Welcome to AtomicMVVM

[](https://nuget.org/packages/AtomicMVVM)

The goal of AtomicMVVM is to allow developers to work with MVVM in their XAML based solutions (XPF, Silverlight, Windows Phone, WinRT) with minimal effort while not getting in the way of the code. It requires .NET 4 or higher.

# Index

* Feature Set
* Design
* AtomicPhoneMVVM
* Getting Started
  + WPF
  + Metro (Windows 8)
  + Silverlight 5
  + Windows Phone 7.1 & Pure AtomicMVVM
  + Windows Phone 7.1 & AtomicPhoneMVVM
* AtomicMVVM Features
  + Attributes
  + Global Commands
* AtomicStorage
  + Getting Started
* Troubleshooting
* License
* In the Wild
* Change Log
* Known Issues

# Feature Set

AtomicMVVM supports the following MVVM features:

* The bootstrapper handles injection of views into a shell, so you don’t have to.
* The bootstrapper handles joining and creating of views and view models, so you don’t have to.
* No more commands, just normal methods.
* No more CanExecute (ICommand again) worries, just normal methods.
* Can easily link the methods to the properties change notification, so you can update CanExecute or fire off code in a single line!
* Can pass information from one view to another.

# Design

The design goals of AtomicMVVM are (in order): simple over feature rich, unobtrusive over less code and convention over configuration.

## Simple over Feature Rich

AtomicMVVM isn’t going to set the world on fire because rather than build something monolithic that can do everything, in every way we decided to focus on the core of MVVM and build for that. This means it easily slots into projects and the source code also easy to understand:

* It is a trivial set of files - between two and three depending on the project type.
* It is less than 300 lines of code per project type.

What we trade-off for this is features and options, so it only works one way and if you don’t like that one way there isn’t much you can do about it unless you change the code and because the code is so simple that isn’t hard either!

An example of this choice is that we do not support convention binding for controls (like textboxes) to properties. Rather we expect you to just use normal XAML binding, where you get all the powerful features of that without us having to reinvent it.

## Unobtrusive over less code

While other MVVM frameworks allow you to do amazing things by writing TINY amounts of code, a lot of the time you take a penalty for that in the way of having a lot of references, or class inheritance. The moment you are unhappy with something (like having to inherit from a specific class) then you are forced to write massive amounts of code (reinventing the wheel).

Rather than model, we have decided to be unobtrusive – for example to convert a normal WPF project to use AtomicMVVM means adding one interface to your Main Window (Shell), implementing a single method, adding one static property to your app class and removing one line of XAML! So you may have a little more setup but in the end it gets out the way easier so you can easily use AtomicMVVM with anything you like.

## Convention over configuration

The final design goal is that there should be as little manual configuration as possible and that if you follow the convention (how we name things) then everything just works. This means the learning curve for you is very shallow, in fact we have only one configuration parameter that is required: What is the name of the shell class and what is the first bit of content to put in it!

# AtomicPhoneMVVM

The original idea behind AtomicMVVM was that it could just work everywhere – but there are places the model we use in AtomicMVVM just doesn’t work. Windows Phone is a key example of that sort of place.

In AtomicMVVM we are swopping the content of a frame out all the time, so you are never really leaving a single window, and while this works for Windows Phone – it also breaks a common design goal of the phone: The navigation stack.

The navigation stack in Windows Phone is controlled by the frame, so you can think of using AtomicMVVM with Windows Phone as having a frame within a frame and thus we lose navigation to the top most frames. The most obvious side effect of this is the back button on the device no longer works correctly ☹

To solve this we have created AtomicPhoneMVVM, a project that is in many ways similar but in many ways very different. We now do not have a concept of the Views, rather now the pages in the phone application are the Views, and each page is individually responsible for its own binding or put another way, no global boot strapper exists.

