## Getting Started with DevExpress MVVM Framework. Commands and View Models.

**Although this post is still actual, there is another post that describes a new modern style for defining commands and bindable properties in ViewModels:** [**POCO ViewModels**](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)**.**

If you’ve developed a WPF or Silverlight application, you’ve likely used the MVVM pattern. You likely also encountered issues implementing some functionality under MVVM. Some mechanisms are difficult to implement without moving away from MVVM. What’s more, MVVM limitations may not relate to any particular control because the WPF/Silverlight platform itself has no full support for MVVM development. The typical problems with MVVM in WPF/Silverlight are well-known. In fact, several frameworks were specifically introduced to address these issues: *PRISM, MVVM Light, Caliburn*, etc. These frameworks can work in conjunction with DevExpress components as with standard components.

*If several solutions already exist, why would DevExpress offer yet another MVVM framework?*   
The answer is, the better part of our customers use the MVVM pattern (including third party MVVM frameworks); we have a clear picture of the challenging scenarios in MVVM development. We can address issues solely within our components, but sometimes issues are related to the MVVM framework.   
With an MVVM library of our own, MVVM capabilities are exposed at the component level.The end-result is a holistic solution for a well-designed MVVM application whose parts fit together perfectly.

In this series of posts we’ll discuss the following features provided by **DevExpress MVVM Framework**:

1. Commands
2. Basic ViewModel classes
3. Attached behaviors
4. Services
5. Value converters
6. Messaging and loosely-coupled MVVM architecture

The MVVM functionality at the **DevExpress.Xpf.Mvvm** library has no external dependencies, so you can use it in the view model part of your project without referencing a UI library.

*Let’s review the commanding mechanism and ViewModel types. While you may find this subject familiar, the details are specific to our framework.*

The MVVM library provides two implementations of the **ICommand** interface:

1. **DelegateCommand** defining a parameterless command;
2. **DelegateCommand<T>** defining a single parameter command of the parametrized type.

Both commands support two constructors: a constructor accepting **Execute** delegate; a second constructor accepting **Execute** and **CanExecute** delegates:

1: DelegateCommand delegateCommand =

2: new DelegateCommand(() => MessageBox.Show("This is a DelegateCommand);

3: DelegateCommand<string> delegateCommand =

4: new DelegateCommand<string>(x => MessageBox.Show(x), x => !string.IsNullOrEmpty(x));

A DelegateCommand<T> automatically converts the command argument to the parameterized type if possible. For example, a CommandParameter string is converted to the parametrized type:

1: <Button Command="{Binding ShowDocumentCommand}" CommandParameter="Text"/>

1: public enum DocumentType { Text, Data }

2: DelegateCommand<DocumentType> ShowDocumentCommand =

3: new DelegateCommand<DocumentType>(OnShowDocumentCommandExecute);

4:

5: void OnShowDocumentCommandExecute(DocumentType parameter) {

6: if(parameter == DocumentType.Text) {

7: ///

8: }

9: if(parameter == DocumentType.Data) {

10: ///

11: }

12: }

An optional constructor parameter (for WPF only) specifies whether a DelegateCommand uses the [**CommandManager**](http://msdn.microsoft.com/en-us/library/System.Windows.Input.CommandManager.aspx) to raise the **CanExecuteChanged** event. By default useCommandManager is true, making it unnecessary to manually implement disabled/enabled logic for your commands. You can just set the CanExecute delegate for the command and the delegate is automatically triggered when an end-user interacts with the UI. Note that when the CommandManager is used for this purpose, the CanExecute handler is called frequently, so avoid performing time-consuming operations in the delegate.

If you pass false as the last constructor argument, the CommandManager is not used. In this case, update your command by calling the **RaiseCanExecuteChanged** method.

