring the File Menu

Depending on the application that we're designing we may want to add a FileMenu control that allows us some additional options to perform tasks or trigger our custom code.
To enable the FileMenu we need to locate `$AppConfig_Show_File_Menu` and set the value to `$True`. Once we have enabled the FileMenu we then need to configure each sub menu. Each sub menu has a ton of controls that we can manipulate by modifying their related variables.

For this demo I will enable all 5 top-level menus for the purpose of cosmetics, I will only configure some of the sub-menus.
After enabling each top-level menu, I will then set the sub-menus that I want to display as visible, I'll configure the text values, I'll add a base64 image to some of the menus and we'll link each sub-menu to a function that triggers *your* custom code.
The file menu is a bit of a beast so I'll try to keep it simple enough to show some of its main traits without going too deep.

First, we will enable all 5 top-level File Menu's. The first number in each variable represents the number of the menu shown from left to right, followed by '0' which represents the Top-Level menu rather than a sub-menu, as below:

```
[Boolean] $AppConfig_File_Menu_1_0_Enabled = $True
[Boolean] $AppConfig_File_Menu_2_0_Enabled = $True
[Boolean] $AppConfig_File_Menu_3_0_Enabled = $True
[Boolean] $AppConfig_File_Menu_4_0_Enabled = $True
[Boolean] $AppConfig_File_Menu_5_0_Enabled = $True
```

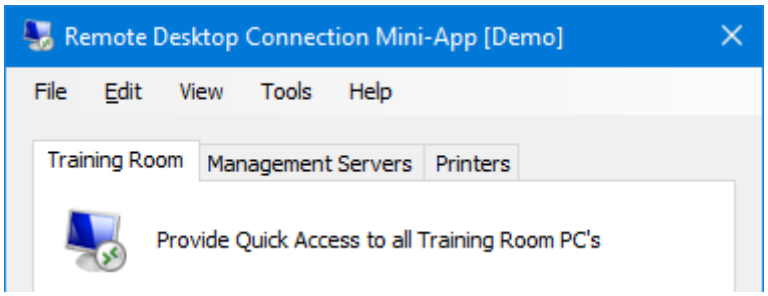
Next, we will need to specify that the same controls should be Visible, we can do so by updating the below values:

```
[Boolean] $AppConfig_File_Menu_1_0_Visible = $True
[Boolean] $AppConfig_File_Menu_2_0_Visible = $True
[Boolean] $AppConfig_File_Menu_3_0_Visible = $True
[Boolean] $AppConfig_File_Menu_4_0_Visible = $True
[Boolean] $AppConfig_File_Menu_5_0_Visible = $True
```

Next, we need to add some text to the FileMenu objects, we'll go for a classic layout:

```
[String]  $AppConfig_File_Menu_1_0_Text = "File"
[String]  $AppConfig_File_Menu_2_0_Text = "&Edit"
[String]  $AppConfig_File_Menu_3_0_Text = "View"
[String]  $AppConfig_File_Menu_4_0_Text = "Tools"
[String]  $AppConfig_File_Menu_5_0_Text = "Help"
```

The result is:



Notice that there's an ampersand in the value of `$AppConfig_File_Menu_2_0_Text` and notice that when we press **Alt** on the keyboard, we see an underscore under the letter E on the Edit menu… For those of you that like to navigate your applications using shortcuts you have the option to use this method if you wish, simply place an ampersand anywhere in the string and the letter to its immediate right will become the shortcut key.
Now that we have our Top-Level FileMenu's visible we'll configure a small set of sub-menus as follows:

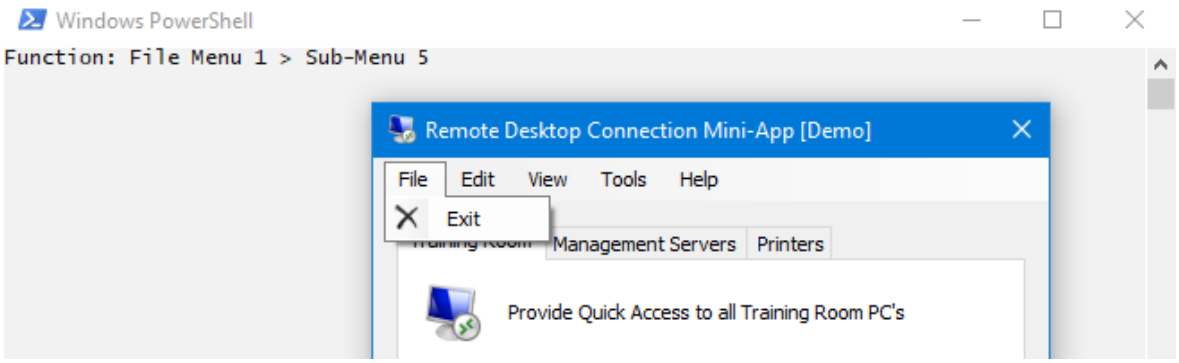
- File Menu 1 = File > Exit
- File Menu 3 = View > RDP History
- File Menu 5 = Help > mstsc.exe
- File Menu 5 = Help > About

File Menu 1
We'll configure the exit sub menu to be the 5th in the list (that's where you usually find the exit command), this means that in future when we add more commands, we don't have to move things around, it will never really need to move from the 5th position, however, because this is the only sub menu that we are configuring under File Menu 1, it will display as the only menu.
We'll update the following to view the results below:

```
[Boolean] $AppConfig_File_Menu_1_5_Enabled = $True
[Boolean] $AppConfig_File_Menu_1_5_Visible = $True
[String]  $AppConfig_File_Menu_1_5_Text    = "Exit"
[String]  $AppConfig_Image_File_Menu_1_5   = "iVBORw0KGgoAAAAN..."
```

The value of `$AppConfig_Image_File_Menu_1_5` is a (cut short) long Base64 string, feel free to replace it with your own.
When you do work with Base64 strings you will either need to add transparent pixels to the image before encoding it, or, rely on the value of the `$AppConfig_Image_Transparency_Colour` variable, which by default is set to `Magenta`, meaning, any pixels in your Base64 decoded image that match this colour will be automatically set to transparent.

Now when we launch the application, we'll see our new File > Exit sub-menu, and when we click on the exit menu you'll notice that the console window displays output:



The output in the console is the result of code that is already connected to each button. This output shows that a function was triggered by clicking on File Menu 1 > Sub-Menu 5.
If we were to locate this function it will look like the below:

```
Function File_Menu_1_5 { Write-Host "Function: File Menu 1 > Sub-Menu 5" }
```

This function shows us that when it's triggered by any means, it will write back to the console.
It is these functions that are managed by yourself, the template provides a simple write host mechanism to help you confirm you are clicking on the correct button then allows you to replace this code with your own.
For example, when we click the newly created Exit button, we want to application to Exit, we don't want it to display a message to the window so, let's remove the Write-Host command and replace it with our own.

We can do this easily by leveraging an existing application function called `Form_Main_Close` which will close down our application for us. Take a look at the modified function:

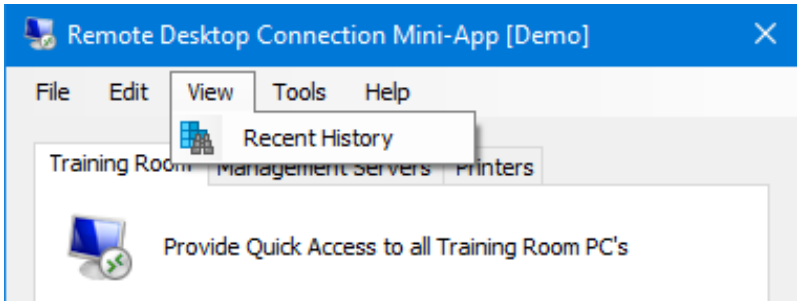
```
Function File_Menu_1_5 { Form_Main_Close; }
```

Now when we click on our Exit sub-menu the application will close. Note that although we have not covered the Cancel button yet, the code that sits behind the Cancel button is exactly the same, it simply makes a call to the `Form_Main_Close;`function.

File Menu 3
This time we'll be editing FileMenu_3_3 properties as follows:

```
[Boolean] $AppConfig_File_Menu_3_3_Enabled = $True
[Boolean] $AppConfig_File_Menu_3_3_Visible = $True
[String]  $AppConfig_File_Menu_3_3_Text    = "Recent History"
[String]  $AppConfig_Image_File_Menu_3_3   = "iVBORw0KGgoAAAAN..."
```

The result is:



Next, it's on us to provide the code to the related function to trigger something special. For the purpose of this demo, I won't provide the code but I will provide a sample result from a basic script that queries the registry under my user account to display the last 5 machines that I have connected to.

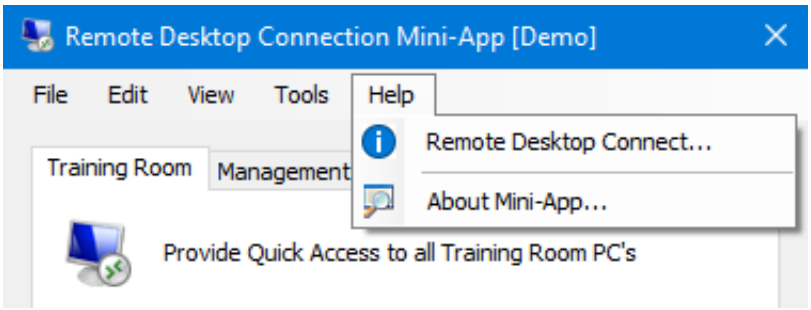
```
Property Value
-------- -----
MRU1     SERVER-01
MRU2     SERVER-02
MRU3     SERVER-03
MRU4     SERVER-04
MRU5     SERVER-05
```

The code inside the function is all yours, you could decide to add the code directly in to the function, or, decide to have the function call and execute an external script instead. The bottom line is that you're in control of the actions performed by the custom buttons.

