File Browser

Contents

[Contents 1](#_Toc38607975)

[Analysis 1](#_Toc38607976)

[Problem background 1](#_Toc38607977)

[Planning 2](#_Toc38607978)

[Current System 2](#_Toc38607979)

[Project Outline 4](#_Toc38607980)

[Objectives 5](#_Toc38607981)

[General objectives 5](#_Toc38607982)

[Specific objectives 5](#_Toc38607983)

[Prospective Users and Acceptable Limitations 6](#_Toc38607984)

[System limitations 6](#_Toc38607985)

[User needs 7](#_Toc38607986)

[Potential solution 8](#_Toc38607987)

[Documented Design 9](#_Toc38607988)

[Object-oriented plan 9](#_Toc38607989)

[MVVM 9](#_Toc38607990)

[Class Diagrams 10](#_Toc38607991)

[User interface (UI) Design 11](#_Toc38607992)

[Flow Diagram 12](#_Toc38607993)

[Key Data Structures 12](#_Toc38607994)

[Technical Solution 12](#_Toc38607995)

[C# 13](#_Toc38607996)

[XAML 20](#_Toc38607997)

# Analysis

## Problem background

A file system is essential for a computer built upon the von Neumann model, as it is where both the system files and user-created files are stored. Not having a file system, data stored in a hard drive, or other storage devices, would just be one massive piece of data, not knowing where one file stops and the other starts. Having software that can traverse it allows the user of the computer to store and retrieve data without the need to know console commands, opening the audience to a broader spectrum of possibilities.

The most common method of displaying a traversing a file system is through an abstract data type called a Tree. A tree is a connected, undirected graph with no cycles meaning the most natural way forward is through the children and going back is through the parent of the item, creating the folder-like system we see on every computer today. Most operating systems have a built-in file browser, some prominent ones being Windows file explorer and Apple Macintosh finder. These file browsers are the software that allows the user to interact with the computer's file system in order to access their documents in a simplified, struggle-free way.

After helping some family members with how to use some features in Windows file explorer, It became apparent to me that most people do not use a good portion of the built-in features of the application. When helping people with computer problems, the most common issue that I have found is misplacing a file or accidentally deleting it. For the majority people, the user interface is where most problems go wrong; when asking classmates or family member the common answers where that Windows file explorer is too chaotic, and Macintosh finder did not have the option they wanted with quick access.

## Planning

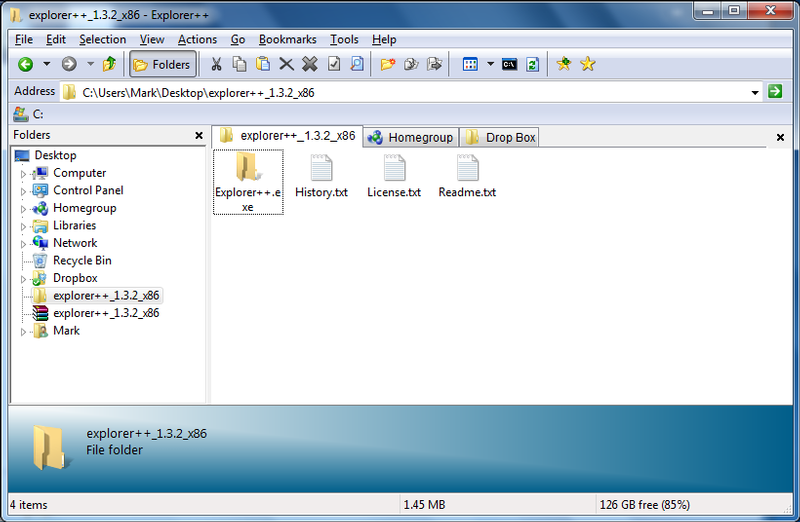
I have been assigned the task of creating a project in response to a contemporary real-world issue of choice. Because I had found problems associated with the current file browsers on today popular operating systems, I have, resultantly, decided on developing my own file browser that attempts to combat the issues associated with these current file browsers. As a means of finding out more about the existing software we can see today, I have assigned myself a task of researching into them further. Hopefully, this will help me grasp a better understanding of the community's perspective and the current responses we have seen in attempts to improve the software.

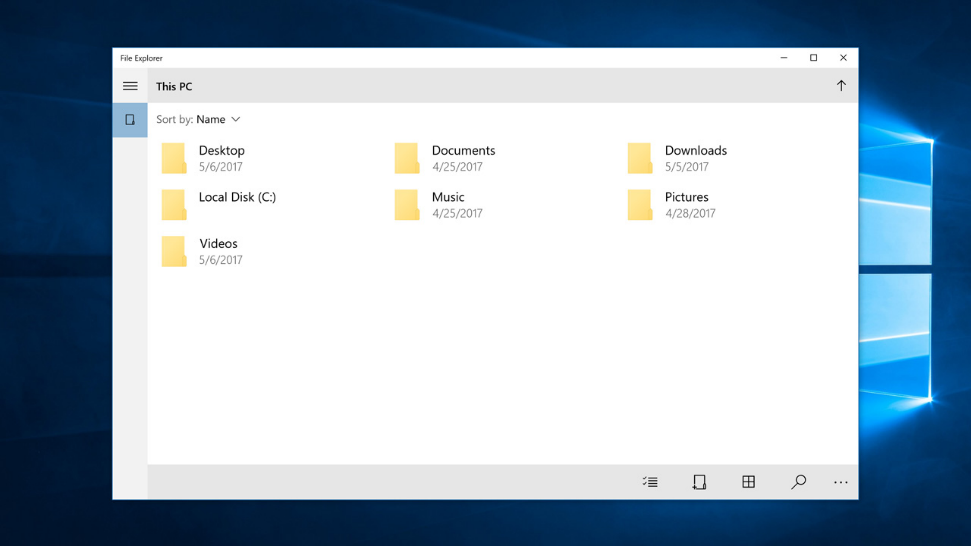
Upon searching the internet, I was able to discover the community's perspective on the current file browser systems through researching through forums and identifying the issues that people commonly face. I was also able to find websites that developed a list of alternative file browsers that proposed new innovative methods in attempts to improve the native software.

Throughout my research, the most common reasons for people changing their choice of file explorer were due to style/ appearance and, for some, adding more features, such as compression or decompression algorithms. These trends were evident in the most popular third-party file browsers from just looking at the User Interface (UI) designs. Most of the file browsers were either open source or free for use, enabling the file browser to grow an audience a lot quicker since people are willing to download and experiment with them.