1: DelegateCommand GoBackCommand =

2: new DelegateCommand(OnGoBackCommandExecute, OnGoBackCommandCanExecute, false);

3: DelegateCommand GoForwardCommand =

4: new DelegateCommand(OnGoForwardCommandExecute, OnGoForwardCommandCanExecute, false);

5: void OnGoBackCommandExecute() {

6: ///

7: UpdateCommandsState();

8: }

9: void OnGoForwardCommandExecute() {

10: ///

11: UpdateCommandsState();

12: }

13: bool OnGoBackCommandCanExecute() {

14: ///

15: }

16: bool OnGoForwardCommandCanExecute() {

17: ///

18: }

19: void UpdateCommandsState() {

20: GoBackCommand.RaiseCanExecuteChanged();

21: GoForwardCommand.RaiseCanExecuteChanged();

22: }

A DelegateCommands can process an event in the view layer using the **EventToCommand** class. The following is a simple example binding a command to an event:

1: <Window ...>

2: <dxmvvm:Interaction.Triggers>

3: <dxmvvm:EventToCommand Command="{Binding InitializeCommand}" EventName="Loaded"/>

4: </dxmvvm:Interaction.Triggers>

5: ...

6: </Window>

The EventToCommand class allows passing event arguments as a command argument. It’s importantly to note in most cases event arguments are a part of the View layer, and it’s necessary to convert event arguments to an object that belonging to the ViewModel. EventToCommand provides an **EventArgsConverter** property for this purpose.

1: <ListBox ...>

2: <dxmvvm:Interaction.Triggers>

3: <dxmvvm:EventToCommand EventName="MouseDoubleClick"

4: Command="{Binding ItemDoubleClickCommand}"

5: PassEventArgsToCommand="True">

6: <dxmvvm:EventToCommand.EventArgsConverter>

7: <Helpers:ListBoxDoubleClickEventArgsConverter/>

8: </dxmvvm:EventToCommand.EventArgsConverter>

9: </dxmvvm:EventToCommand>

10: </dxmvvm:Interaction.Triggers>

11: </ListBox>

1: public class ListBoxDoubleClickEventArgsConverter : EventArgsConverterBase<MouseButtonEventArgs> {

2: protected override object Convert(MouseButtonEventArgs args) {

3: ///

4: }

5: }

Some DevExpress components provide ready-to-use event argument converters. For instance, the **EventArgsToDataRowConverter** and **EventArgsToDataCellConverter** converters can be used with the **GridControl**. These converters pass a data row object or information about a cell to a command.

1: <dxg:GridControl ... >

2: <dxmvvm:Interaction.Triggers>

3: <dxmvvm:EventToCommand Command="{Binding EditCommand}"

4: EventName="MouseDoubleClick"

5: PassEventArgsToCommand="True">

6: <dxmvvm:EventToCommand.EventArgsConverter>

7: <dx:EventArgsToDataRowConverter />

8: </dxmvvm:EventToCommand.EventArgsConverter>

9: </dxmvvm:EventToCommand>

10: </dxmvvm:Interaction.Triggers>

11: </dxg:GridControl>

*Let’s review the classes for ViewModel creation.*

**BindableBase** is a simple implementation of the **INotifyPropertyChanged** interface with two additional methods for implementing view model properties: **SetProperty** and **RaisePropertyChanged**. The following example demonstrates simple and complex scenarios for these methods.

1: public class BindableObject : BindableBase {

2: string stringProperty1;

3: public string StringProperty1 {

4: get { return stringProperty1; }

5: set { SetProperty(ref stringProperty1, value, () => StringProperty1); }

6: }

7: string stringProperty2;

8: public string StringProperty2 {

9: get { return stringProperty2; }

10: set { SetProperty(ref stringProperty2, value, () => StringProperty2); }

11: }

12:

13: string stringProperty3;

14: public string StringProperty3 {

15: get { return stringProperty3; }

16: set {

17: if(SetProperty(ref stringProperty3, value, () => StringProperty3)) {

18: RaisePropertiesChanged(() => StringProperty1, () => StringProperty2);

19: }

20: }

21: }

22: }

Notice we’ve declared lambda expressions returning properties. This approach is very useful because it allows code checking during compilation and easy property renaming. However, in rare cases, application performance is degraded when a property is frequently updated. To accommodate these scenarios, BindableBase provides a static **GetPropertyName** method to calculate property names once from the ViewModel static constructor.