AtomicPhoneMVVM also has a bunch of features not applicable to AtomicMVVM:

* IWhenReady – this is an interface you can add to your ViewModel to be notified when the binding is finished – this allows you to update items post binding.
* Navigate – This method allows the ViewModel to do navigation on the Views (pages).
* AppBar support – The Windows Phone appbar menu and commands are not based off buttons and thus need special support to work with an MVVM framework.

# Getting started

## WPF

We will start off with an empty WPF project; just to show you how easy it is to get started!

1. Grab AtomicMVVM from Nuget - this will create the two folders needed, namely: Views & ViewModels.
2. In the code behind for MainWindow.xaml, have the class inherit from IShell and implement the interface. It has one method which is used to set the content of our ContentControl when the view changes.

public partial class MainWindow : Window, IShell

{

public MainWindow()

{

InitializeComponent();

}

public void ChangeContent(UserControl viewContent)

{

this.Content = viewContent;

}

}

1. Now we will create the view model. Right click on the ViewModels folder and add a new public class that inherits from CoreData called Simple (i.e. Simple.cs). This view model has a Name property, a method that says hello and a method that states if Name is not null or white space.

public class Simple : CoreData

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public void SayHi()

{

MessageBox.Show("Hello "+this.Name);

}

[ReevaluateProperty("Name")]

public bool CanSayHi()

{

return !string.IsNullOrWhiteSpace("Name");

}

}

Notes:

1. The Name property calls RaisePropertyChanged, which is a helper method to INotifyPropertyChanged.
2. The name of the SayHi method and its signature (returns void, takes no parameters). This is important for the convention based binding for commands.
3. The name of CanSayHi method, it’s convention of the Can prefix, and its signature (returns Boolean, takes no parameters). This is important for the convention based binding for commands.
4. The ReevaluateProperty attribute which tells the bootstrapper that when the Name property changes the result of this method changes too.
5. Now for the view, right click on the Views folder and add a UserControl named Simple (i.e. Simple.xaml) and in there we add the XAML to enter a name and click a button.

<UserControl x:Class="DocumentationDemo.Views.Simple"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="300">

<StackPanel>

<TextBlock Text="Name"/>

<TextBox Text="{Binding Name}"/>

<Button x:Name="SayHi" Content="Say Hello"/>

</StackPanel>

</UserControl>

Notes:

1. We use normal binding for the Text property of the TextBox to the Name property of the view model.
2. The Button (which is an ICommandSource) name is the same as our method name in the view model.
3. Open the App.xaml file and remove the StartUri attribute. If you don’t, you’ll end up with two windows, one that works and one that is blank.
4. Open the App.xaml.cs file and add a property or field of Bootstrapper<TShell,TContent>. The first generic type is the class to use as the shell (i.e. our MainWindow) and the second is the default view model to load into the shell.

public partial class App : Application

{

public static Bootstrapper<MainWindow, ViewModels.Simple> Bootstrapper {get;set;}

public App()

{

Bootstrapper = new Bootstrapper<MainWindow,ViewModels.Simple>();

Bootstrapper.Start();

}

}

Hint: I prefer a static property for the boot strapper as this allows you to access the boot strapper pretty much everywhere in your project by calling App.Bootstrapper.

1. In the constructor, or whenever you want to identify the system is ready, call the Start method on the boot strapper.

Press F5 and enjoy!

## Metro (Windows 8)

We will start off with an empty Metro project; just to show you how easy it is to get started!

1. Grab AtomicMVVM from Nuget - this will create the two folders needed, namely: Views & ViewModels.

Open your MainWindow.xaml (this will be our shell) and remove the Grid.