## Current System

Through this research, I was able to discover a range of software that attempted to perform improvements to the native software and have aimed to utilize them as inspiration for my own project; here are some examples of what I found:

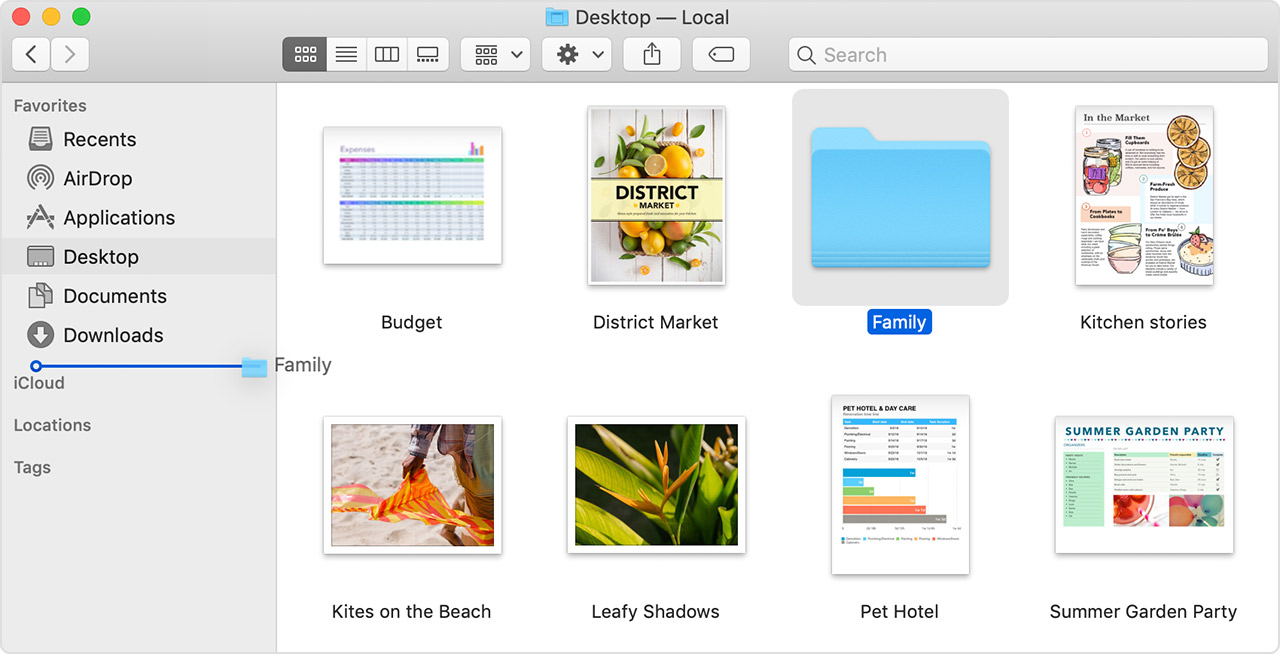
Explorer++ is a file browser that tries to add more on to the native windows file explorer by giving the user more settings and options. Explorer++ has a built-in tab system that allows you open folder up like tabs in an intent browser, this feature enables the user to look and navigate through folders quickly. One of my goals, like explorer++, is to have smooth, quick navigation through folders.



On the opposite side, there are file explorers meant for a visual improvement over functionality, like taking away many settings and give the user a programme that the primary function is to browse. UWP File Explorer is an excellent example of this. Having a minimalistic design allows for the user to see what they are doing within the application and is a feature I want my file browser to have a similar design.

The two main file browsers used are Window's file explorer and Macintosh's Finder. Both being the native file browser to their operating systems and have been designed developed to decades. Both have features that make them stand out. Finder has efficient searching algorithms and windows, taking up 35.2% of the market share compared to apples 7.7% market share. With a gap that big in the market share, Windows file explorer is used by most people and is seen as the default file browser.





## Project Outline

As a result of this research, I have concluded on proposing my project of a file browser. This file browser will attempt to solve the issues associated with poor user interface and traversal of the tree structure. I will address these issues through developing my own piece of software, using WPF to create a more user-friendly interface, also creating settings for the user to filter what is on the Screen. This should, hopefully, solve the issues I aim to address, creating a file browser that is far more universal for all sorts of users since it is a browser that is friendly to use while, simultaneously, remaining sophisticated. I will take inspiration from software by the likes of Macintosh's Finder and UPF file explorer; however, I will make sure it will remain personal to my needs.

## Objectives

Subsequently, from understanding what I want to develop, I have set myself a list of tasks in which I would like to meet by the end of this project. Some of the goals may be ambitious; however, it allows a higher level of room for improvement if the more straightforward objectives are met before the deadline; Here is what I set:

### General objectives

1. To be run on any computer running the Windows operating system.
2. Have a UI User-friendly.
   * The surface to be uncluttered
   * All fonts should be readable
   * Application wide colour scheme
   * Efficient layout of all of the interactable
3. Give different options on how to move files around.
4. The target audience should understand the application, e.g. no spelling or grammar errors
5. The application should run and be efficient without any freezing
6. Navigation should be comfortable and straight forward
7. To offer an alternative to a file browser then the native built on to the operating system.

### Specific objectives

1. The application should show all files, folder and drives that the user has permission to see.
2. Should have a search bar so the user can search through a directory or the whole computer.
3. Can navigate using both the sidebar and main viewport.
4. Have a forward and back button for navigation.
5. Use a zipper-like structure to hold all of the navigation histories and allow the user to move around using the zipper structure.
6. Have a setting to turn on and off the zipper navigation and other settings.
7. Style the Window to have a material theme as it is more of modern design.

## Prospective Users and Acceptable Limitations

This programme has the potential to affect a wide variety of different people who utilize the computer and the file system within their everyday life. It can change the user productivity as it will allow them to be able to focus on more critical tasks, rather than having disputes with their file system while, at the same time, allowing them to improve their workflow through the integrated features.

Aside from people who work heavily on computers, the everyday user may also be able to benefit from this. The user-friendly helps with a better understanding of what is happening on Screen; as well as accessing and traverse their files. People that are less confident with technology will be able to understand the interface and focus on more critical tasks, possibly even utilizing the advantages of all the different integrated features, ultimately, improving both the efficiency and ease of use within the computer's file system.

I'm planning on using certain people as example users for my project in order to ensure that the software has met the requirements for the everyday user. I plan to use my supervisor, Matthew, as an example user for this project due to the fact he is someone who uses a computer very often and can provide me with feedback on both my code and project functionality. I also, intend to use my mother as an example user as she is an accountant that often has to work with files and has even had issues with window file explorer recently where she almost lost all of her work due to the poorly designed system. This should, hopefully, give me two entirely different perspectives that can provide me with useful feedback upon many aspects of my project. I will also take myself as a prospective user as I tend to work a lot with files and will able to determine whether it is feasible development or not.