File Menu 5

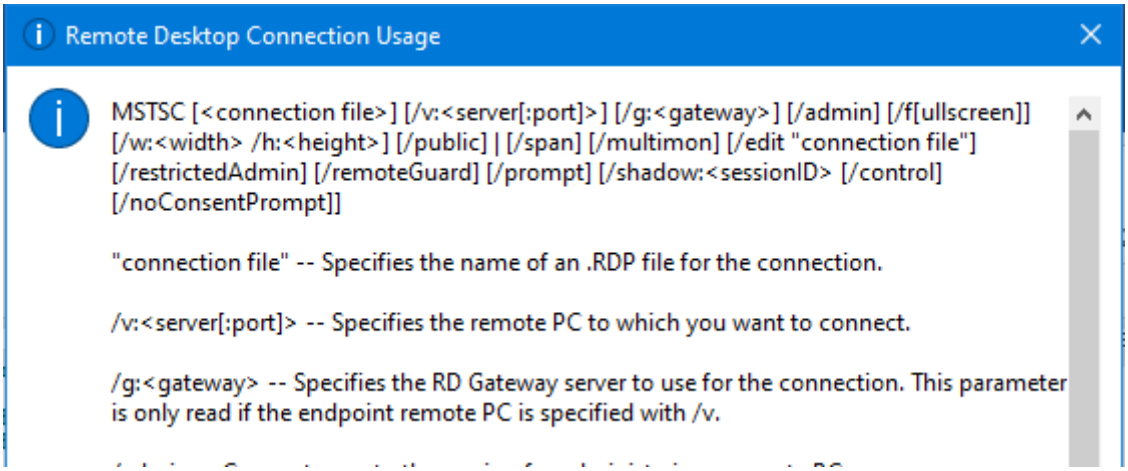
Typically, inside a FileMenu you'll have a menu labelled *Help*, you'll then usually see a sub-menu labelled *About*. We'll be creating the same here, we'll also add a 2nd sub-menu to the Help menu so that we can make use of a FileMenu separator control and to try to give you more ideas as to what you can cram in to your FileMenu to make it useful to the agent that will be using it.

```
[Boolean] $AppConfig_File_Menu_5_3_Enabled           = $True
[Boolean] $AppConfig_File_Menu_5_5_Enabled           = $True

[Boolean] $AppConfig_File_Menu_5_3_Visible           = $True
[Boolean] $AppConfig_File_Menu_5_5_Visible           = $True

[String]  $AppConfig_File_Menu_5_3_Text              = "Remote Desktop Connect..."
[String]  $AppConfig_File_Menu_5_5_Text              = "About Mini-App..."

[Boolean] $AppConfig_File_Menu_5_4_Separator_Visible = $True

[String]  $AppConfig_Image_File_Menu_5_3             = "iVBORw0KGgoA.."
[String]  $AppConfig_Image_File_Menu_5_5             = "iVBORw0KGgoA..."
```



Connected to sub-menu 5_3 we could place a simple command such as `mstsc /?` Which when clicked would display the following:

```
Function File_Menu_5_3 { mstsc /? }
```

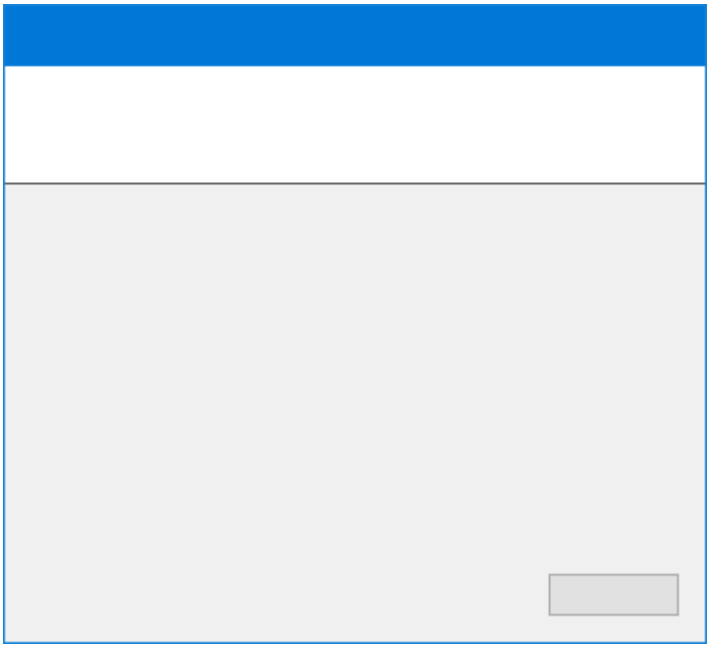


Connected to the 'About Mini-App' sub-menu we can make a call to display our About form, it's as simple as the following:

```
Function File_Menu_5_5 { Form_About }
```

The About Form

The about form can be triggered in 2 ways, it can either be triggered by making a call to a function named `Form_About`, or it can be triggered by enabling a variable called `$AppConfig_Hook_Button_F1_To_About_Form` then pressing **Alt+F1** at any time while the main application form has focus.
When the about form loads with its default empty settings, you'll see the following:



The only functioning control by default is the OK button in the bottom right corner – this button closes the form. There's no option to set a form icon here. The form is set as a modal over the main form, this means that you won't be able to give the main form focus until you close the About form.

I'll edit all of the available variables for this form but only briefly cover them in detail as a lot of them should be self explanatory once you start to play with their values.

```
$AppConfig_Application_Version = "1.0.0"
```

This variable is not required but it offers a chance at the top of the script to stamp your application version in a single place allowing you to refer to it from other variables

```
$AppConfig_Form_About_Title_Text = "About " + $AppConfig_Application_Name
```

This variable refers to the Caption text on the form

```
$AppConfig_Form_About_Label_Heading_1_Text = "Remote Desktop Connection Mini-A..."
```

This variable refers to the Blue label that you see when enabled

```
$AppConfig_Form_About_Label_Heading_2_Text = "Created by: HerbsRy-Cyber"
```

This variable refers to the second heading displayed underneath the Blue heading text. If this heading is not enabled then the Blue heading will be more centralised vertically within the white panel header.

```
$AppConfig_Form_About_Label_Heading_3_Text = "Application Version: " +
$AppConfig_Application_Version
```

This variable refers to the light grey text above the description in the main body which contains the version number on the image below. This heading can be disabled / hidden.

```
$AppConfig_Form_About_Link_Label_Text = "GitHub Repository"
```

This variable refers to the text that's displayed on the link in the bottom left corner

```
$AppConfig_Form_About_Link_Label_URL = "https://github.com/herbsry-..."
```

This variable contains the actual URL in which you'll navigate to when clicked.

```
$AppConfig_Form_About_Button_OK_Text = "Got it!"
```

This variable refers to the text value that displays on the OK button

```
$AppConfig_Form_About_Label_Main_Text = "Lorem ipsum..."
```

This variable refers to the text in the main description area. If the text reaches a certain number of lines vertically then a vertical scrollbar will appear.
If you want to use multiple lines / line breaks you will need to use the PowerShell methods of `r`n which will cause line breaks and new lines. Example:
"Line One.`r`n" + "Line Two.`r`n" + "Line Three"

```
$AppConfig_Form_About_Show_Label_Heading_2 = $True
```

This variable enables Heading 2 and displays it

```
$AppConfig_Form_About_Show_Label_Heading_3 = $True
```

This variable enables and displays Heading 3

```
$AppConfig_Form_About_Show_Link_Label      = $True
```

This variable will enable and display the custom LinkLabel control.

```
$AppConfig_Hook_Button_F1_To_About_Form                = $True
```

This variable allows you to switch on a keypress trigger, when enabled you can access the About form by pressing Alt+F1.

```
$AppConfig_Form_About_Panel_Header_Label_1_Colour      =
[System.Drawing.Color]::RoyalBlue
```

This variable allows you to change the colour of the control

```
$AppConfig_Form_About_Panel_Header_Label_2_Colour      =
[System.Drawing.Color]::FromArgb(64, 64, 64)
```

This variable allows you to change the colour of the control

```
$AppConfig_Form_About_Label_Heading_3_Colour           =
[System.Drawing.Color]::FromArgb(64, 64, 64)
```

This variable allows you to change the colour of the control

```
$AppConfig_Form_About_Link_Label_Active_Link_Colour    =
[System.Drawing.Color]::RoyalBlue
```

This variable allows you to change the colour of the control

```
$AppConfig_Form_About_Link_Label_Link_Colour           =
[System.Drawing.Color]::RoyalBlue
```

This variable allows you to change the colour of the control

```
$AppConfig_Form_About_Link_Label_Visited_Link_Colour =
[System.Drawing.Color]::RoyalBlue
```

This variable allows you to change the colour of the control

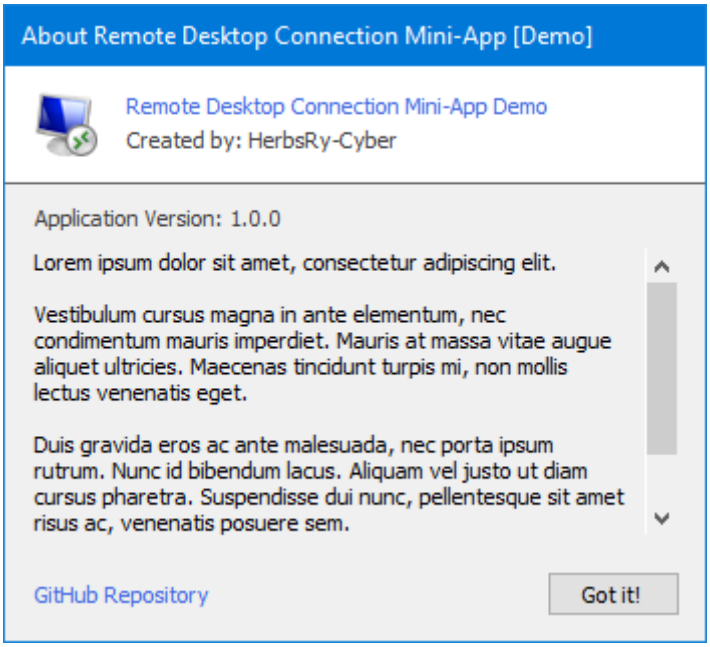
```
$AppConfig_Form_About_Link_Label_Link_Behavior         =
[System.Windows.Forms.LinkBehavior]::HoverUnderline
```

This variable allows you to change the behaviour of the control

```
$AppConfig_Image_Form_About_Picture_Box = (Get-Command MSTSC).Path
```

This variable controls the image / Icon that displays inside the white panel.

