Lakana v2 WPF Framework

Quickstart

[1 Lakana WPF Navigation 3](#_Toc355884379)

[1.1 Assemblies 3](#_Toc355884380)

[1.2 Setting up the main workspace 3](#_Toc355884381)

[1.3 Setting up views and view models 3](#_Toc355884382)

[1.4 Simple navigation to a view 3](#_Toc355884383)

[1.5 Stacking views 4](#_Toc355884384)

[1.6 Displaying modal views 4](#_Toc355884385)

[1.7 Displaying message box 5](#_Toc355884386)

[1.8 Closing view 5](#_Toc355884387)

[1.9 Closing application 5](#_Toc355884388)

[2 Lakana WPF Forms 7](#_Toc355884389)

[2.1 Assemblies 7](#_Toc355884390)

[2.2 Basics 7](#_Toc355884391)

[2.3 Create a view model for an editable form 7](#_Toc355884392)

[2.4 Associate the Model and the View Model 8](#_Toc355884393)

[2.5 Create view model properties 8](#_Toc355884394)

[2.6 Create view model commands (Optional) 9](#_Toc355884395)

[2.7 Associate a validation engine 9](#_Toc355884396)

# Lakana WPF Navigation

## Assemblies

In order to use Lakana WPF Navigation, you’ll have to reference two assemblies: **GasyTek.Lakana.Navigation.dll** and **GasyTek.Lakana.Common.dll** from your project.

Prior to version 2, only GasyTek.Lakana.Navigation.dll was required.

## Setting up the main workspace

First of all, you have to set up the main workspace. The main workspace is the main control that will host all of your views; actually we will use a Grid control.

In order to do so, reference Lakana namespace in your main window Xaml code.

|  |
| --- |
| xmlns:lakana="http://schemas.gasytek.com/wpf/lakana" |

Then set up the attached property NavigationManager.IsMainWorkspace to true on the Grid control that will serve as the main workspace.

|  |
| --- |
| <Grid x:Name="MainWorkspace" lakana:NavigationManager.IsMainWorkspace="True" /> |

## Setting up views and view models

Each view has to be identified by a unique identifier so that it can be reached by the navigation engine. In order to do so, you have to decorate your views with the ViewKeyAttribute attribute.

|  |
| --- |
| [ViewKey("MyViewUniqueId")]  public partial class MyView … |

You can also create the corresponding view model if you apply MVVM rules.

## Simple navigation to a view

As far as setting up the application is concerned, that’s all!

Now you can begin navigating to your first view and in order to do so, you’ll use the NavigationManager object.

|  |
| --- |
| // Navigate to MyView  NavigationManager.NavigateTo(“MyViewUniqueId”); |

The navigation engine will look for the view that has the specified unique id, if it doesn’t exist yet, it will be instantiated otherwise the current instance will be fetched then added to the main workspace.

You can have many instances of the same view; you can achieve that by appending “**#**” to the view unique id then append a key that will distinguish this view instance from others. Here is the pattern:

**viewUniqueId [ # key ]**

where part specified between [ .. ] is optional.

|  |
| --- |
| // Navigate to an instance of MyView  NavigationManager.NavigateTo(“MyViewUniqueId**#view1**”);  // Navigate to another instance of MyView  NavigationManager.NavigateTo(“MyViewUniqueId**#view2**”); |

## Stacking views

If you want to stack a view on top of another existing one, you’ll have to specify both views unique identifier. Here is the pattern:

**parent**V**iewUniqueId [ # key ] / child**V**iewUniqueId [ # key ]**

|  |
| --- |
| // Navigate to MyView  NavigationManager.NavigateTo(“MyParentViewUniqueId**/MyChildViewUniqueId**”); |

Notice that the parent view must already exist before you can stack another view on top of it. Notice also the use of the “**/**” character to separate the parent and child view unique ids.

## Displaying modal views

To display a view as modal on top of another existing one, you use the same navigation key pattern as you use when stacking views.

|  |
| --- |
| var modalResult = NavigationManager.ShowModal<string>("MyParentView/MyChildView"); |

Now MyChildView will be displayed as modal on top of MyParentView and when you close the modal view, it will return a result of type string to its parent.

ShowModal method will return a ModalResult object. It has a property named AsyncResult which type is .NET Task object. You can await for this task to obtain the actual modal result returned by the modal view.

## Displaying message box

To display message boxes, you’ll have to specify the parent unique identifier.

|  |
| --- |
| var messageBoxResult = NavigationManager.ShowMessageBox("MyParendUniqueId", "Save changes ?", MessageBoxImage.Question, MessageBoxButton.YesNoCancel); |

It will return a Task object that you can await for in order to fetch the actual result.

## Closing view

In order to close a view, you’ll have to specify its current key. By implementing the IViewKeyAware interface, your view or view model will be automatically aware of its actual view instance key.

|  |
| --- |
| public class MyViewModel : IViewKeyAware  {  public string ViewInstanceKey { get; set; }  } |

Then you can close the current view:

|  |
| --- |
| public class MyViewModel : IViewKeyAware  {  public string ViewInstanceKey { get; set; }  public void CLoseMyView()  {  NavigationManager.Close(ViewInstanceKey);  }  } |

## Closing application

|  |
| --- |
| NavigationManager.CloseApplication(); |

This will try to close your application. However if some of your views or view models needs to be closed gracefully, the closing process will be suspended the navigation engine will give you a last chance to close all of your views and view models.