### System limitations

As this is a project that is high level and contains a significant amount of content, the project will have limitations to it. The biggest one being the finishing touches that will mean that the code will not be industry-standard level due to the fact it is only one developer (me) under a small-time frame to produce it in. Having a fully bug free application would take far longer than the time given so; as a result, having minor bugs will be inevitable.

As previously stated, the app will not be industry-standard, keeping standard features from both Finder and explorer, such as moving, deleting or creating files or folders, that are basic necessities of a file browser. Adding a setting for people who have protanomaly, deuteranomaly and tritanomaly colour blindness or having different themes the user can choose might not be achievable due to the time limit. Making complex algorithms for compression and decompression would also not be a viable option due to the time limit of the project.

## User needs

In order to get a better understanding of what the user needs are, I have developed a questionnaire to which I have distributed to a range of different people who may be associated with files in various ways. This will, hopefully, allow me to understand what features will be essential to emphasize while, at the same time, allowing me to gain new possible ideas; Here is what I have asked:

1. What operating system do you use?
2. Do you use the default file browser?
3. Do you enjoy the functionality of this file browser?
   1. If so, what features do you enjoy?
   2. What features do you dislike?
4. What feature would you like to see in a file browser?
5. What makes a browser UI friendly?
6. Are you intrigued by a system that enables you to search through the folder easier?
7. Are you intrigued by a system that allows you to hide files?
8. Is there anything you'd like to add?

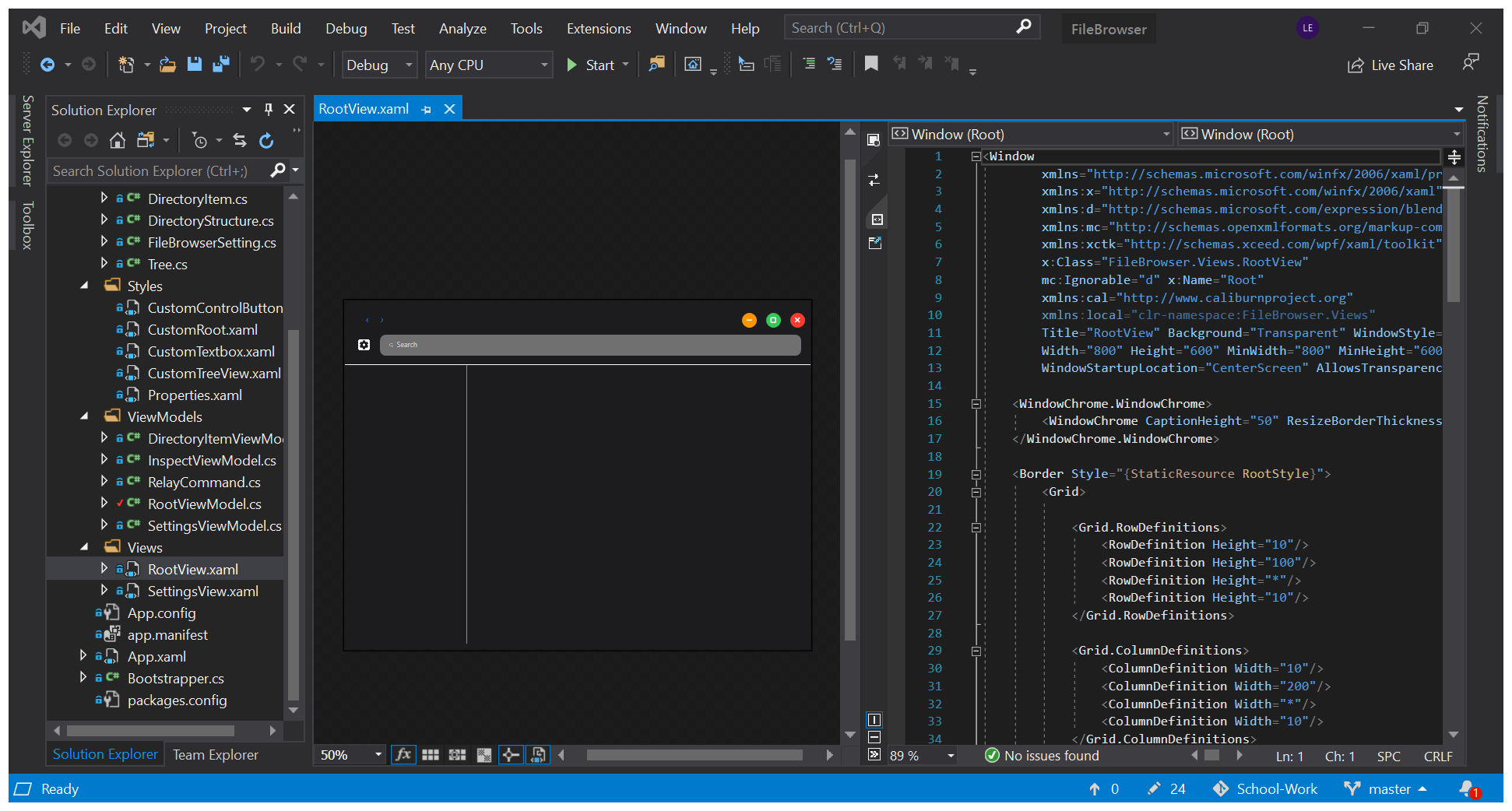
These responses have provided me with a significant level of information, letting me understand what proposed features are likely to be successful. The questionnaires also provided me with many new ideas that can be implemented into my software. As a means of ensuring my target audience is satisfied with my development proposals, I have distributed a second questionnaire that proposes all the features I intend to implement into my software, letting them determine which features they would like to see developed.

Most recipients of my questionnaire used windows operating system and used the file explorer as their default file browser, but a few did use Macintosh with Finder as there default file browser. There was a resounding agreement that windows file explorer was easy to use if you know where your files are and that the searching in the application is slow and sub-par.

When creating a user interface, the responses from my questionnaire all had common features some being; ease of use, good colour scheme and clear being a non-messy interface with clear text and pictures. (see appendix c for the responses).

## Potential solution

Visual Studio was designed to design and build many programmes, having templates from many different types of projects. I will be using Visual Studio for my project as I'm familiar with the editor and will be able to create an application with it.



For my potential solution for this project, I plan to use a WPF application because it is a programme, I am familiar with, as well as the fact that it allows access for a user interface. The WPF application is designed for creating user interfaces through the XAML mark-up language. This will prove very useful for a file browser since an interface is, arguably, one of the essential aspects to a file browser. I may also utilize C++ in order to have better control over the file system as C# becomes too high level of a language at times. I am using a package called Caliburn. Micro, allowing me to create a mode, view, View Model (MVVM) structure that enables a dynamic update to the user interface base on the variables.

# Documented Design

The application I am designing is a file browser that runs on Windows operating system. The user will able to view and traverse through their file system.