If we applied all of the settings above, our results would be as follows:



Configuring the Quick Access Toolbar

The option to enable and use a classic ToolBar control is available to you if you decide to use it. Its configuration is similar to the FileMenu controls however there's only one Toolbar and only 10 buttons, there are no sub-menus to contend with.
Let's set something simple up with 2 buttons.

```
[Boolean] $AppConfig_Show_Tool_Strip                    = $True
```

Enables and displays the Tool Strip control

```
[Boolean] $AppConfig_Show_Tool_Strip_Grip_Handle        = $True
```

Enables and displays the Grip Handle (cosmetic purposes only)

```
[Boolean] $AppConfig_Tool_Strip_Button_1_Enabled        = $True
[Boolean] $AppConfig_Tool_Strip_Button_2_Enabled        = $True
```

Enables the buttons

```
[Boolean] $AppConfig_Tool_Strip_Button_1_Separator_Visible  = $True
[Boolean] $AppConfig_Tool_Strip_Button_2_Separator_Visible  = $True
```

Displays the separator line between each button (to the immediate right of a button)

```
[String]  $AppConfig_Image_Tool_Strip_Button_1          = "iVBORw0..."
[String]  $AppConfig_Image_Tool_Strip_Button_2          = "iVBORw0..."
```

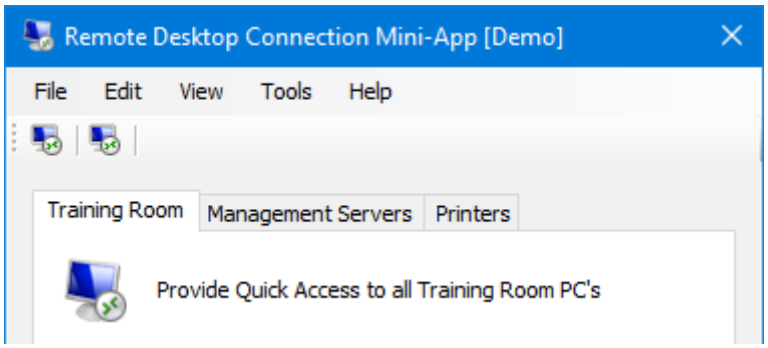
Images that will be displayed on the button face. If no image is present the button may look like it's not there – Buttons require images to be visibly seen because the ToolBar buttons do not have a border.

```
[String]  $AppConfig_Tool_Strip_Button_1_Tool_Tip_Text  = ""
```

Allows you to construct a ToolTip bubble when hovering over a button, setting this value to a blank string will cause no tool tip to display

```
[Object]   $AppConfig_Tool_Strip_Button_1_Margin = New-Object ...Padding(0,1,0,2)
```

Allows you to adjust the margin of the button by points. As some images may not be centred by design it may cause the buttons to look un-centred or none-symmetrical on the left and right edges - this allows you to nudge a button over by a point or two to make the toolbar neater.
There are plenty of variables that you can play with to build a nice-looking toolbar.



As with the FileMenu controls, when clicking a ToolBar button, you will see output written to the screen confirming the button you have pressed and the function name. You can use this output to find the related Function and modify the function code to suit your requirements so that when you click a button the application triggers your custom code.

Side note: when creating images for your quick access toolbar I would suggest putting in the time to test different colour pallets. We each have our own opinion and each application is different but I don't like bold colours on these classic .NET Tool Strips. I personally feel an ugly constructed toolbar can easily make or break the professional feel of an application. Choose wisely!
Take the above example, a more professional look would be to first choose different images for the two buttons, or two different colour pallets to help represent each button's meaning, then, we could also remove the second separator as it's not really required.

With that said, we'll leave the demo toolbar as it is and move on the CheckedListBox control.

Configuring the CheckedListBox & HashTables

Enter the CheckedListBox control.
We will concentrate only on the list for Tab 1, however all 3 tabs would be set up in the same way.
To set up the CheckedListBox and populate some CheckBoxes we'll need to locate the HashTable for *Items* and the associated HashTable for *Descriptions*:

By default, there are 20 items pre-configured for each tab at your disposal, albeit, they are initially commented out. For the purpose of this document, I have trimmed the number of items shown in the example below to make for easier reading.

```
$AppConfig_Hash_Table_Tab_1_Items = [Ordered]@{

    #"0"  = "";
    #"1"  = "";
    #"2"  = "";
    #"3"  = "";
    #"4"  = "";
    #"5"  = "";
    #"6"  = "";
    #"7"  = "";
    #"8"  = "";
    #"9"  = "";

}
$AppConfig_Hash_Table_Tab_1_Description = [Ordered]@{

    #"0"  = "No description available for this item.";
    #"1"  = "No description available for this item.";
    #"2"  = "No description available for this item.";
    #"3"  = "No description available for this item.";
    #"4"  = "No description available for this item.";
    #"5"  = "No description available for this item.";
    #"6"  = "No description available for this item.";
    #"7"  = "No description available for this item.";
    #"8"  = "No description available for this item.";
    #"9"  = "No description available for this item.";

}
```

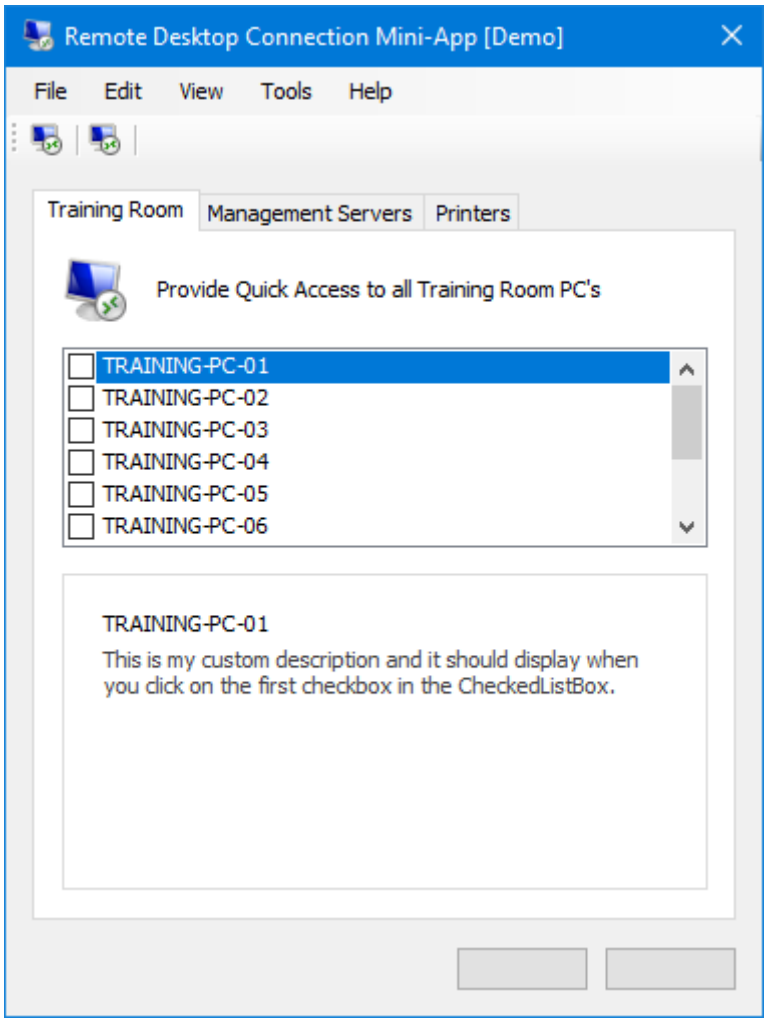
The 2 HashTables for Tab 1 are as above. Both HashTables use a Zero-Based Indexing system, meaning, the first item in the list is item Zero (not item 1).
You should also note that although these 2 HashTables are not physically linked, they are logically and our application will rely on this logical link.
You can think of them as 2 SQL tables that connect to each other by a primary key system, meaning, item 0 in `$AppConfig_Hash_Table_Tab_1_Items` logically connects to item 0 in `$AppConfig_Hash_Table_Tab_1_Description`.
The items in `$AppConfig_Hash_Table_Tab_1_Items` contain the text value of a checkbox that we want to load inside the application interface, whereas the items in `$AppConfig_Hash_Table_Tab_1_Description` contain the text description for the related checkbox in the description area of the application.

Let's populate some checkboxes...

Here's a very quickly modified set of HashTables that will populate 10 checkboxes, each representing a different PC name:

```
$AppConfig_Hash_Table_Tab_1_Items = [Ordered]@{

    "0"  = "TRAINING-PC-01";
    "1"  = "TRAINING-PC-02";
    "2"  = "TRAINING-PC-03";
    "3"  = "TRAINING-PC-04";
    "4"  = "TRAINING-PC-05";
    "5"  = "TRAINING-PC-06";
    "6"  = "TRAINING-PC-07";
    "7"  = "TRAINING-PC-08";
    "8"  = "TRAINING-PC-09";
    "9"  = "TRAINING-PC-10";

}
$AppConfig_Hash_Table_Tab_1_Description = [Ordered]@{

    "0"  = "This is my custom description...";
    "1"  = "No description available for this item.";
    "2"  = "No description available for this item.";
    "3"  = "No description available for this item.";
    "4"  = "No description available for this item.";
    "5"  = "No description available for this item.";
    "6"  = "No description available for this item.";
    "7"  = "No description available for this item.";
    "8"  = "No description available for this item.";
    "9"  = "No description available for this item.";

}
```

This is the result when we relaunch the application:



Notice that the correct description displays for the item that we have selected in the CheckedListBox. Each item selected will now display its own description.