1: public class BindableObject : BindableBase {

2: static string StringProperty1Name = string.Empty;

3: static BindableObject() {

4: BindableObject obj = null;

5: StringProperty1Name = BindableBase.GetPropertyName(() => obj.StringProperty1);

6: }

7:

8: string stringProperty1;

9: public string StringProperty1 {

10: get { return stringProperty1; }

11: set { SetProperty(ref stringProperty1, value, StringProperty1Name); }

12: }

13: }

The BindableBase class only provides basic capabilities for implementing bindable objects, so you’ll likely use **ViewModelBase** as the base class for your base ViewModel. ViewModelBase descends from BindableBase and offers additional capabilities which may prove helpful. For instance, you can initialize ViewModel parameters at design-time. Set properties by overriding the **OnInitializeInDesignMode** method:

1: public class ViewModel : ViewModelBase {

2: string stringProperty1;

3: public string StringProperty1 {

4: get { return stringProperty1; }

5: set { SetProperty(ref stringProperty1, value, () => StringProperty1); }

6: }

7: protected override void OnInitializeInDesignMode() {

8: base.OnInitializeInDesignMode();

9: StringProperty1 = "TestString";

10: }

11: }

In complex applications, you may opt for design-time and runtime ViewModel registration via dependency injection. In simple cases, however, overriding OnInitializeInDesignMode is useful.

ViewModelBase also implements several interfaces: **ISupportParentViewModel**, **ISupportServices**, **ISupportParameter**. These interfaces are used by the service mechanism, to be discussed in upcoming posts.

Thank you for your time. See you!

**OTHER RELATED ARTICLES:**

1. **THIS POST:** Getting Started with DevExpress MVVM Framework. Commands and View Models.
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.

Previously, we examined the basic capabilities of the **DevExpress MVVM Framework** ([Commands and View Models](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)). Today, we will explore more interest things by examining how to perform View-related actions from the View Model layer.

**We start by looking at the simple task: displaying a MessageBox through a View Model.**

The easiest, yet incorrect, way to solve this task is to use the MessageBox.Show method directly from the View Model. Obviously, this approach has a number of serious drawbacks. In fact, the MessageBox is a part of the View, so the use of the MessageBox from the View Model layer breaks the main MVVM rule: the View Model layer should not refer to the View layer. Thus, using this incorrect approach makes it impossible to write unit-tests for View Models.

For solving such tasks in MVVM, the DevExpress MVVM Framework has a special mechanism – **Services**.

Let’s discuss how to solve the posed task with Services. We have the following View Model…

1: public class DocumentViewModel : ViewModelBase {

2: public ICommand CloseDocumentCommand { get; private set; }

3: public DocumentViewModel() {

4: CloseDocumentCommand = new DelegateCommand(OnCloseDocumentCommandExecute);

5: }

6: void OnCloseDocumentCommandExecute() {

7: MessageBoxResult canCloseDocument;

8: //canCloseDocument =

9: // MessageBox.Show("Want to save your changes?", " Document", MessageBoxButton.YesNoCancel);

10: if(canCloseDocument == MessageBoxResult.Yes) {

11: //...

12: }

13: }

14: }

… and the following View:

1: <UserControl x:Class="MessageBoxServiceTest.View.DocumentView"

2: xmlns:ViewModel="clr-namespace:MessageBoxServiceTest.ViewModel"

3: ...>

4: <UserControl.DataContext>

5: <ViewModel:DocumentViewModel/>

6: </UserControl.DataContext>

7: ...

8: <Button Content="Close Document" Command="{Binding CloseDocumentCommand}" .../>

9: ...

10: </UserControl>

The **DevExpress.Xpf.Mvvm** library provides the **IMessageBoxService** interface. The implementation of this interface is contained in the **DevExpress.Xpf.Core** library – the **DXMessageBoxService**. To add this service to our View (DocumentView), add it to the **Interaction.Behaviors** collection as follows.

1: <UserControl xmlns:dx="http://schemas.devexpress.com/winfx/2008/xaml/core"

2: xmlns:dxmvvm=http://schemas.devexpress.com/winfx/2008/xaml/mvvm ...>

3: <UserControl.DataContext>

4: ...

5: </UserControl.DataContext>

6: <dxmvvm:Interaction.Behaviors>

7: <dx:DXMessageBoxService/>

8: </dxmvvm:Interaction.Behaviors>

9: ...

10: </UserControl>

Services are automatically injected to View Models, so they are available from there via an interface that is provided by a certain service.