## Object-oriented plan

As I will be using object-oriented programming throughout the application, I have decided to base my architecture off the MVVM model and utilizing a package called Caliburn.micro to help classes Inherit the right C# interfaces.

### MVVM

#### Model

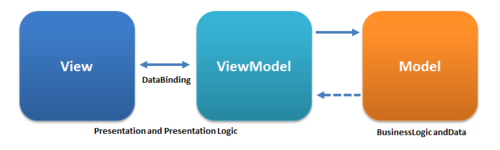
Models are, in essence, regular C# classes libraries meant only for creating objects and holding data.

#### ViewModel

View models are C# class that Inherit from the Screen class from Caliburn.micro package and then connected to the corresponding view. Caliburn.micro connects the by names of the files, e.g. as RootViewmodel.cs is attached to the RootView.xmal. The attachment can be made as all windows made in XAML have a property called DataContex that inherited from the Window class from System.windows namespace. Setting a C# class as DataContext allows any fields or attributes to be bound to the XAML window through built-in data binding in WPF applications.

#### Views

Views are WPF windows or user controls. They are programmed using a mark-up language called XAML and are how the user interface is created.



## Class Diagrams

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

## User interface (UI) Design

The reason I went for a native dark theme instead of a light theme as it's a recent popular design as it helps with not only the battery life if using a laptop but when using in low lights the user won't be blinded by bright colours or lights. I took inspiration from Finder as Apple is known for having aesthetically pleasing designs. I decided to keep with the sidebar on the right as it is a common feature in all file browser I have seen. This will also help with users knowing how to use it straight off, as there is no learning curve.

A screenshot of a computer screen

Description automatically generated

A picture containing drawing

Description automatically generated

Instead of having toggles on the top bar of the file browser, I went with a different window that will toggle settings on and off. A choice I made to keep the UI uncluttered as possible.

## Flow Diagram

A close up of a logo

Description automatically generated

## Key Data Structures

In my programme, there will be various structure used to store data. Ill will need a structure to hold the current directory with all of the files and folders held in children variable. Other structures will be a tree-like structure for breadth-first and depth-first searches; the tree will need to keep node structures that will contain the data of each directory.

# Technical Solution

Below is shown the entry point to my programme; as this is a WPF application, the application recourse is also located here.

1. <Application **x:Class**="FileBrowser.App"
2. **xmlns**="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3. **xmlns:x**="http://schemas.microsoft.com/winfx/2006/xaml"
4. **xmlns:local**="clr-namespace:FileBrowser">
5. <Application.Resources>
6. <ResourceDictionary>
7. <ResourceDictionary.MergedDictionaries>
8. <ResourceDictionary **Source**="/Styles/Properties.xaml"/>
9. <ResourceDictionary **Source**="/Styles/CustomTextbox.xaml"/>
10. <ResourceDictionary **Source**="/Styles/CustomRoot.xaml"/>
11. <ResourceDictionary **Source**="/Styles/CustomControlButtons.xaml"/>
12. <ResourceDictionary **Source**="/Styles/CustomTreeView.xaml"/>
13. <ResourceDictionary>
14. <local:Bootstrapper **x:Key**="Bootstrapper"/>
15. </ResourceDictionary>
16. </ResourceDictionary.MergedDictionaries>
17. </ResourceDictionary>
19. </Application.Resources>
20. </Application>

  On line 14, you can see where the application initializes the. The bootstrapper is then called.

1. public class Bootstrapper : BootstrapperBase
2. {
3. public Bootstrapper()
4. {
5. Initialize();
6. }
8. protected override void OnStartup(object sender, StartupEventArgs e)
9. {
10. DisplayRootViewFor<RootViewModel>();
11. }
12. }

## C#

The bootstrapper inherits from a class in the Caliburn.micro package called BootStrapperbase this, allows me to then override a method from that class to display my root view by passing my root view model as a type to the DisplayRootViewFor method. The root view and root view model is where the main code runs.

All the following code is from the root view model class. The rootviewmodel class inherits from a class from Caliburn.micro called Screen Which helps connect the model view to the view.

The root view model class has eight attributes to it each having a different role on how data is displayed on the screen. Drives variable is used for the sidebar of the application, holding all the most used directories and the logical drives of the computer. viewport is a view model bound to an Item control in the rootview.xaml this is variable is key for showing the user all the folders and files in that directory. The backstack and forwardstack are used for the forward and back button. settingview is a XAML window for when the user wants to see the settings. settingviewmodel is the class that will be the data context for the settingview allowing the windows toggle buttons to access the settings for the application.

1. private string \_searchValue, \_name;
2. private ObservableCollection<DirectoryItemViewModel> \_drives;
3. private InspectViewModel \_viewport;
4. private Stack<InspectViewModel> \_backstack;
5. private Stack<InspectViewModel> \_forwardstack;
6. private SettingsView \_settingview;
7. private SettingsViewModel \_settingviewmodel;

1. public ObservableCollection<InspectViewModel> ViewPortChildren
2. {
3. get
4. {
5. ObservableCollection<InspectViewModel> temp = ViewPort.Children;
6. if (FileBrowserSetting.FilterNotSearch && searchValue != null)
7. {
8. temp = new ObservableCollection<InspectViewModel>(temp.Where(x => x.FullPath.Contains(searchValue)));
9. }
10. if (FileBrowserSetting.OnlyShowFilesVisted && searchValue != null)
11. {
12. temp = new ObservableCollection<InspectViewModel>(temp.Where(x => x.Visted == true));
13. }
14. if (FileBrowserSetting.OnlyShowFilesHaventVisted && searchValue != null)
15. {
16. temp = new ObservableCollection<InspectViewModel>(temp.Where(x => x.Visted == false));
17. }
18. return temp;
19. }
20. }

This public attribute is the key to show all the files and folders to the user it gets the all of the children of the current directory by accessing the viewport variable that holds all the current directory. If then has a set of if statement that only triggers if a setting is turned to true. Each if statement will, if triggered, apply a filter to the Observable collection, this why more then one filter can be applied.

1. public string searchValue
2. {
3. get { return \_searchValue; }
4. set
5. {
6. \_searchValue = value;
7. NotifyOfPropertyChange(() => searchValue);
8. }
9. }

 All public properties follow the same structure by setting the private variable and then notifying the application that the variable has changed; so, the user interface can update a display the right information.

1. this.Drives = new ObservableCollection<DirectoryItemViewModel>(DirectoryStructure.GetLogicalDrives().Select(e => new DirectoryItemViewModel(e.FullPath, e.Type, e.Name, e.Hidden)));

 in the constructor I used an advance list operation in which in functional paradigm sense is a map function. As select (the map function) Is a higher order function it allows me to pass another function in. in this example I map all items from the method GetLogicalDrives, which returns DirectoryItems a custom struct I made, and mapped them to DirectoryItemViewModel.