Attention!: *Drawbacks to using these HashTables alongside dynamically populating a CheckedListBox control. When creating or modifying your HashTable list, always re-order the index numbers in a sequential order starting with 0. If this rule is not followed you will find that the description falls out of sync with the checkbox selected. For example, do not order your items in the following manner:*

```
"0" = "TRAINING-PC-01";
"2" = "TRAINING-PC-03";
"3" = "TRAINING-PC-04";
```

If the above was to be configured, the checkboxes will still display in the correct order graphically but you'll find that when you select item 2 (Training-PC-03) you possibly won't see a description because although TRAINING-PC-03 is the second graphical checkBox in the list it will have received a CheckedListBox number of 1 (Zero-Based), therefore, when you select that item the description will be looking for an item in the HashTable with a key value of 1, which of course doesn't exist.
The reason is that regardless of the number of items in your HashTable and regardless of the numbering system we apply, when the CheckedListBox populates for the first time it will add each item from the HashTable as a CheckBox and it will automatically assign its own internal index number to that CheckBox and it may not match the number of the HashTable index. Unfortunately,

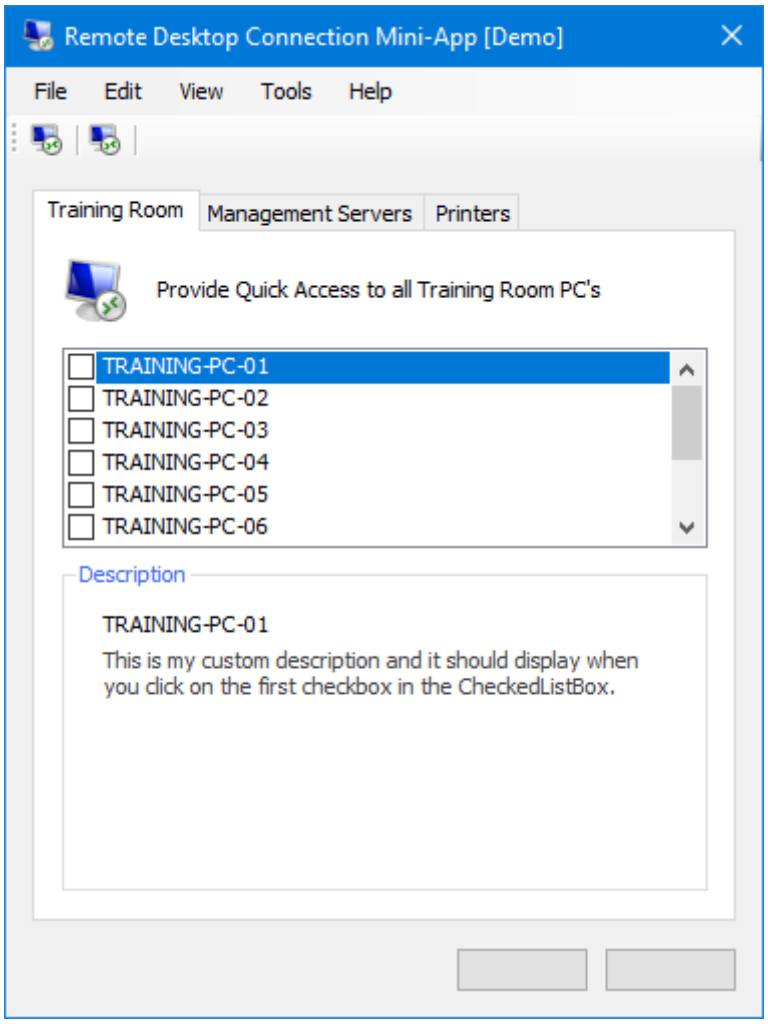
that will break the description based on how our description code works because our code updates descriptions based on the CheckBox which needs to match the HasTable order. The bottom line is, always confirm that your HashTable indexes start at 0 and there are no missing numbers – always keep a consistent sequential number in your index.
Please always test your application thoroughly before deploying it and make sure that the correct description is displayed when selecting an item as it could potentially cause issues if the wrong description is displayed and the user presses to OK button….

In the Description area of the GUI you will see the description extracted from it's associated record from within its parent HashTable, as the description grows in size vertically you should expect to see a vertical scrollbar appear to accommodate the text being pushed out of view, when the scrollbar appears it may dynamically rearrange the text slightly to make the text fit neatly inside its parent container – nothing to worry about, just an FYI.

Another way in which we can customise the GUI is to add a Description text header, as follows:

```
[String]  $AppConfig_Group_Box_Description_Text = "Description"
```

If this variable is left blank then no text will display and the border around the description area will remain a solid line. If the value is modified to contain text such as above then the text will now appear in the top left corner of the description area. Note that setting this in one place will activate the same text for each available tab (one description fits all). The result is as below:



As you can see, we now see the word 'Description' in blue just above the description area. This colour can also be quickly changed to something that may be more fitting to your application, such as Black, by modifying the following variable.

```
$AppConfig_Group_Box_Heading_Text_Colour = [System.Drawing.Color]::RoyalBlue
```

Configuring the Custom Label

There's a hidden label in the bottom left corner of the form that we can use to show some helpful text, such as the logged-on user name, application version, the day of the week…whatever you choose. It could display "Press Alt + F1 for help" which of course would prompt the user to load the About form where you dump release notes for each version of the application.

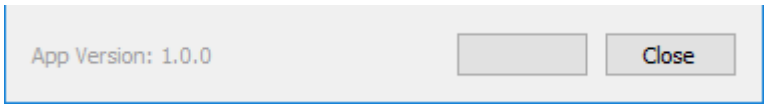
We will drop something in there just now by modifying the `$AppConfig_Label_Custom_Text` variable and enable / show the label by modifying `$AppConfig_Show_Label_Custom`, as follows:

```
$AppConfig_Show_Label_Custom = $True
$AppConfig_Label_Custom_Text = "App Version: " + $AppConfig_Application_Version
```

Configuring the Cancel Button & MessageBox

Given that the Cancel button closes the application, this one's a no-brainer, all we need to do is add some text to the button, as follows:

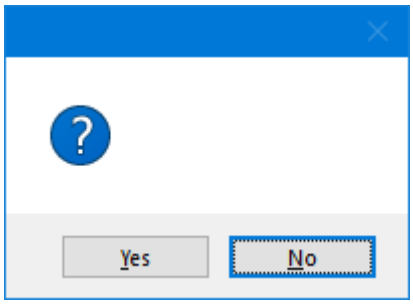
```
[String]  $AppConfig_Button_Cancel_Text = "Close"
```



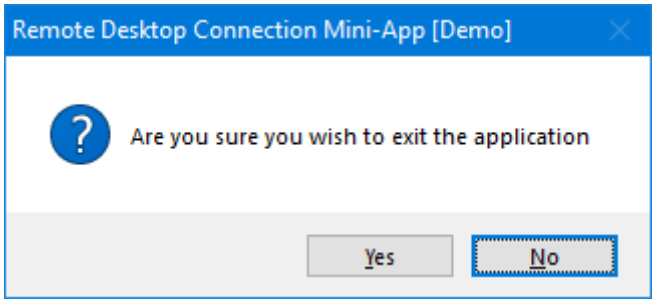
Now, depending on the nature of the application, we may not want to the application to instantly close when we click Close, rather, you may want to display a MessageBox with a confirmation prompt, annoyingly asking the user "are you sure you want to close this application?". In the event your application requires such a feature we can easily do so by targeting the following 3 variables

```
[Boolean] $AppConfig_Hook_Button_Cancel_Message_Box          = $True
[String]  $AppConfig_Message_Box_Button_Cancel_Title_Text    = ""
[String]  $AppConfig_Message_Box_Button_Cancel_Body_Text     = ""
```

If we set the value of `$AppConfig_Hook_Button_Cancel_Message_Box` to `$True` and attempt to close the application we are now presented with the following message box:



If we now update `$AppConfig_Message_Box_Button_Cancel_Title_Text` and `$AppConfig_Message_Box_Button_Cancel_Body_Text` with our custom string our MessageBox will now display as:

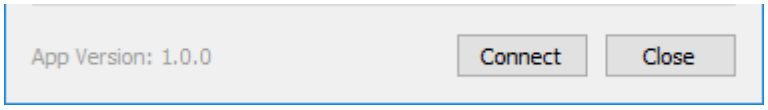


Pressing Yes will close the application while pressing No will close the MessageBox and the main form will regain focus.

When enabling the MessageBox, it will appear not only when the close button is pressed but also when F4 is pressed, or, when the application toolbox close button is pressed.

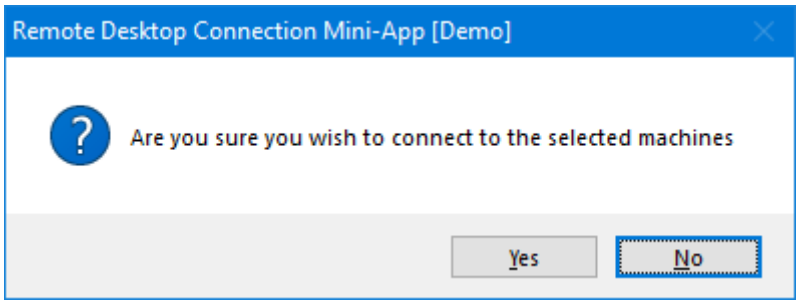
Configuring the OK Button

Just like the Cancel Button, we can customise the text by configuring the `$AppConfig_Button_OK_Text` variable, as follows:



When we click the OK (Connect) button, the application will check to see which tabs are visible, then for each tab we can see, the application will iterate through each checkbox to see if any of them are checked. If any checkbox is checked an internal variable flag will be set to $True.
If the variable is set to $True then the application will know that we are expecting to trigger some code, therefore, just to confirm we are happy with our selection, another MessageBox will display. This MessageBox also allows us to configure it with the following variables, as follows: (note that if no checkboxes are selected then this MessageBox will not display, the form will regain focus taking focus away from the button and no code will be triggered.)

```
[String] $AppConfig_Message_Box_Button_OK_Title_Text
[String] $AppConfig_Message_Box_Button_OK_Body_
```



If we select Yes on this MessageBox then the application code calls the `Form_Main_Button_OK_Master_Switch` function.
At this point if you are following along, select any random checkbox on Tab 1 and press OK, then select Yes on the MessageBox – you'll see that the console window displays the output and will confirm that it has actually run the function code related directly to that checkbox.