As you may have noticed, our View Model (DocumentViewModel) is inherited from the ViewModelBase class. So, the DocumentViewModel supports the **GetService<T>** method that returns an interface used to access the DXMessageBoxService.

1: public class DocumentViewModel : ViewModelBase {

2: public ICommand CloseDocumentCommand { get; private set; }

3: public IMessageBoxService MessageBoxService { get { return GetService<IMessageBoxService>(); } }

4: ...

5: void OnCloseDocumentCommandExecute() {

6: MessageBoxResult canCloseDocument = MessageBoxService.Show(

7: messageBoxText: "Want to save your changes?",

8: caption: "Document",

9: button: MessageBoxButton.YesNoCancel);

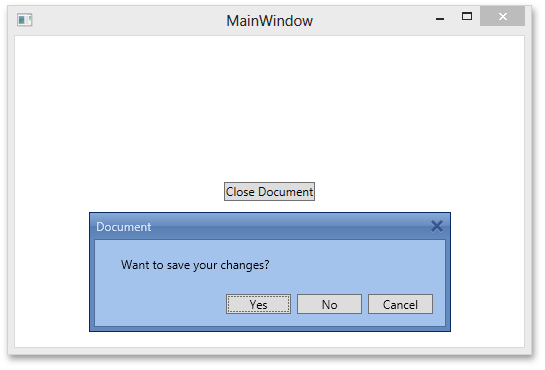
10: if(canCloseDocument == MessageBoxResult.Yes) {

11: //...

12: }

13: }

14: }

**[](http://community.devexpress.com/blogs/wpf/Blog.Services.001_43C62F1E.png)**

**Thus, follow the steps below to use Services.**

1. Define a service in XAML
2. Get access to the service’s interface from your View Model via the GetService<T> method

The use of Services makes it easy to create unit-tests for your View Models. Let’s write a test for the above-mentioned DocumentViewModel ([Moq Framework](http://code.msdn.microsoft.com/windowsdesktop/How-to-Use-Mock-Library-f7af332b) is used).

1: [TestFixture]

2: public class DocumentViewModelTests {

3: [Test]

4: public void Test() {

5: bool serviceIsCalled = false;

6: var viewModel = new DocumentViewModel();

7: var service = new Mock<IMessageBoxService>(MockBehavior.Strict);

8: service.

9: Setup(foo => foo.Show(

10: "Want to save your changes?", "Document", MessageBoxButton.YesNoCancel,

11: MessageBoxImage.None, MessageBoxResult.None)).

12: Returns((string text, string caption, MessageBoxButton button,

13: MessageBoxImage image, MessageBoxResult none) => {

14: serviceIsCalled = true;

15: return MessageBoxResult.OK;

16: });

17: ((ISupportServices)viewModel).ServiceContainer.RegisterService(service.Object);

18: viewModel.CloseDocumentCommand.Execute(null);

19: Assert.IsTrue(serviceIsCalled);

20: }

21: }

***The DXMessageBoxService for Silverlight will be available with the one of minor 13.1 versions.***

**DialogService.**

Now, let’s move to a more complex task. For instance, we need to show a modal window for user registration. This window should contain two buttons: Register and Cancel. The Register button should be disabled if inputted data is invalid. When the window is closed, registration results should be passed to the main View Model. For this task, the DevExpress MVVM Framework provides the **IDialogService** interface and its implementation – **DialogService**.

Suppose that we have a View with the Show Registration Form button.

1: <UserControl x:Class="DialogServiceTest.View.MainView"

2: xmlns:ViewModel="clr-namespace:DialogServiceTest.ViewModel" ...>

3: <UserControl.DataContext>

4: <ViewModel:MainViewModel/>

5: </UserControl.DataContext>

6: ...

7: <Button Content="Show Registration Form" Command="{Binding ShowRegistrationFormCommand}" .../>

8: ...

9: </UserControl>

The button is bound to the MainViewModel command.

1: public class MainViewModel : ViewModelBase {

2: public ICommand ShowRegistrationFormCommand { get; private set; }

3: public MainViewModel() {

4: ShowRegistrationFormCommand = new DelegateCommand(OnShowRegistrationFormCommandExecute);

5: }

6: void OnShowRegistrationFormCommandExecute() {

7: }

8: }

When the ShowRegistrationFormCommand is invoked, we need to show a dialog window that contains another View.