1. public void Exit(Window MainWindow)
2. {
3. this.\_settingview.Close();
4. MainWindow.Close();
5. }
7. public void Minimise(Window MainWindow)
8. {
9. MainWindow.WindowState = WindowState.Minimized;
10. }
12. public void Maximise(Window MainWindow)
13. {
14. if(MainWindow.WindowState != WindowState.Maximized)
15. {
16. MainWindow.WindowState = WindowState.Maximized;
17. }
18. else
19. {
20. MainWindow.WindowState = WindowState.Normal;
21. }
22. }

 These three methods are bound to buttons in the rootview.xaml and bound to their click events allowing me to make custom buttons for the window.

1. public void OpenChild(InspectViewModel child, MouseButtonEventArgs e)
2. {
3. if (e.ClickCount >= 2)
4. {
5. if (child.Type != DirectoryType.File)
6. {
7. this.\_backstack.Push(this.ViewPort);
8. if(this.\_forwardstack.Count > 0)
9. {
10. this.\_forwardstack.Clear();
11. }
12. for(int i = 0; i < ViewPort.Children.Count; i++)
13. {
14. if(ViewPort.Children[i].FullPath == child.FullPath)
15. {
16. ViewPort.Children[i].Visted = true;
17. }
18. }
19. this.ViewPort = child;
20. ViewPort.Visted = true;
21. RefreshViewPort();
23. }
24. else
25. {
26. DirectoryStructure.OpenFileInProgramme(child.FullPath);
27. }
28. }
29. }

 This subroutine Is bound to the click event on the file or folder. Allowing to either open the file or load new directory depending on the type of the directory item clicked. We also take in mouse button event arguments to make sure the user doubled clicked (line 5) on the item not just single clicked, as a signal click could be a mistake. On line 8 you can see a check for on the forward stack this is because if you are going forward, then the forward stack should be cleared as you are taken a new path forward. From line 12 to 18, we are doing another check to make sure the child visited attribute is set to true, as a setting need the information when filtering the children when hiding visited or non-visited folders

1. public void BackButton()
2. {
3. if(this.\_backstack.Count > 0)
4. {
5. this.\_forwardstack.Push(this.\_viewport);
6. this.ViewPort = this.\_backstack.Pop();
7. this.ViewPort = this.ViewPort;
8. NotifyOfPropertyChange(() => ViewPortChildren);
9. }
10. }
12. public void ForwardButton()
13. {
14. if(this.\_forwardstack.Count > 0)
15. {
16. this.\_backstack.Push(this.ViewPort);
17. this.ViewPort = this.\_forwardstack.Pop();
18. NotifyOfPropertyChange(() => ViewPortChildren);
19. }
20. }

 These two subroutines are bound to the forward and back button in the rootview.xaml allowing the user to traverse back and forth through folders.

1. public async void Search(KeyEventArgs e)
2. {
3. if(e.Key == Key.Enter || e.Key == Key.Return)
4. {
5. if (!FileBrowserSetting.FilterNotSearch)
6. {
7. await TreeSearch();
8. }
9. else
10. {
11. NotifyChlidren();
12. }
13. }
14. }

This subroutine is how the user search in the file browser, there are two different ways of searching. The first one being the normal tree search using either breadth-first or depth-first searches or if the setting for filter instead of search is it will apply a filter on the folders in the current directory on whether they contain the search term. Making this subroutine asynchronous allowed the window not to freeze any time the user searched.

1. private async Task TreeSearch()
2. {
3. Regex regex = new Regex("\\.[a-z | A-Z]+");
4. Tree search = new Tree(this.\_viewport);
5. List<TreeNode> results = new List<TreeNode>();
6. if (regex.IsMatch(searchValue))
7. {
8. results = await search.DepthSearch(search.root, results, searchValue);
9. }
10. else
11. {
12. results = await Task.Run(() => search.BredthSearch(searchValue));
13. }
14. if (results.Count > 0)
15. {
16. this.ViewPort = new InspectViewModel($"Search: {searchValue}, {results.Count} Items", new ObservableCollection<InspectViewModel>(results.Select(x => new InspectViewModel(x.FullPath, x.Type, x.Name, x.Hidden))));
17. RefreshViewPort();
18. }
19. else
20. {
21. Name = "No results found";
22. }
23. }

For the tree search I first setup a tree with root being the current directory the user is in, then seeing if the search term matches the predefined regular expression. The regular expression looks for and extension e.g. txt or .cs if it is a match, I decided to use a depth first search as all files will be leaves on the tree as they have no children. The depth first search starts at the root and heads down a path till it gets to a leaf, meaning the search is getting to files quicker then breadth first. Both tree searches are asynchronous to reduce the load on the UI thread as much as possible.

The InspectViewModel also like the rootviewmodel inherits from screen. This class contains 7 attributes relating to the current directory as this is the class, we hold the data in.

1. private ObservableCollection<InspectViewModel> \_children;
2. private string \_fullpath, \_name;
3. private bool \_hidden;
4. private BitmapImage \_image;
5. private DirectoryType \_type;
6. private bool \_visted;
7. public BitmapImage Image
8. {
9. get { return this.\_image; }
10. set
11. {
12. this.\_image = value;
13. NotifyOfPropertyChange(() => Image);
14. }
15. }

 The public variable work the same way as in the rootviewmodel.

1. if (FileBrowserSetting.ShowHiddenFiles)
2. {
3. Hidden = false;
4. }
5. else
6. {
7. Hidden = hidden;
8. }

 Within the constructor for the InspectViewModel this if statement helps with the setting show hidden files, as it makes all file shown.

1. public void GetChildren()
2. {
3. if (FullPath != null && Type != DirectoryType.File)
4. {
5. List<DirectoryItem> item = DirectoryStructure.GetDirectoryItems(this.FullPath);
6. Children = new ObservableCollection<InspectViewModel>(item.Select(x => new InspectViewModel(x.FullPath, x.Type, x.Name, x.Hidden)));
7. }
8. }

 The GetChldren Subroutine uses the same map-like function to cast the DirectoryItem to InspectViewModel.

1. public static implicit operator InspectViewModel(DirectoryItemViewModel e) => new InspectViewModel(e.FullPath, e.Type, e.Name, e.Hidden);

This custom operator allows me to put any DirectoryItemViewModel in a structure that takes InspectViewModel which is useful for when the user clicks on the side bar and want to go to that directory.

The DirectoryItemViewModel is the view model I bind to the tree view on the sidebar they are like the InspectVeiwModel, but they only hold folders no files. As well as Icommand that I bind to the opening trigger on the tree to expand the item and get the children.