The next step after clicking OK (Connect) then Yes > The Master Function will do the following:

- Checks the value of `$AppConfig_Master_Function_Disable_Button_OK`, if set to `$True` the OK Button will be disabled while our code runs, if set to `$False` nothing will happen to the button.
- Checks the value of `$AppConfig_Master_Function_Disable_Button_Cancel`, if set to `$True` the Cancel Button will be disabled while our code runs, if set to `$False` nothing will happen to the button.
- Enter a Switch statement that looks for the index key to the checkbox that is selected. The switch locates the index and triggers the aptly named function. If we look at Checkbox 0 on Tab 1, the Function name will be `Tab_1_Checked_List_Box_Item_0` and by default it outputs `"Function: Tab 1 > Checked List Box Item 0"` to the console window – this is the function / code that you must update to make your Checkbox trigger custom code (just as we did with the File Menu functions and Toolbar button functions). If you do add more than 20 items to any given HashTable then you'd first want to add those additional triggers to the switch statement and then create the related function name.
- After the switch statement has triggered the function and the function exits returning back to the switch, the following variable will be queried: `$AppConfig_Master_Function_Sleep_Timer` and if customised to a number other than 0, that number of Milliseconds will be counted before the Switch statement continues to work through and trigger any additional functions based on the checkboxes you have selected. The default value is 0. Updating this to 2000 would make the application wait 2 seconds before triggering code for each checkbox, and the checkbox after that, and so on. This can either give commands breathing space, or simply allow you to control and view the results better.
- When the master function executes code, it will do so in this order (if the Tab is visible)
    - Tab 1
        - CheckBox1
        - Checkbox2
        - Checkbox3
            - Etc..
    - Tab2
        - CheckBox1
        - CheckBox2
        - CheckBox3

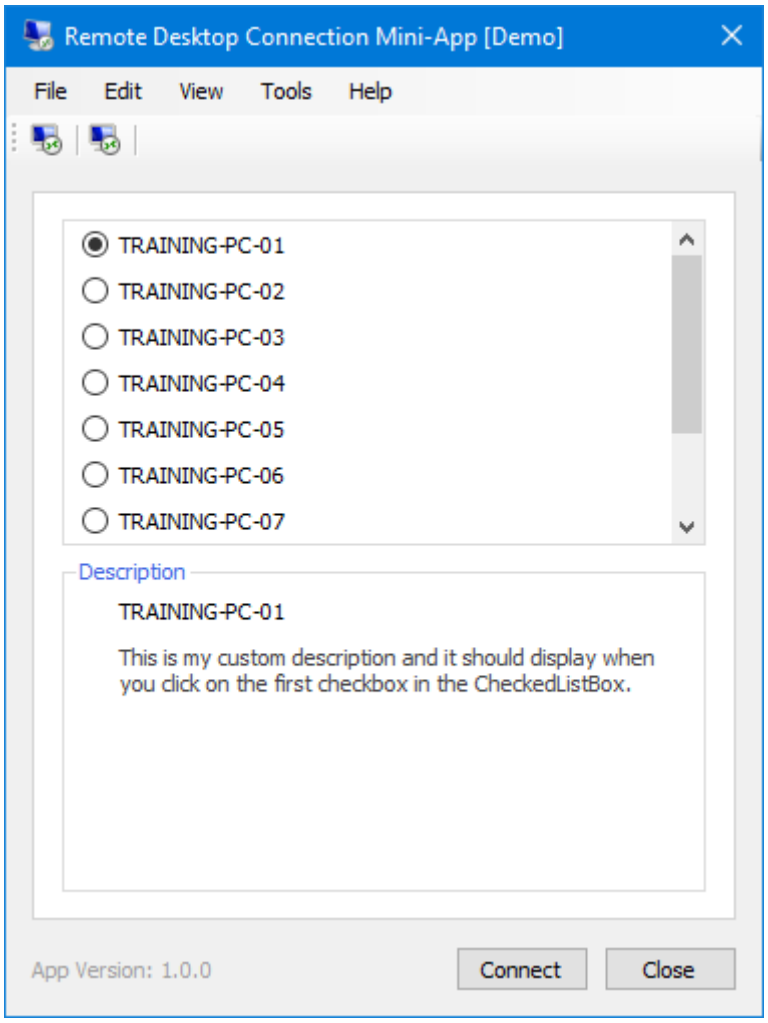
        - Etc..
    - Tab 3
        - CheckBox1
        - CheckBox2
        - CheckBox3
            - Etc..
- The function will then check against `$AppConfig_Master_Function_Clear_All_Check_Boxes` and if set to `$True`, all selected CheckBoxes will be unchecked. If set to `$False`, all selected checkboxes will remain checked. This option allows you to return back to a cleaner application and may mitigate against running the same code again if you were to select the Connect button again.
- The final 2 checks are made against the OK and Cancel buttons: if they are disabled then at this point, they will be re-enabled and the main form will regain focus. Disabling the OK and Cancel buttons helps mitigate against double pressing them and triggering the same code twice but as with any other custom variable in our AppConfig it's completely up to you how you manage the behaviour of the controls.

All that's left to do is locate the function behind each checkbox and add the PowerShell command that will connect to the machine you intend to connect to. The floor is yours.

Enforcing the Radio-Style GUI

So far, we have worked with a Tab Style interface with CheckBoxes allowing for Multi-Select items. There may be times where we don't want to give the user of the application such luxury, instead we may want to give them a more restricted interface that does not allow Multi-Select, rather, we want to present the user with single select option and no way of cheating the system (forcing them to trigger one command at a time).

Enter `$AppConfig_Radio_GUI_Enforced`.
When this variable is set to `$True`, the tab system will not be loaded when the application launches and it will be replaced with a single panel area containing a GroupBox control that houses a grouped set of Radio Buttons. If we set this variable value to `$Tue` then populate both `$AppConfig_Hash_Table_Radio_GUI_Items` and `$AppConfig_Hash_Table_Radio_GUI_Description` HashTables then launch the application - here's what we have.

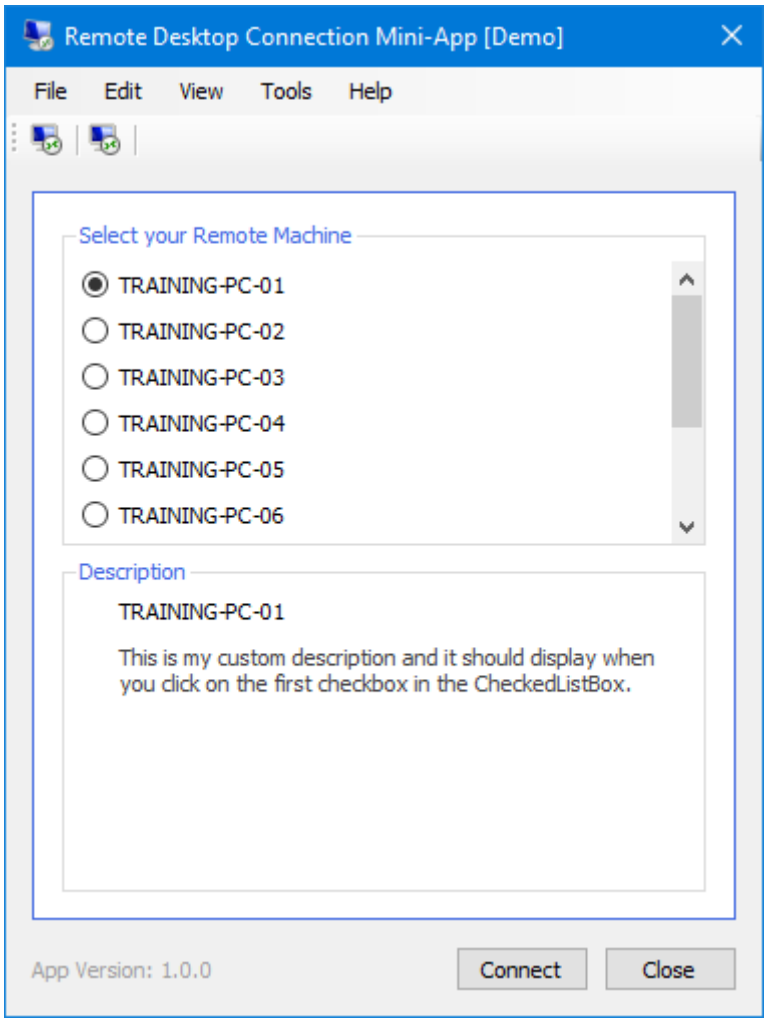


The multi-tab-multi-checkbox system has been replaced with a single panel control housing a grouped set of Radio buttons that are not multi-select by nature, so given the nature of the application, this may be a more suited style of interface.

When selecting a radio button this has to be done manually with a mouse click, the reason for this is to make it even more restrictive, the form stops a user selecting a button, pressing OK then tabbing or scrolling down and pressing OK again (it basically stops quick execution of code and forces a user to click the selected radio button, click OK, then click the next radio button and press OK – unlike the Tab System that allows you to Ctrl+Tab through the tabs and use the Up/Down keys to highlight a checkbox and press Space to select the check box.)

There's no Icon to prettify the interface here, we can however add a short description just above the top Radio button and we can customise the colour of the 1-pixel border that surrounds the entire child controls.
We can do so by modifying the following variables:

```
$AppConfig_Radio_GUI_Panel_Border_Colour = [System.Drawing.Color]::RoyalBlue
$AppConfig_Radio_GUI_Group_Box_Main_Text = "Select your Remote Machine"
```



We now have a RoyalBlue border around the controls and a RoyalBlue description for both GroupBoxes. There's not much to customise here as it's meant to be a more restrictive interface but there's enough to give it all meaning.

Each Radio Button has its own function, and just like our checkboxes they write output to the console so that you can locate them easier.