<Window x:Class="DocumentationDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="350" Width="525">

</Window>

In the code behind for MainWindow.xaml, have the class inherit from IShell and implement the interface. It has one method which is used to set the content of our ContentControl when the view changes.

public partial class MainWindow : Window, IShell

{

public MainWindow()

{

InitializeComponent();

}

public void ChangeContent(UserControl viewContent)

{

this.Content = viewContent;

}

}

Now we will create the view model. Right click on the ViewModels folder and add a new public class that inherits from CoreData called Simple (i.e. Simple.cs). This view model has a Name property, a method that says hello and a method that states if Name is not null or white space.

public class Simple : CoreData

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public async void SayHi()

{

await new MessageDialog().ShowAsync("Hello "+this.Name);

}

[ReevaluateProperty("Name")]

public bool CanSayHi()

{

return !string.IsNullOrWhiteSpace("Name");

}

}

Notes:

* 1. The Name property calls RaisePropertyChanged, which is a helper method to INotifyPropertyChanged.
  2. The name of the SayHi method and its signature (returns void, takes no parameters). This is important for the convention based binding for commands.
  3. The name of CanSayHi method, it’s convention of the Can prefix, and its signature (returns Boolean, takes no parameters). This is important for the convention based binding for commands.
  4. The ReevaluateProperty attribute which tells the bootstrapper that when the Name property changes the result of this method changes too.

Now for the view, right click on the Views folder and add a UserControl named Simple (i.e. Simple.xaml) and in there we add the XAML to enter a name and click a button.

<UserControl x:Class="DocumentationDemo.Views.Simple" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="300">

<StackPanel>

<TextBlock Text="Name"/>

<TextBox Text="{Binding Name}"/>

<Button x:Name="SayHi" Content="Say Hello"/>

</StackPanel>

</UserControl>

Notes:

* 1. We use normal binding for the Text property of the TextBox to the Name property of the view model.
  2. The Button name is the same as our method name in the view model.

Open the App.xaml.cs file and add a property of Bootstrapper<TShell,TContent>. The first generic type is the class to use as the shell (i.e. our MainWindow) and the second is the default view model to load into the shell.

public partial class App : Application

{

public static Bootstrapper<MainWindow, ViewModels.Simple> Bootstrapper {get;set;}

}

Next replace the OnLaunched method content so that it creates the Bootstrapper and calls the start method of the boot strapper.

private void OnLaunched(LaunchActivatedEventArgs args)

{

Bootstrapper = new Bootstrapper<MainWindow, ViewModels.Simple>();

//todo: set any global commands here

Bootstrapper.Start();

}

Press F5 and enjoy

## Silverlight 5

1. We will start off with an empty Silverlight project; just to show you how easy it is to get started!
2. Grab AtomicMVVM from Nuget - this will create the two folders needed, namely: Views & ViewModels.

Open your MainWindow.xaml (this will be our shell) and remove the Grid.

<Window x:Class="DocumentationDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="350" Width="525">

</Window>

In the code behind for MainWindow.xaml, have the class inherit from IShell and implement the interface. It has one method which is used to set the content of our ContentControl when the view changes.

public partial class MainWindow : Window, IShell

{

public MainWindow()

{

InitializeComponent();

}

public void ChangeContent(UserControl viewContent)

{

this.Content = viewContent;

}

}

Now we will create the view model. Right click on the ViewModels folder and add a new public class that inherits from CoreData called Simple (i.e. Simple.cs). This view model has a Name property, a method that says hello and a method that states if Name is not null or white space.

public class Simple : CoreData

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public void SayHi()

{

MessageBox.Show("Hello "+this.Name);

}

[ReevaluateProperty("Name")]

public bool CanSayHi()

{

return !string.IsNullOrWhiteSpace("Name");

}

}

Notes:

* 1. The Name property calls RaisePropertyChanged, which is a helper method to INotifyPropertyChanged.
  2. The name of the SayHi method and its signature (returns void, takes no parameters). This is important for the convention based binding for commands.
  3. The name of CanSayHi method, it’s convention of the Can prefix, and its signature (returns Boolean, takes no parameters). This is important for the convention based binding for commands.
  4. The ReevaluateProperty attribute which tells the bootstrapper that when the Name property changes the result of this method changes too.

Now for the view, right click on the Views folder and add a UserControl named Simple (i.e. Simple.xaml) and in there we add the XAML to enter a name and click a button.