1. private ObservableCollection<DirectoryItemViewModel> \_children;
2. private string \_fullpath;
3. private DirectoryType \_type;
4. private string \_name;
5. private BitmapImage \_image;
6. private bool \_hidden;
7. public string FullPath
8. {
9. get { return \_fullpath; }
10. set
11. {
12. this.\_fullpath = value;
13. NotifyOfPropertyChange(() => FullPath);
14. }

  Public variables work the same as the other view models.

1. public DirectoryItemViewModel(string fullpath, DirectoryType type, string Name, bool hidden)
2. {
3. this.ExpandCommand = new RelayCommand(Expand);
4. this.Hidden = hidden;
5. this.FullPath = fullpath;
6. this.\_type = type;
7. this.Name = Name;
8. this.ClearChildren();
9. this.Image = new BitmapImage(new Uri(DirectoryStructure.GetPicture(this.Type)));
10. }

 A new Relay command is assigned to the Expand command with the subroutine Expand from the class passed in as and argument. The RelayCommand is a class that take in an action and can execute that action when the Execute method in the RelayCommand is called.

1. private void Expand()
2. {
3. if(this.Type == DirectoryType.File)
4. {
5. return;
6. }
7. List<DirectoryItem> items = DirectoryStructure.GetDirectoryFolders(this.FullPath);
8. Children = new ObservableCollection<DirectoryItemViewModel>(items.Select(x => new DirectoryItemViewModel(x.FullPath, x.Type, x.Name, x.Hidden)));
9. }

 Within the Expand subroutine on line eight another mapping function is being used.

1. class RelayCommand : ICommand
2. {
3. private Action mAction;

6. public event EventHandler CanExecuteChanged = (sender, e) => { };
8. public RelayCommand(Action action)
9. {
10. mAction = action;
11. }
13. public bool CanExecute(object parameter)
14. {
15. return true;
16. }
18. public void Execute(object parameter)
19. {
20. mAction();
21. }
22. }

 The Relay command inherits from the Command interface, and all subroutine are from that interface. The two subroutines I need are the Execute and CanExecute; the CanExecute is set to true so we can also ways Execute out Execute method. The Execute method is where we execute the action (subroutine) we passed into the class.

The tree model has one attribute, which is the root node of the tree. It also contains two methods breadth-first and depth-first searches.

1. public async Task<List<TreeNode>> DepthSearch(TreeNode CurrentNode, List<TreeNode> results, string value)
2. {
3. if (CurrentNode.Type != DirectoryType.File)
4. {
5. CurrentNode.GetChildren();
6. foreach (TreeNode child in CurrentNode.Children)
7. {
8. if (!results.Contains(child) && child.FullPath.Contains(value))
9. {
10. results.Add(child);
11. }
12. await Task.Run(() => DepthSearch(child, results, value));
13. }
14. }
15. return results;
16. }

 This Depth first search includes recursion as it calls its sell for all children of the node. I go the pseudo code from AQA Computer science textbook under Graph traversal algorithms (page 243). Making the subroutine asynchronous helps with reduce the work the UI thread has to do.

1. public List<TreeNode> BredthSearch(string value)
2. {
3. Queue<TreeNode> ToVist = new Queue<TreeNode>();
4. List<TreeNode> results = new List<TreeNode>();
5. ToVist.Enqueue(this.root);
6. while (ToVist.Count > 0)
7. {
8. TreeNode visiting = ToVist.Dequeue();
9. visiting.GetChildren();
10. if (visiting.FullPath.Contains(value))
11. {
12. results.Add(visiting);
13. }
14. if (visiting.Type != DirectoryType.File && visiting.HasChildren())
15. {
16. foreach(TreeNode child in visiting.Children)
17. {
18. if (!child.Hidden)
19. {
20. ToVist.Enqueue(child);
21. }
22. }
23. }
24. }
25. ToVist.Clear();
26. return results;
27. }

The Breadth first search is also from the AQA computer Science text box under Graph traversal algorithms (page 246). I was not able to make this subroutine asynchronous but when I call it in the RootViewModel class I run it through a Task.Run function offloading it to another thread.

The TreeNode class is very simple holding basic data for directories and having two methods hasChildern and GetChildren. The GetChildren subroutine has the same mapping function seen throughout the application

1. public void GetChildren()
2. {
3. if (this.Type != DirectoryType.File)
4. {
5. List<DirectoryItem> Items = DirectoryStructure.GetDi rectoryItems(this.FullPath);
6. this.Children = Items.Select(x => new TreeNode(x.FullPath, x.Type, x.Name, x.Hidden)).ToList();
7. }
8. }
10. public bool HasChildren() => Children.Count > 0;

The FileBrowserSetting class is a static class allowing any of the public variable in the class can be accessed by all the application at once, acting like global variables

1. public static class FileBrowserSetting
2. {
3. public static bool ShowHiddenFiles { get; set; } = false;
4. public static bool OnlyShowFilesVisted { get; set; } = false;
5. public static bool FilterNotSearch { get; set; } = false;
6. public static bool OnlyShowFilesHaventVisted { get; set; } = false;
7. }

 The DirectoryItem struct holds directory information and is the base form of the data in the application. This is because they are cast to a different type e.g. DirectoryItemViewModel

1. public struct DirectoryItem
2. {
3. public DirectoryType Type { set; get; }
4. public string FullPath { set; get; }
5. public bool Hidden { set; get; }
6. public string Name { get { return this.Type == DirectoryType.Drive ? $"(Local Disk): {this.FullPath}" : DirectoryStructure.GetFileOrFolderName(this.FullPath); } }
7. }

 The DirectoryType Enum allows to specify between folder, files and certain folders. This I usefully when searching of or placing the data in a tree structure to determine if the nod will have children.

1. public enum DirectoryType
2. {
3. File,
4. Folder,
5. Drive,
6. MyDocuments,
7. MyDownloads,
8. MyPhotos,
9. MyVideos,
10. MyMusic,
11. Desktop,
12. NUll
13. }

 The Algorithms and DirectoryStructure class hold Subroutine like get files and folders from a directory and Quick sort algorithm.