If we switch back to the Tab System layout by changing `$AppConfig_Radio_GUI_Enforced` from `$True` to `$False` we can take a look at some of the last remaining variables.

Misc Variables

Here we'll list some variables that we have not yet configured and explain what they do.

```
[Boolean] $AppConfig_Hook_Button_OK_To_Return_Key
```

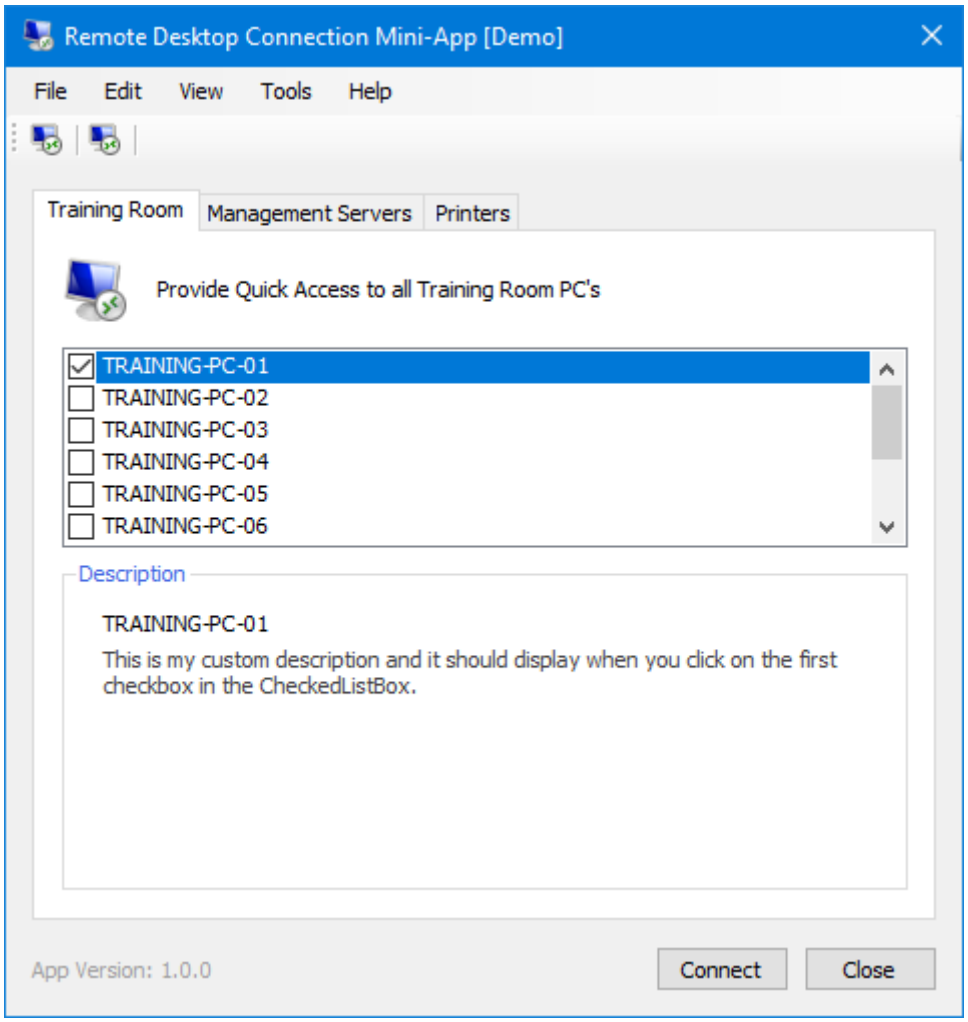
This variable controls whether or not you want to implicitly hook the **Return** key to the OK button on both the main form and the About form. Pressing Return will then trigger, or, invoke the button control as if you had clicked it. Ideal if the application you are designing is for you and you want to quickly execute your code.

```
[Boolean] $AppConfig_Hook_Button_Cancel_To_Escape_Key
```

This variable controls whether or not you want to hook the **Escape** key to the Cancel button, and, the OK button on the About form. Pressing Escape would trigger the pressing of those buttons / closing the form(s)

```
[Int]     $AppConfig_Extended_Width
```

This variable controls the extended width of the application interface. The default value is 0. When you want to extend the width of the application slightly so that it cleanly fits your long string variable values, you can extend it by system points. If we change the value of 0 to 100 this is the result:



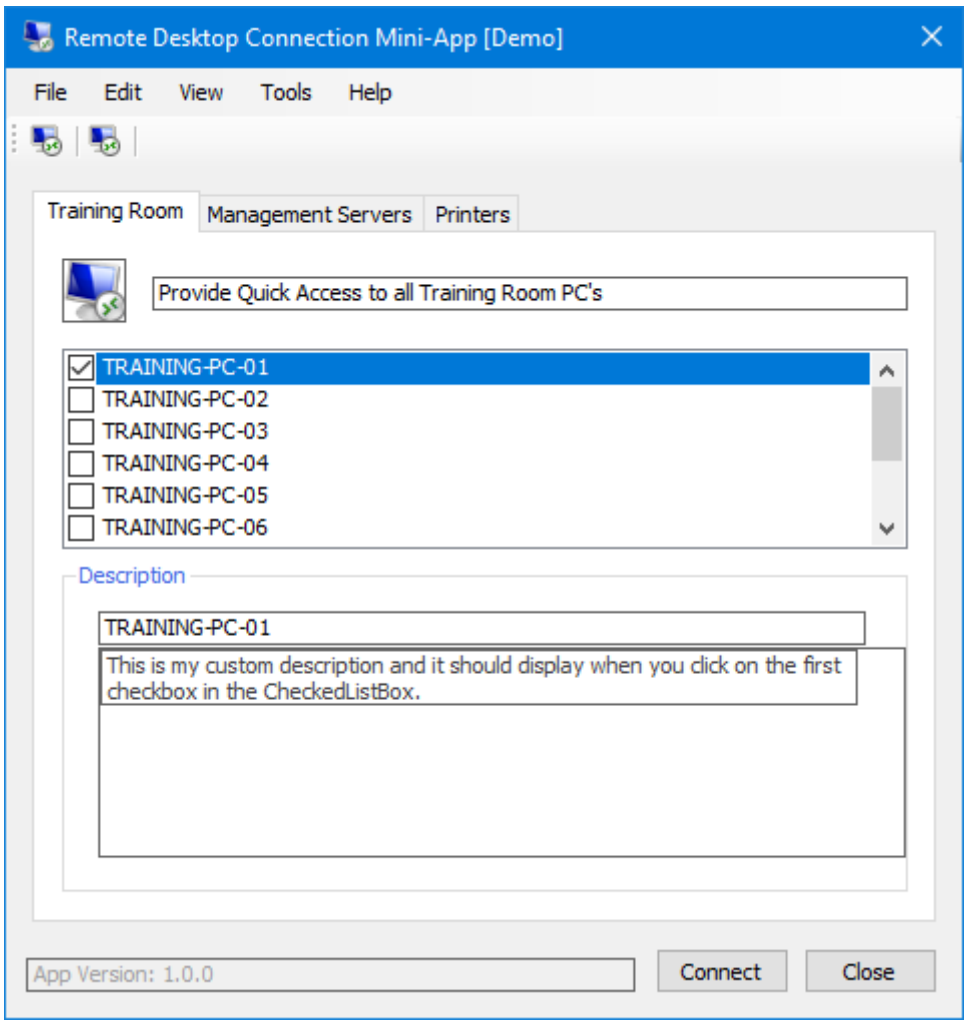
Such modification could allow you to add longer FileMenu names, longer Tab names, Tab descriptions, Checkbox string names, descriptions, and even longer Custom Label text. You'll want to play around with this integer increasing the number slightly until you find that sweet spot.

```
[Object]  $AppConfig_Font
```

This variable controls the Font for the entire application. I have tested with a lot of system fonts and found the current configuration to work nicely on various systems.

```
[Boolean] $AppConfig_Show_Borders
```

I had created this variable for what I call 'Design Mode'. Basically, when the value is set to `True` it will display a solid border around all controls that don't currently have a border, this allows me (and you) to align all controls to suit our requirements. Here's what the application looks like in 'Design mode':



Design mode will show you how wide your text controls are so that you can create your string variables to suit. If you enable design mode and toggle the width of the interface, you'll see which objects remain anchored to the left and which object stretch to match the extended width that you have supplied.

Application Information Area

This area should make more sense here rather than introducing it at the beginning.

At the very top of the template files there's an information area that allows you to add useful information both for yourself and your colleagues, to which I'll populate with some dummy info, as shown below:

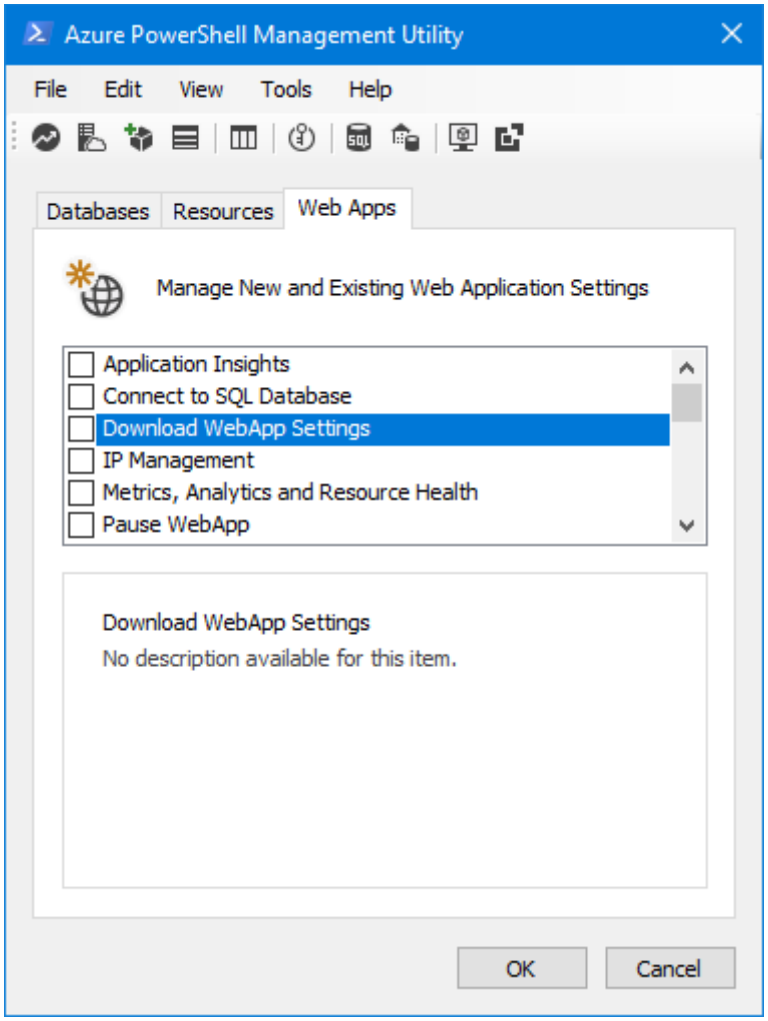
```
<#
                Application Name: My Demo Application
                Template Version: 1.0.0
             Application Version: 1.0.0
                       File Name: Mini-App Demo.ps1
             Required Assemblies: [System.Windows.Forms]
           Visual Styles Enabled: True
         Hash Table Ordered Cast: True (We are not using GetEnumerator())
            Custom Areas Updated: Form_Main_Loading & Form_Main_Shown
                        Synopsis: Demo Application with some functional buttons
                     Description: Lorem ipsum...
                  Known Issue(s): This application does not work on Monday mornings
              Additional Note(s): Lorem ipsum...
                    Reference(s): URL, URL, URL
                 Original Author: You
                  Author Contact: Your Site URL or Tel Number
                     Author Site: https://YourSite.com
              Author Repository: https://YourSite.com/Mini-App-Repo
                      Change Log: V1.00.00 - 00/00/0000 - Initial Release
#>
```

Application Ideas & Examples

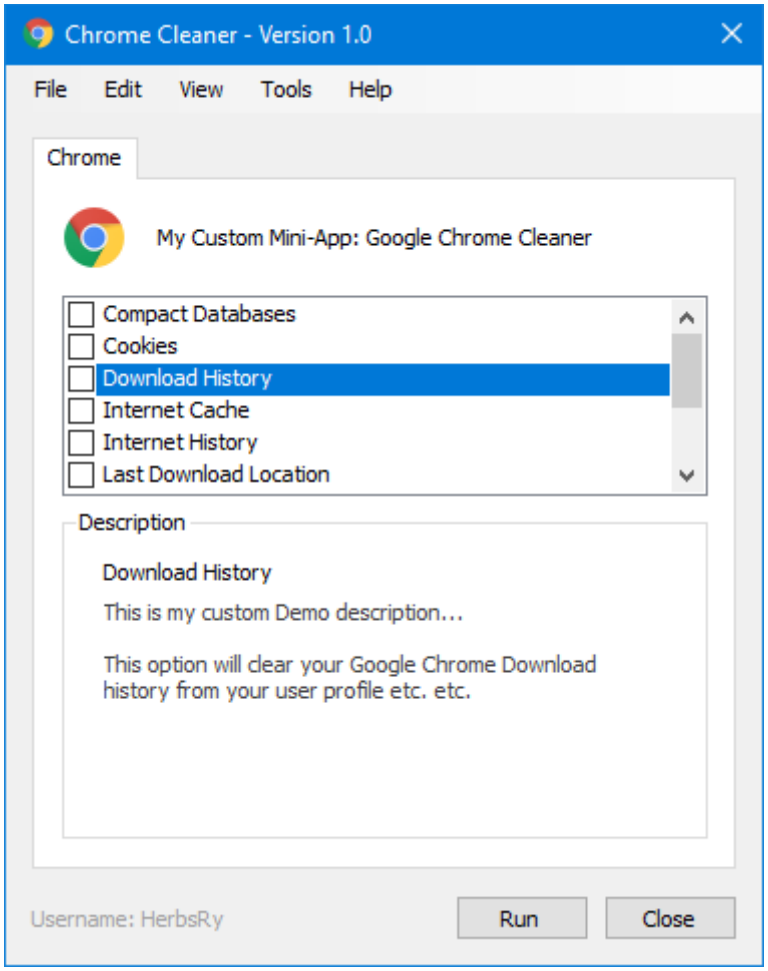
Here's a list of some apps that I have created for demo purposes using this current template.
I hope that it gives you some ideas on what you can create and how things can look, and, get those creative juices flowing. All below images are for my own personal use and were created solely for the purpose of demonstrating what can be achieved from using this template.

For those of you that go ahead and create some cool looking applications and would like them to be available in the main GitHub project repository, feel free to contact me. Likewise, if you have any ideas or suggestions please reach out.