1: <UserControl x:Class="DialogServiceTest.View.RegistrationView" ...>

2: <StackPanel Orientation="Horizontal" ...>

3: <TextBlock Text="User Name: " .../>

4: <TextBox Text="{Binding UserName, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" .../>

5: </StackPanel>

6: </UserControl>

The RegistrationView contains only one editor bound to the RegistrationViewModel. Please note that we do not set the View Model for the RegistrationView directly. In our case, we’ll create a View Model for the RegistrationView at the main View Model level. We’ll return to this mechanism later. If you need to set a design-time View Model for your view, use the d:DataContext property.  The RegistrationViewModel implementation is as follows.

1: public class RegistrationViewModel : ViewModelBase {

2: string userName = string.Empty;

3: public string UserName {

4: get { return userName; }

5: set { SetProperty(ref userName, value, () => UserName); }

6: }

7: }

Next, we implement display a dialog window that contains the RegistrationView. To do this, add the **DialogService** to the main View.

1: <UserControl x:Class="DialogServiceTest.View.MainView"

2: xmlns:View="clr-namespace:DialogServiceTest.View"

3: xmlns:dx="http://schemas.devexpress.com/winfx/2008/xaml/core"

4: xmlns:dxmvvm="http://schemas.devexpress.com/winfx/2008/xaml/mvvm" ...>

5: ...

6: <dxmvvm:Interaction.Behaviors>

7: <dx:DialogService DialogWindowStartupLocation="CenterOwner">

8: <dx:DialogService.ViewTemplate>

9: <DataTemplate>

10: <View:RegistrationView/>

11: </DataTemplate>

12: </dx:DialogService.ViewTemplate>

13: <dx:DialogService.DialogStyle>

14: <Style TargetType="dx:DXDialogWindow">

15: <Setter Property="Width" Value="300"/>

16: <Setter Property="Height" Value="160"/>

17: </Style>

18: </dx:DialogService.DialogStyle>

19: </dx:DialogService>

20: </dxmvvm:Interaction.Behaviors>

21: ...

22: </UserControl>

The DialogService provides several properties for the customization of the dialog window.

1. The ViewTemplate is used for setting a View that will be shown inside the dialog window.
2. The ViewTemplateSelector sets a DataTemplateSelector that returns a certain ViewTemplate by the passed View Model.
3. The DialogStyle sets a Style for the dialog window.
4. The DialogWindowStartupLocation sets window’s startup position. This property is added to the DialogService, because the Window.WindowStartupLocation is not a dependency property and it cannot be set from a Window style.

Next, we’ll use the defined DialogService from our View Model.

1: public class MainViewModel : ViewModelBase {

2: public ICommand ShowRegistrationFormCommand { get; private set; }

3: IDialogService DialogService { get { return GetService<IDialogService>(); } }

4: ...

5: void OnShowRegistrationFormCommandExecute() {

6: RegistrationViewModel registrationViewModel = new RegistrationViewModel();

7: DialogService.ShowDialog(

8: dialogCommands: null,

9: title : "Registration Dialog",

10: viewModel : registrationViewModel

11: );

12: }

13: }

As you can see in the above code snippet, we show the dialog window via the **ShowDialog** method. The IDialogService has several extension methods for different scenarios. For instance, to set standard buttons for the dialog, use the following extension.

1: MessageBoxResult ShowDialog(this IDialogService service,

2: MessageBoxButton dialogButtons, string title, object viewModel);

Our scenario is more complex, so we’ll use another extension that allows setting custom buttons:

1: UICommand ShowDialog(this IDialogService service,

2: IList<UICommand> dialogCommands, string title, object viewModel);

This method takes a collection of UICommand objects as a parameter. In fact, the UICommand is a ready-to-use View Model for a dialog button. The **UICommand** is defined as follows.

1: public class UICommand : BindableBase {

2: public object Caption { get; set; }

3: public ICommand Command { get; set; }

4: public object Id { get; set; }

5: public bool IsCancel { get; set; }

6: public bool IsDefault { get; set; }

7: public object Tag { get; set; }

8: }

Let’s create two UICommands at the MainViewModel level and pass these UICommands to the ShowDialog method.

1: public class MainViewModel : ViewModelBase {

2: ...