1. public static List<DirectoryItem> GetDirectoryItems(string fullPath)
2. {
3. List<DirectoryItem> items = new List<DirectoryItem>();
4. try
5. {
6. string[] folders = Directory.GetDirectories(fullPath);
7. if (folders.Length > 0)
8. {
9. foreach (string folder in folders)
10. {
11. if (IsHidden(new DirectoryInfo(folder)))
12. {
13. items.Add(new DirectoryItem() { FullPath = folder, Type = DirectoryType.Folder, Hidden = true });
14. }
15. else
16. {
17. items.Add(new DirectoryItem() { FullPath = folder, Type = DirectoryType.Folder, Hidden = false });
18. }
19. }
20. }
21. }
22. catch
23. {
24. return new List<DirectoryItem>();
25. }
26. try
27. {
28. string[] files = Directory.GetFiles(fullPath);
29. if (files.Length > 0)
30. {
31. foreach (string file in files)
32. {
33. if (IsHidden(new FileInfo(file)))
34. {
35. items.Add(new DirectoryItem() { FullPath = file, Type = DirectoryType.File, Hidden = true });
36. }
37. else
38. {
39. items.Add(new DirectoryItem() { FullPath = file, Type = DirectoryType.File, Hidden = false });
40. }
41. }
42. }
43. }
44. catch
45. {
46. return new List<DirectoryItem>();
47. }
48. return items;
49. }
50. public static List<DirectoryItem> quickSort(List<DirectoryItem> array, int low, int high)
51. {
52. if (low < high)
53. {
54. int pi = partition(array, low, high);
55. quickSort(array, low, pi - 1);
56. quickSort(array, pi + 1, high);
57. }
58. return array;
59. }
60. public static int partition(List<DirectoryItem> array, int low, int high)
61. {
62. string piviot = array[high].FullPath;
63. int i = (low - 1);
64. for (int j = low; j < high; j++)
65. {
66. if (string.Compare(array[j].FullPath, piviot) < 0)
67. {
68. i++;
69. DirectoryItem temp = array[i];
70. array[i] = array[j];
71. array[j] = temp;
72. }
73. }
75. DirectoryItem temp2 = array[i + 1];
76. array[i + 1] = array[high];
77. array[high] = temp2;
78. return i + 1;
79. }

## XAML

The XAML file are how the user interface is made

1. <Window
2. **xmlns**="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3. **xmlns:x**="http://schemas.microsoft.com/winfx/2006/xaml"
4. **xmlns:d**="http://schemas.microsoft.com/expression/blend/2008"
5. **xmlns:mc**="http://schemas.openxmlformats.org/markup-compatibility/2006"
6. **xmlns:xctk**="http://schemas.xceed.com/wpf/xaml/toolkit"
7. **x:Class**="FileBrowser.Views.RootView"
8. **mc:Ignorable**="d" **x:Name**="Root"
9. **xmlns:cal**="http://www.caliburnproject.org"
10. **xmlns:local**="clr-namespace:FileBrowser.Views"
11. **Title**="RootView" **Background**="Transparent" **WindowStyle**="None"
12. **Width**="800" **Height**="600" **MinWidth**="800" **MinHeight**="600"
13. **WindowStartupLocation**="CenterScreen" **AllowsTransparency**="True" >
15. <WindowChrome.WindowChrome>
16. <WindowChrome **CaptionHeight**="50" **ResizeBorderThickness**="5"/>
17. </WindowChrome.WindowChrome>
19. <Border **Style**="{StaticResource RootStyle}">
20. <Grid>
22. <Grid.RowDefinitions>
23. <RowDefinition **Height**="10"/>
24. <RowDefinition **Height**="100"/>
25. <RowDefinition **Height**="\*"/>
26. <RowDefinition **Height**="10"/>
27. </Grid.RowDefinitions>
29. <Grid.ColumnDefinitions>
30. <ColumnDefinition **Width**="10"/>
31. <ColumnDefinition **Width**="200"/>
32. <ColumnDefinition **Width**="\*"/>
33. <ColumnDefinition **Width**="10"/>
34. </Grid.ColumnDefinitions>


38. <!--Row 1-->
39. <Border **BorderThickness**="0 0 0 1" **BorderBrush**="White" **Grid**.**Row**="1" **Grid**.**Column**="0"
40. **Grid**.**ColumnSpan**="4"/>
42. <!--Undo buttons-->
43. <StackPanel **Orientation**="Horizontal" **Grid**.**Row**="1" **Grid**.**Column**="1"
44. **HorizontalAlignment**="Left" **VerticalAlignment**="Top" **WindowChrome**.**IsHitTestVisibleInChrome**="True">
45. <Button **Style**="{StaticResource BackButton}" **Margin**="16,10,0,0"
46. **cal:Message**.**Attach**="[Event Click] = [Action BackButton()]"/>
47. <Button **Style**="{StaticResource ForwardButton}" **Margin**="0,10,0,0"
48. **cal:Message**.**Attach**="[Event Click] = [Action ForwardButton()]"/>
49. </StackPanel>
51. <!--Control Buttons-->
52. <StackPanel **Orientation**="Horizontal" **Grid**.**Row**="1" **Grid**.**Column**="1"
53. **Grid**.**ColumnSpan**="2" **HorizontalAlignment**="Right"
54. **VerticalAlignment**="Top" **WindowChrome**.**IsHitTestVisibleInChrome**="True">
55. <Button **Style**="{StaticResource MinimiseButton}"
56. **Margin**="0,10,0,0" **cal:Message**.**Attach**="Minimise($view)"/>
57. <Button **Style**="{StaticResource Maximse}"
58. **Margin**="16,10,0,0" **cal:Message**.**Attach**="Maximise($view)"/>
59. <Button **Style**="{StaticResource CloseButton}"
60. **Margin**="16,10,0,0" **cal:Message**.**Attach**="Exit($view)"/>
61. </StackPanel>
63. <Button **Style**="{StaticResource SettingButton}" **Grid**.**Row**="1" **Grid**.**Column**="1"
64. **HorizontalAlignment**="Left" **VerticalAlignment**="Bottom"
65. **Margin**="10,0,0,22"
66. **cal:Message**.**Attach**="[Event Click] = [Action Settings()]"/>
68. <!--Search Bar-->
69. <StackPanel **Grid**.**Row**="1" **Grid**.**Column**="1" **Grid**.**ColumnSpan**="3"
70. **VerticalAlignment**="Bottom">
72. <TextBlock **Style**="{StaticResource SearchBoxTitle}" **Grid**.**Row**="0" **Grid**.**Column**="0"
73. **Grid**.**ColumnSpan**="3" **Text**="{Binding Path=Name, Mode=OneWay}"/>
75. <xctk:WatermarkTextBox **Style**="{StaticResource CustomTextbox}" **Grid**.**Column**="0" **Grid**.**Row**="0"
76. **Grid**.**ColumnSpan**="3" **Margin**="50,0,16,16" **Padding**="15,0,0,0" **Height**="36"
77. **x:Name**="searchValue"
78. **cal:Message**.**Attach**="[Event KeyDown] = [Search($eventargs)]"/>
79. </StackPanel>
81. <!--Row Two-->
82. <!--Side Bar-->
84. <TreeView **Style**="{StaticResource CustomTreeView}" **Grid**.**Row**="2" **Grid**.**Column**="1"
85. **ItemsSource**="{Binding Path=Drives}" **ItemContainerStyle**="{StaticResource TreeViewItemBase}">
86. <TreeView.ItemTemplate>
87. <HierarchicalDataTemplate **ItemsSource**="{Binding Path=Children}">
88. <StackPanel **Orientation**="Horizontal" **Cursor**="Hand"
89. **cal:Message**.**Attach**="[Event MouseLeftButtonDown] = [Action OpenFromSideBar($dataContext)]">
90. <Image **Width**="25" **Source**="{Binding Path=Image}"/>
91. <TextBlock **VerticalAlignment**="Center" **Text**="{Binding Path=Name, Mode=OneWay}"
92. **cal:Message**.**Attach**=""/>
93. </StackPanel>
94. </HierarchicalDataTemplate>
95. </TreeView.ItemTemplate>
96. </TreeView>
98. <ScrollViewer **Grid**.**Row**="2" **Grid**.**Column**="2" **VerticalScrollBarVisibility**="Auto"
99. **HorizontalScrollBarVisibility**="Auto">
100. <ScrollViewer.ContextMenu>
101. <ContextMenu>
102. <MenuItem **Header**="New Folder" **Cursor**="Hand"
103. **cal:Message**.**Attach**=""/>
104. </ContextMenu>
105. </ScrollViewer.ContextMenu>
106. <ItemsControl **ItemsSource**="{Binding Path=ViewPortChildren}">
107. <ItemsControl.ItemTemplate>
108. <DataTemplate>
109. <StackPanel **Style**="{StaticResource FileItem}"
110. **cal:Message**.**Attach**="[Event MouseLeftButtonDown] = [Action OpenChild($dataContext, $eventargs)]">
111. <Image **Source**="{Binding Path=Image}"/>
112. <TextBlock **Text**="{Binding Path=Name}" **Style**="{StaticResource FolderOrFileNameBox}"
113. />
114. </StackPanel>
115. </DataTemplate>
116. </ItemsControl.ItemTemplate>
117. </ItemsControl>
118. </ScrollViewer>
119. </Grid>
120. </Border>
121. </Window>