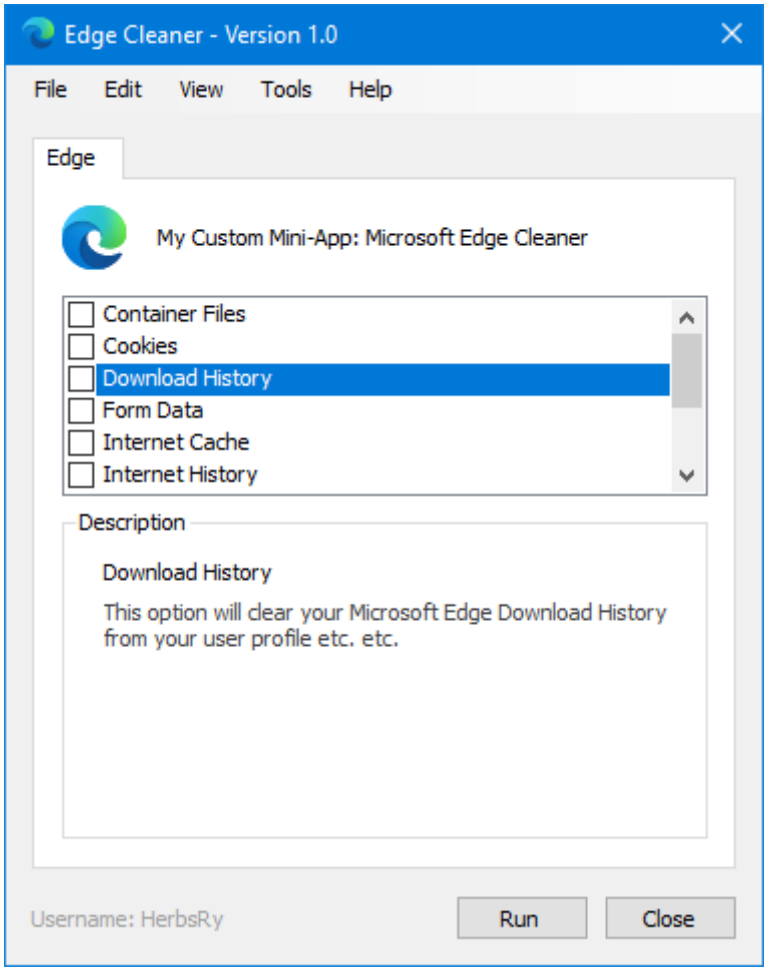
Azure PowerShell Management Utility



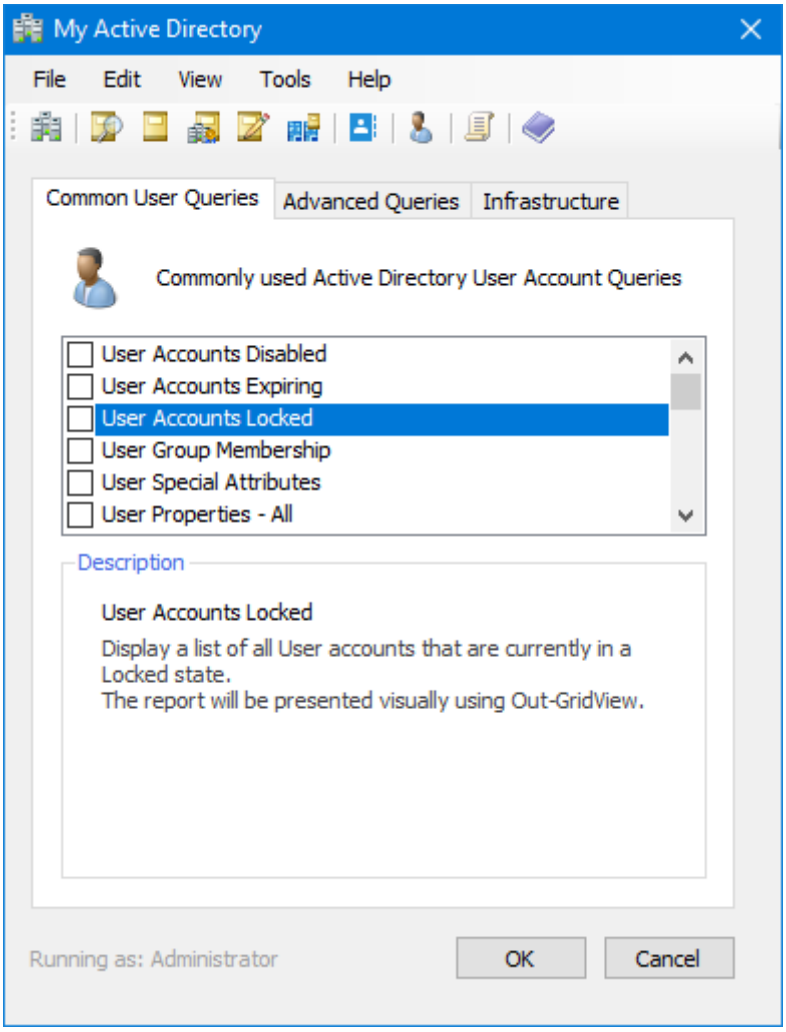
Chrome Cleaner



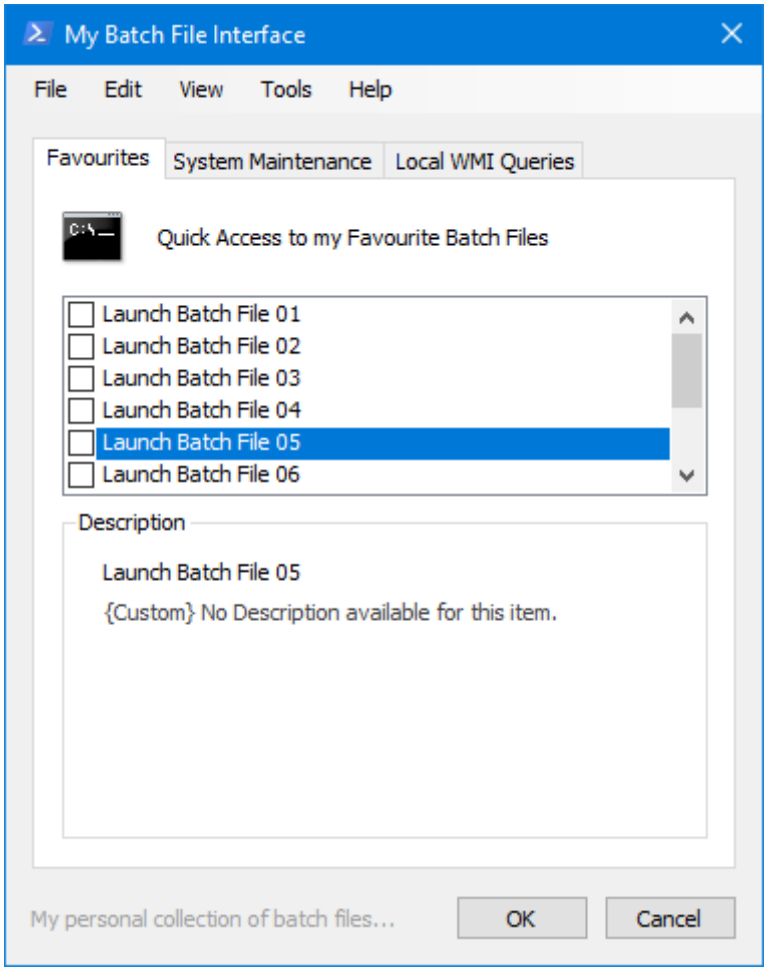
Edge Cleaner



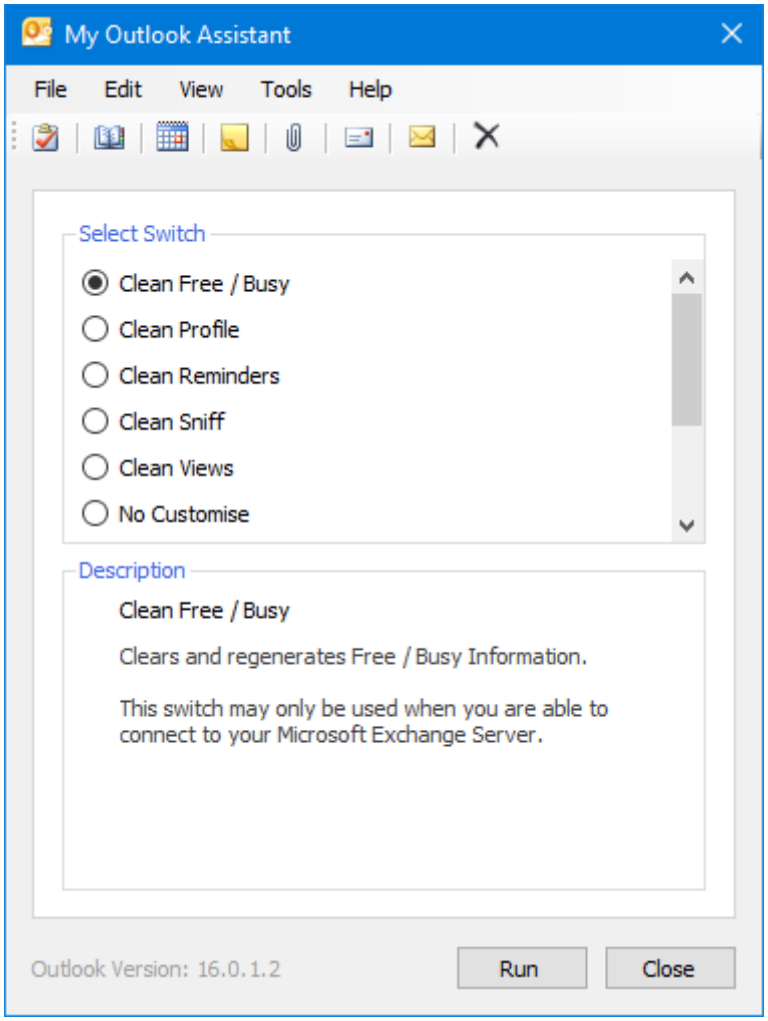
My Active Directory



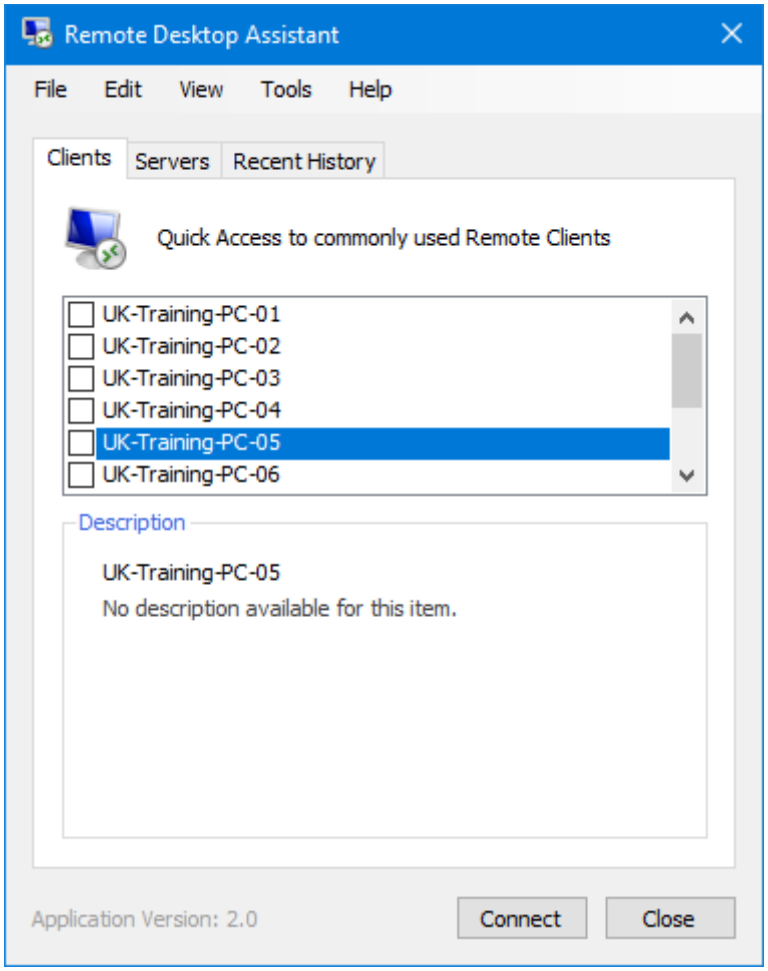
My Batch File Interface



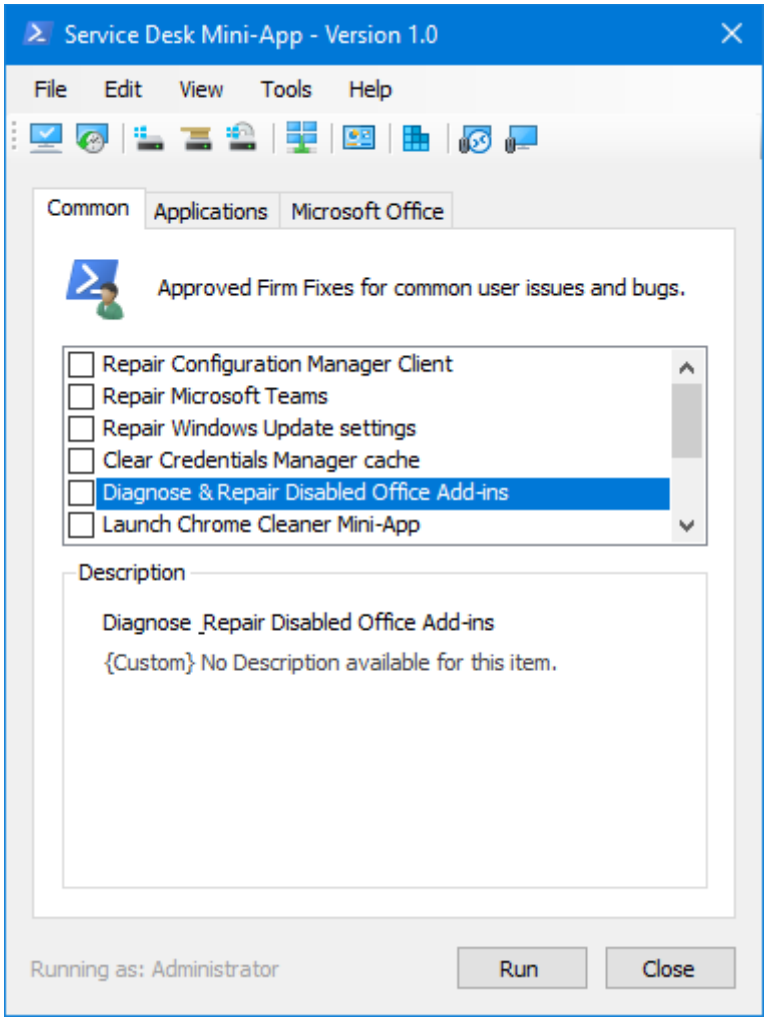
My Outlook Assistant



Remote Desktop Assistant



Service Desk Mini-App



System Maintenance