<UserControl x:Class="DocumentationDemo.Views.Simple" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="300">

<StackPanel>

<TextBlock Text="Name"/>

<TextBox Text="{Binding Name}"/>

<Button x:Name="SayHi" Content="Say Hello"/>

</StackPanel>

</UserControl>

Notes:

* 1. We use normal binding for the Text property of the TextBox to the Name property of the view model.
  2. The Button name is the same as our method name in the view model.

Open the App.xaml.cs file and add a property or field of Bootstrapper<TShell,TContent>. The first generic type is the class to use as the shell (i.e. our MainWindow) and the second is the default view model to load into the shell.

public partial class App : Application

{

public static Bootstrapper<MainWindow, ViewModels.Simple> Bootstrapper {get;set;}

}

Next replace the Application\_Startup method content so that it creates the Bootstrapper and calls the Start method to tie everything together.

private void Application\_Startup(object sender, StartupEventArgs e)

{

Bootstrapper = new Bootstrapper<MainWindow, ViewModels.Simple>();

//todo: set any global commands here

Bootstrapper.Start();

}

Press F5 and enjoy

## Windows Phone 7.1 & Pure AtomicMVVM

Before you begin with Windows Phone 7.1, I urge you to read the AtomicPhoneMVVM section to decide if this is the correct MVVM framework for you.

1. We will start off with an empty Windows Phone project; just to show you how easy it is to get started!
2. Grab AtomicMVVM from Nuget - this will create the two folders needed, namely: Views & ViewModels.

Open your MainPage.xaml (this will be our shell) and remove the Grid.

<phone:PhoneApplicationPage

x:Class="WP71Demo.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"

xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"

FontFamily="{StaticResource PhoneFontFamilyNormal}"

FontSize="{StaticResource PhoneFontSizeNormal}"

Foreground="{StaticResource PhoneForegroundBrush}"

SupportedOrientations="Portrait" Orientation="Portrait"

shell:SystemTray.IsVisible="True">

</phone:PhoneApplicationPage>

In the code behind for MainPage.xaml, have the class inherit from IShell and implement the interface. It has one method which is used to set the content of our ContentControl when the view changes.

public partial class MainPage : PhoneApplicationPage, IShell

{

public MainWindow()

{

InitializeComponent();

}

public void ChangeContent(UserControl viewContent)

{

this.Content = viewContent;

}

}

Now we will create the view model. Right click on the ViewModels folder and add a new public class that inherits from CoreData called Simple (i.e. Simple.cs). This view model has a Name property, a method that says hello and a method that states if Name is not null or white space.

public class Simple : CoreData

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public void SayHi()

{

MessageBox.Show("Hello "+this.Name);

}

[ReevaluateProperty("Name")]

public bool CanSayHi()

{

return !string.IsNullOrWhiteSpace("Name");

}

}

Notes:

* 1. The Name property calls RaisePropertyChanged, which is a helper method to INotifyPropertyChanged.
  2. The name of the SayHi method and its signature (returns void, takes no parameters). This is important for the convention based binding for commands.
  3. The name of CanSayHi method, it’s convention of the Can prefix, and its signature (returns Boolean, takes no parameters). This is important for the convention based binding for commands.
  4. The ReevaluateProperty attribute which tells the bootstrapper that when the Name property changes the result of this method changes too.

Now for the view, right click on the Views folder and add a UserControl named Simple (i.e. Simple.xaml) and in there we add the XAML to enter a name and click a button.

<UserControl x:Class="WP71Demo.Views.Simple" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="300">

<StackPanel>

<TextBlock Text="Name"/>

<TextBox Text="{Binding Name}"/>

<Button x:Name="SayHi" Content="Say Hello"/>

</StackPanel>

</UserControl>

Notes:

* 1. We use normal binding for the Text property of the TextBox to the Name property of the view model.
  2. The Button name is the same as our method name in the view model.