In order to make your views or view models support the gracefully shutdown, you’ll have to implement ICloseable interface.

|  |
| --- |
| public class MyViewModel : ICloseable  {  public bool CanClose()  {  // put the logic here  }  } |

# Lakana WPF Forms

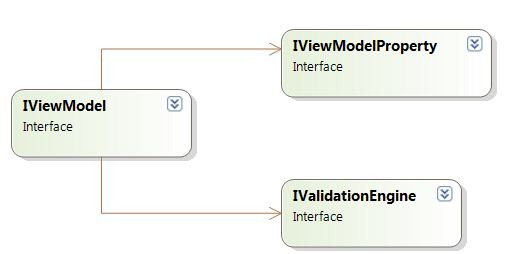
## Assemblies

In order to use Lakana WPF Navigation, you’ll have to reference two assemblies: **GasyTek.Lakana.Mvvm.dll** and **GasyTek.Lakana.Common.dll** from your project.

## Basics

Lakana WPF Forms relies on three core interfaces: IViewModel, IViewModelProperty and IValidationEngine.

* IViewModel: represents a view model that supports view model properties, view model commands and a validation engine.
* IViewModelProperty: represents the concept of rich property that is most of the time attached to IViewModel instance.
* IValidationEngine: abstracts a field validation engine that supports both synchronous and asynchronous operations. A validation engine is attached to IViewModel.



## Create a view model for an editable form

To create a view model that will be used to edit your data through a form, you have to inherit from the abstract class EditableViewModelBase which is indirect implementers of IViewModel.

It is a generic class and takes the type of the Model object that you want to edit, as parameter. For instance, if you want to edit an Employee object, you will create an EmployeeViewModel that inherits from EditableViewModelBase<Employee>. In an n-tiers scenario, your Data Transfer Object (DTO) is a good candidate to be used as Model object for the view model.

|  |
| --- |
| public class MyEditableViewModel : EditableViewModelBase<MyModel> |

## Associate the Model and the View Model

To associate the Model and View Model objects, you just have to assign the "Model" property of the view model.

|  |
| --- |
| var myViewModel = new MyEditableViewModel();  myViewModel.Model = myModelInstance; |

## Create view model properties

The next step is to create view model properties. You cannot instantiate view model properties via the traditional "new" operator. Rather, the view model provides factory methods to create them. There are three kinds of view model property:

* IValueViewModelProperty: which wraps a single-valued property
* ILookupValueViewModelProperty: which wraps multi-valued property. Only one value at a time can be selected.
* IEnumViewModelProperty: which is a special kind of ILookupValueViewModelProperty and it wraps enumeration property.

Each view model property will wrap each property of the model that you want to be editable. For instance, if you want "Name" property of the Model to be editable, you will create "Name" view model property on the view model. "Name" is a single valued property so you will create a Value View Model property.

|  |
| --- |
| public class MyEditableViewModel : EditableViewModelBase<MyModel>  {  public IValueViewModelProperty<string> Name { get; set; }  protected override void OnCreateViewModelProperties()  {  Name = this.CreateValueProperty(Model.Name);  }  …  } |

Note here that the **OnCreateViewModelProperties** method is the right place to create a view model property because at this stage, you can be sure that the Model and the View Model are tied together, so that you can use the Model's values safely. Under the hood, this method is called right after you assigned the model to the view model.

## Create view model commands (Optional)

You can also create view model commands by overriding OnCreateViewModelCommands method. View model commands are automatically refreshed whenever any view model property value is changed. Two obvious commands are provided by default by the view model: SaveCommand and CancelCommand.

## Associate a validation engine

Fields on editable forms often need a validation mechanism to avoid errors, and to keep your data coherent. To provide a validation engine to the view model, you implement the **OnCreateValidationEngine** method. Validation takes place whenever a view model property value is changed. Three validation engines are provided out of the box: **Data annotations, Fluent Validation and Custom**.

* Data annotations

Data annotations are part of the .NET Framework. They offer a metadata-oriented validation rules, meaning you decorate your properties with attributes that represent validation rules. The validation engine to use for this scenario is DataAnnotationValidationEngine.

|  |
| --- |
| public class MyEditableViewModel : EditableViewModelBase<MyModel>  {  [Required("Name is required")]  public IValueViewModelProperty<string> Name { get; set; }  protected override void OnCreateViewModelProperties()  {  Name = this.CreateValueProperty(Model.Name);  }  protected override IValidationEngine OnCreateValidationEngine()  {  return new DataAnnotationValidationEngine();  }  } |

* Fluent Validation

The Fluent Validation API allows you to define your validation rules in a fluent and more accurate manner. Using it will improve the readability of your code. The validation engine to use for this scenario is an implementation of FluentValidationEngine.

|  |
| --- |
| public class MyEditableViewModel : EditableViewModelBase<MyModel>  {  public IValueViewModelProperty<string> Name { get; set; }  protected override void OnCreateViewModelProperties()  {  Name = this.CreateValueProperty(Model.Name);  }  protected override IValidationEngine OnCreateValidationEngine()  {  return new MyFluentValidationEngine(this);  }  }  public class MyFluentValidationEngine : FluentValidationEngine<MyEditableViewModel>  {  public MyFluentValidationEngine(MyEditableViewModel viewModelInstance)  : base(viewModelInstance)  {  }  protected override void OnDefineRules()  {  // Example of simple Fluent rules  Property<string>(vm => vm.Name).Is.Required().Otherwise("Name required.");  }  } |

* Custom Validation

To create your own validation engine that can still be used with the framework, you can inherit from ValidationEngineBase.