 This is the RootView.xmal throughout the code you will see the key work Binding, this is where a variable from the RootViewModel is being bound to. As I am using the MVVM model structure binding method to event is different (cal:Message.Attach="[Event MouseLeftButtonDown] = [Action OpenChild($dataContext, $eventargs)]") is an example of how to do it. Luckily Caliburn.micro does a lot of the work in the background to make this work.

1. <Window **x:Class**="FileBrowser.Views.SettingsView"
2. **xmlns**="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3. **xmlns:x**="http://schemas.microsoft.com/winfx/2006/xaml"
4. **xmlns:d**="http://schemas.microsoft.com/expression/blend/2008"
5. **xmlns:mc**="http://schemas.openxmlformats.org/markup-compatibility/2006"
6. **xmlns:local**="clr-namespace:FileBrowser.Views"
7. **xmlns:cal**="http://www.caliburnproject.org"
8. **mc:Ignorable**="d" **AllowsTransparency**="True" **Background**="Transparent"
9. **ResizeMode**="NoResize" **WindowStartupLocation**="CenterScreen" **WindowStyle**="None"
10. **Title**="SettingsView" **Height**="300" **Width**="300">
12. <WindowChrome.WindowChrome>
13. <WindowChrome **CaptionHeight**="50"/>
14. </WindowChrome.WindowChrome>
16. <Border **Style**="{StaticResource RootStyle}">
17. <Grid>
19. <Grid.RowDefinitions>
20. <RowDefinition **Height**="50"/>
21. <RowDefinition **Height**="\*"/>
22. </Grid.RowDefinitions>
24. <!--Control Buttons-->
25. <Button **Style**="{StaticResource CloseButton}"
26. **WindowChrome**.**IsHitTestVisibleInChrome**="True"
27. **VerticalAlignment**="Center"
28. **Margin**="0,0,10,0" **cal:Message**.**Attach**="Exit()"/>
30. <TextBlock **Text**="Settings" **Foreground**="White" **VerticalAlignment**="Center"
31. **Grid**.**Row**="0" **FontSize**="24" **HorizontalAlignment**="Center"/>
33. <Border **Grid**.**Row**="0" **Background**="White" **BorderThickness**="0 0 0 1" **Height**="3"
34. **VerticalAlignment**="Bottom"/>
36. <StackPanel **Grid**.**Row**="1" **VerticalAlignment**="Center" **HorizontalAlignment**="Center"
37. **Width**="250" **Margin**="0,0,0,40">
38. <CheckBox **Content**="Show hidden items" **IsChecked**="{Binding Path=ShowHiddenItems, Mode=TwoWay}"
39. **Foreground**="White" **HorizontalAlignment**="Left"/>
40. <CheckBox **Content**="Only Show files or folder you've visited" **IsChecked**="{Binding Path=OnlyShowFilesVisted, Mode=TwoWay}"
41. **Foreground**="White" **Margin**="0,20,0,0" **HorizontalAlignment**="Left"/>
42. <CheckBox **Content**="Only show file you haven't visted yet" **IsChecked**="{Binding Path=OnlyShowFileNotVisted, Mode=TwoWay}"
43. **Foreground**="White" **Margin**="0,20,0,0" **HorizontalAlignment**="Left"/>
44. <CheckBox **Content**="Filter the directory instead of searching" **IsChecked**="{Binding Path=FilterNotSearch, Mode=TwoWay}"
45. **Foreground**="White" **Margin**="0,20,0,0" **HorizontalAlignment**="Left"/>
46. </StackPanel>

49. </Grid>
50. </Border>
51. </Window>

 The settings.xaml has a few things in common with the rootview.xaml is sharing of styles. The styles are stored in resource dictionaries and linked to the App.xaml allow for all windows in this application to access them. This keeps the UI design uniform across the whole application.

# Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | Test objective | Exceptive outcome | Result |
| 1 | Setting for Hiding folder and file works | For files and folders that have the hidden attribute to not been shown | Successful |
| 2 | Setting for Filter not search to filter the directory | Only folders and file that have the search value in their full path to be shown | Successful |
| 3 | Setting for Show only Visited | Only files we have visited should be shown | Successful |
| 4 | Setting for Don't show Visited | Folders you have visited should be hidden | Successful |
| 5 | Handle search in a large file directory | To come back with a result | Kind of successful freeze after 100MB of ram is used but if left alone will return the results |
| 6 | Regex search | To have no results as I wasn't able to incorporate regex searches yet | Successful |
| 7 | Has normal windows interaction | Should be able to snap, maximize, minimize and close | Successful |

Zoom in on the pictures if need to

|  |  |
| --- | --- |
| Test Number |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 | Window snap    The closing and minimize works as well |
|  | Video test |

# Evaluation