Open the App.xaml.cs file and add a property or field of Bootstrapper<TShell,TContent>. The first generic type is the class to use as the shell (i.e. our MainWindow) and the second is the default view model to load into the shell.

public partial class App : Application

{

public static Bootstrapper<MainWindow, ViewModels.Simple> Bootstrapper {get;set;}

1. Next modify the InitializePhoneApplication method content so that it creates the Bootstrapper and calls the Start method to tie everything together. You also need to remove all the RootFrame code. You can also clean out RootFrame from the rest of the app class too.

private void InitializePhoneApplication()

{

if (phoneApplicationInitialized)

return;

Bootstrapper = new Bootstrapper<MainPage, Main>();

Bootstrapper.Start();

phoneApplicationInitialized = true;

}

1. Press F5 and enjoy

## Windows Phone 7.1 & AtomicPhoneMVVM

1. We will start off with a blank Windows Phone project.
2. Grab Atomic**Phone**MVVM from Nuget - this will create the one folder we need, namely: Views.
3. In Views add a class named MainPage (i.e. mainpage.cs) and have your new class inherit from CoreData.
4. Add some properties that use INotifyPropertyChanged via the RaisePropertyChanged method and some methods.

public class MainPage : CoreData

{

private string \_name;

private string message;

public string Message

{

get { return message; }

set

{

if (value != message)

{

message = value;

RaisePropertyChanged("Message");

}

}

}

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public void SayHi()

{

Message = "Hi " + Name;

}

[ReevaluateProperty("Name")]

public bool CanSayHi()

{

return !string.IsNullOrWhiteSpace("Name");

}

}

1. Next let’s add some UI controls to our page – we will have a TextBox, TextBlock and Button controls to the MainPage.xaml. Note we use standard binding for the TextBox and TextBlock and how we set the button name to be the same as our method name.

<phone:PhoneApplicationPage

x:Class="AtomicPhoneMVVMDemo.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"

xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"

FontFamily="{StaticResource PhoneFontFamilyNormal}"

FontSize="{StaticResource PhoneFontSizeNormal}"

Foreground="{StaticResource PhoneForegroundBrush}"

SupportedOrientations="Portrait" Orientation="Portrait"

shell:SystemTray.IsVisible="True">

<StackPanel>

<TextBox Text="{Binding Name , Mode=TwoWay}"/>

<Button x:Name="SayHi" Content="Say Hello"/>

<TextBlock Text="{Binding Message}"/>

</StackPanel>

</phone:PhoneApplicationPage>

In the code behind for the page add the AtomicPhoneMVVM namespace, and then in the constructor call the BindData extension method passing in the View class that we created earlier.

using AtomicPhoneMVVM;

public partial class MainPage : PhoneApplicationPage

{

// Constructor

public MainPage()

{

InitializeComponent();

this.BindData<Views.MainPage>();

}

}

Press F5 and enjoy.

# AtomicMVVM Features

This section contains information on features found in AtomicMVVM. For the most part these features are available in AtomicPhoneMVVM too.

## Attributes

### ReevaluateProperty

In the getting started we introduced one attribute, ReevaluateProperty which is used on Can methods, i.e. methods that are prefixed with Can and tell the Command that is created if it can be executed. By adding these attributes to the method it tells the bootstrapper, which properties affect the Can method result. Note: You can have multiple ReevaluateProperty attributes on a method.

### TriggerProperty

The other attribute is called TriggerProperty and it is used in a similar way to ReevaluateProperty, except it causes the attributed method to fire when the property updates. This is useful for scenarios where one property change causes the value of other properties to change.