3: RegistrationViewModel RegistrationViewModel = null;

4: void OnShowRegistrationFormCommandExecute() {

5: if(RegistrationViewModel == null)

6: RegistrationViewModel = new RegistrationViewModel();

7: UICommand registerCommand = new UICommand() {

8: Caption = "Register",

9: IsCancel = false,

10: IsDefault = true,

11: Command = new DelegateCommand<CancelEventArgs>(

12: x => { },

13: x => !string.IsNullOrEmpty(RegistrationViewModel.UserName)

14: ),

15: };

16: UICommand cancelCommand = new UICommand() {

17: Id = MessageBoxResult.Cancel,

18: Caption = "Cancel",

19: IsCancel = true,

20: IsDefault = false,

21: };

22: UICommand result = DialogService.ShowDialog(

23: dialogCommands: new List<UICommand>() { registerCommand, cancelCommand },

24: title : "Registration Dialog",

25: viewModel : RegistrationViewModel

26: );

27: if(result == registerCommand) {

28: //...

29: }

30: }

31: }

Please note that dialog commands take a **CancelEventArgs** object as a parameter. When dialog commands are invoked, the dialog is closed by default. To prevent this behavior, it is necessary to set the **CancelEventArgs.Cancel** parameter to True. For instance:

1: UICommand dialogCommand = new UICommand() {

2: Command = new DelegateCommand<CancelEventArgs>(OnDialogCommandExecute),

3: };

4: void OnDialogCommandExecute(CancelEventArgs parameter) {

5: parameter.Cancel = true;

6: }

When the registration process is finished and the dialog is closed, the ShowDialog method returns a command, which closed the dialog. After that, you can implement the necessary logic at the main View Model level.

1: void OnShowRegistrationFormCommandExecute() {

2: ...

3: UICommand result = DialogService.ShowDialog(

4: dialogCommands: new List<UICommand>() { registerCommand, cancelCommand },

5: title : "Registration Dialog",

6: viewModel : RegistrationViewModel

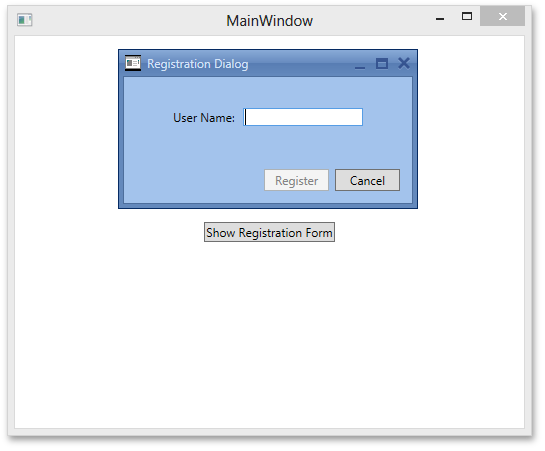
7: );

8: if(result == registerCommand) {

9: //...

10: }

11: }

[](http://community.devexpress.com/blogs/wpf/Blog.Services.002_747936C9.png)

***An asynchronous DialogService version for Silverlight will be available in 13.2.***

*You can find more information about Services in* [*our documentation*](http://documentation.devexpress.com/#WPF/CustomDocument15162)*. This page contains links to Code Examples for each Service.*

This is all for today. Thank you for your time!

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. **THIS POST:** DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService

[Previously](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx), we examined a common Service mechanism and two Service implementations: **DXMessageBoxService** and **DialogService**. In this post, we’ll explore View creation and ViewModel interaction mechanisms.

### View creation mechanisms

There are two main approaches to create Views with Services:

1. ViewTemplate and ViewTemplateSelector.
2. ViewLocator.

**ViewTemplate and ViewTemplateSelector**

In our previous blog post, we created a Registration View and displayed it in a separate dialog window via a DialogService. Our View was defined at the DialogService **ViewTemplate**.

1: <dx:DialogService ...>

2: <dx:DialogService.ViewTemplate>

3: <DataTemplate>

4: <View:RegistrationView/>

5: </DataTemplate>

6: </dx:DialogService.ViewTemplate>

7: ...

8: </dx:DialogService>

The Registration ViewModel was created from the main ViewModel and passed as a parameter to the DialogService.