[TriggerProperty("SelectedMake")]

public void SetModels()

{

if (string.IsNullOrWhiteSpace(this.SelectedMake))

{

this.Models.Clear();

return;

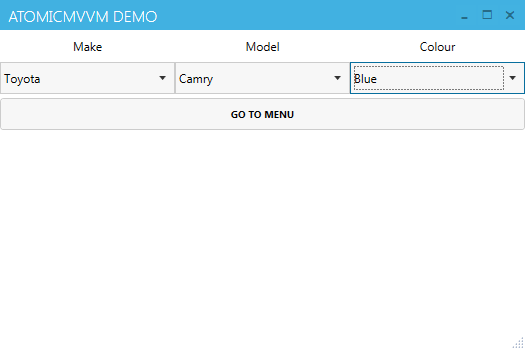
}

var models = data.Single(\_ => \_.Name == this.SelectedMake).Data.Select(\_ => \_.Name);

UpdateCollection(this.Models, models);

}

An example of this is in the general demo app, in the Car Selector demo, where changing the Make dropdown changes the values of the Model and Colour dropdowns and changing the Model drop down changes the value of the Colour drop down.



Note: It is possible to create an infinite loop, which will result in a stack overflow if the attributed method updates any of the properties that attribute it – so be careful.

This attribute may seem not to be a MVVM item and thus violates the first design goal, but this is so easily implemented with the bootstrapper and it is so common that any other way to do this would not be correct.

## Global Commands

The bootstrapper has a property named GlobalCommands which is a List<Tuple<string,Action>>. This list stores a set of global commands that can be used when the view model does not contain a method already. It works on a similar process, the name of the ICommandSource (for example a button) must match the string item of the Tuple, if it does and there is no command in the view model that matches it will run the action item of the Tuple.

Bootstrapper.GlobalCommands.Add("GoToMenu", () => App.Bootstrapper.ChangeView<Menu>());

This is useful where you have a common set of commands that are used in many view models and you do not wish to repeat them in each view model.

An example of this is in the general demo app, where most views have a Button that allows you to return to the menu. The Button is always named GoToMenu and no view models have a method like that, so the global command is used rather.

# AtomicStorage

AtomicStorage is a spin off project from AtomicMVVM & AtomicPhoneMVVM, where it aims to make the storage of data easy and simple for systems that have a process lifecycle – in other words it means that suspend/resume issues on Windows Phone and with Windows 8 (Metro) can easily be handled.

An example of this can be found in the AtomicPhoneMVVM demo code.

## Getting Started

1. To get started first grab AtomicStorage from Nuget.
2. For the view class that you want to enable this feature on, change it so it inherits from IDataStore. This interface has no methods that need to be implemented.
3. Next attribute any properties that you want to store with the Storage attribute and give it a name. Names must be unique.
4. Chose logical places in your class to do the loading (for example the constructor) and the saving (for example when a property changes). This is done with the LoadSettings and SaveSettings extension methods respectively.
5. If you are using this with a WinRT application, then you can pass true to those methods to have them store the settings in the roaming cloud profile.
6. Done & Profit ☺

public class MainPage : CoreData, IDataStore

{

[Storage("Name")]

public string Name

{

get { return \_name; }

set

{

if (\_name != value)

{

\_name = value;

RaisePropertyChanged("Name");

this.SaveSettings();

}

}

}

public MainPage()

{

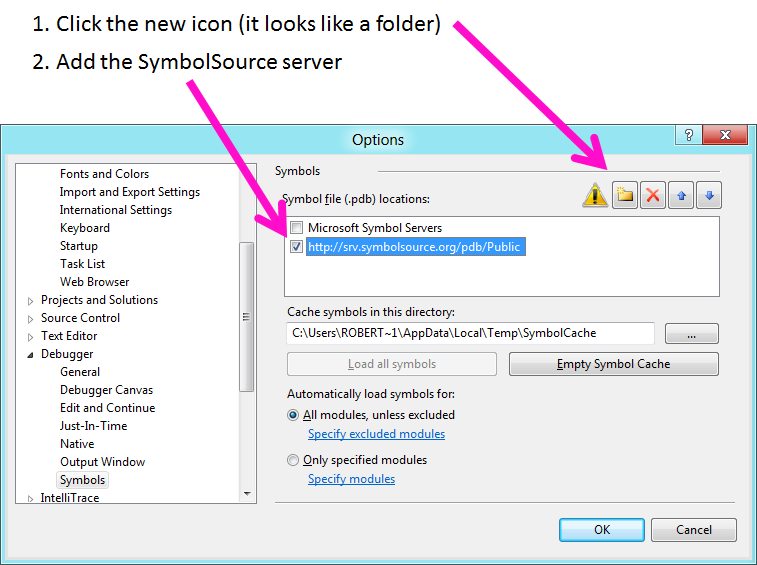
this.LoadSettings();

}

# Troubleshooting

So you think you have found an issue in AtomicMVVM but aren't 100% sure? Or you KNOW there is a bug but you want more insight for the fantastic bug report you are going to send? You could download the source, compile it yourself and change all the references and recreate it that way - but seems like a lot of work.

So to make it easier we support SymbolSource.org which is a public location for project source code and symbols (those .pdb files). To use it, simply in Visual Studio go to Tools, Options, Debugger and Symbols. Once there add the SymbolSource.org server and you will get full symbol and source code support!



# License

AtomicMVVM is released under a [MS-PL license](http://www.opensource.org/licenses/ms-pl.html959595), so that it’s friendly to developers of both open and closed source software.

# in the Wild

## SharePoint Holiday Loader

* Site: <https://bitbucket.org/rmaclean/sharepoint-holiday-loader/overview>
* Details: WPF application that makes use of AtomicMVVM

## Postal Codes

* Site: <https://bitbucket.org/rmaclean/postal-codes/overview>
* Details: Windows Phone 7.1 application that makes use of AtomicPhoneMVVM

# Change Log

Version numbering follows Semantic Versioning 1.0.0: <http://semver.org/spec/v1.0.0.html>

## AtomicMVVM

### 3.0.0

* Significant changes to how the boot strapper works
  + Can no longer pass in commands to the boot strapper constructor (work done in 1.0.11).
  + You must call the Start method to start the binding.
* Added support (finally) for Windows Phone 7.1 – not that it was a good idea.
* Switched from Mercurial to Git.
* Massive amounts of clean up to the WinRT version.

### 2.0.1

* Updated to support Visual Studio “11” & Windows 8 Consumer Preview changes to Metro style app changes.

### 2.0.0

* Changed the way attributes (Trigger & Reevaluate work). No longer can you do multiple. Now you can only have one, but with multiple properties. This solves some edge case bugs & improves performance.
* Added small tweak around warning of unused event.
* Create the Views and ViewModels folder for you (Nuget package).

### 1.0.11

* Bug fixes for WinRT command support (grr)
* Ability to pass in global commands in the constructor so they can be applied to the first view too.

### 1.0.10

* Improved the Nuget package quality (and managed to knock off about 90% of the file size too) and published a symbols version.

### 1.0.9

* Updated Nuget with a different icon.

### 1.0.8

* Updated Nuget with icon.

### 1.0.7

* Bug fixes for Metro & Silverlight 5 support.

### 1.0.6

* Added support for .NET 4.5, Silverlight 5 & Metro in NuGetPackage.

### 1.0.5

* Added support for non-default view constructors which allows for data passing.
* Added support for global app commands.

### 1.0.4

* Building for .NET 4 client profile now (rather than 4.5 as was doing).

### 1.0.3

* Doesn't help if I don't include the updated assembly. So this is 1.0.2 proper.

### 1.0.2

* Updated project to be the bitbucket site.
* Helps if I make classes public (doh).
* Added trigger property logic.
* Fixed bug in the type location.

### 1.0.1

* First release.

## AtomicPhoneMVVM

### 1.0.0

Pretty much everything is new and shiny captain.

## AtomicStorage

### 1.0.0

Pretty much everything is new and shiny captain.

# Known Issues

No known issues! That either means there are none (because we are awesome) or no one has reported any!