1: RegistrationViewModel registrationViewModel = new RegistrationViewModel();

2: DialogService.ShowDialog(..., viewModel: registrationViewModel);

Alternatively, a View can be dynamically generated from the passed ViewModel. The **ViewTemplateSelector** supports this approach.

1: public class DialogViewTemplateSelector : DataTemplateSelector {

2: public DataTemplate RegistrationViewTemplate { get; set; }

3: public DataTemplate DefaultViewTemplate { get; set; }

4: public override DataTemplate SelectTemplate(object item, DependencyObject container) {

5: if(item is RegistrationViewModel)

6: return RegistrationViewTemplate;

7: return DefaultViewTemplate;

8: }

9: }

1: <Helper:DialogViewTemplateSelector x:Key="dialogViewTemplateSelector">

2: <Helper:DialogViewTemplateSelector.RegistrationViewTemplate>

3: <DataTemplate>

4: <View:RegistrationView/>

5: </DataTemplate>

6: </Helper:DialogViewTemplateSelector.RegistrationViewTemplate>

7: <Helper:DialogViewTemplateSelector.DefaultViewTemplate>

8: <DataTemplate>

9: <TextBlock Text="Default Template"/>

10: </DataTemplate>

11: </Helper:DialogViewTemplateSelector.DefaultViewTemplate>

12: </Helper:DialogViewTemplateSelector>

13: <dx:DialogService ViewTemplateSelector="{StaticResource dialogViewTemplateSelector}" ...>

14: ...

15: </dx:DialogService>

**ViewLocator**

Alternatively, the ViewLocator provides a simple composition mechanism and also supports creating a View by name.

1: <dxmvvm:Interaction.Behaviors>

2: <dx:DialogService/>

3: </dxmvvm:Interaction.Behaviors>

1: UICommand result = DialogService.ShowDialog(

2: dialogCommands: new List<UICommand>() { registerCommand, cancelCommand },

3: title: "Registration Dialog",

4: documentType: "RegistrationView",

5: viewModel: registrationViewModel

6: );

The preceding code implicitly calls the default ViewLocator to create a View for the document type at the parameter.

The default ViewLocator is limited to creating Views contained in the main application. To create Views from other libraries, register these additional libraries by creating a custom ViewLocator on application startup.

1: ViewLocator viewLocator =

2: new ViewLocator(typeof(MainView).Assembly, typeof(RegistrationView).Assembly);

3: ViewLocator.Default = viewLocator;

If the default ViewLocator implementation does not meet your requirements, you can implement a custom ViewLocator. For instance, the following ViewLocator on-demand library loading:

1: ViewLocator.Default = new MainViewLocator();

2:

3: public class MainViewLocator : IViewLocator {

4: public object ResolveView(string name) {

5: if(name == "RegistrationView") {

6: Assembly assembly = Assembly.LoadFrom(@"DialogServiceTestLib.dll");

7: Type t = assembly.GetType("DialogServiceTest.View.RegistrationView");

8: return Activator.CreateInstance(t);

9: }

10: return null;

11: }

12: }

You may also set a particular ViewLocator at the Service level. Any custom Service can support View creation by inheriting from the **ViewServiceBase** class and exposing its ViewLocator property.

### Interaction of ViewModels

We’ve explored the mechanisms of View creation. Since Views are generally unaware of each other, interaction is mediated at the ViewModel level. ViewModel interactions are implemented by either of the following approaches:

1. Closely-coupled ViewModels.
2. Loosely-coupled ViewModels.

Closely-coupling implies ViewModel creation by other ViewModels. Loose-coupling implies ViewModels bind to Views while ViewModel interaction is handled with parameter passing. Both strategies can be used simultaneously.

We’ll examine these interaction patterns in detail.

**Closely-coupled ViewModels**

The previous DialogService example is built entirely on this interaction type. The Registration ViewModel is created in the main ViewModel code which has access to the properties on the child ViewModel.

1: RegistrationViewModel registrationViewModel = new RegistrationViewModel();

2: DialogService.ShowDialog(..., viewModel : registrationViewModel);

**Loosely-coupled ViewModels**

The **ViewModelBase** is a basic ViewModel class supporting two interfaces:

1: public interface ISupportParameter {

2: object Parameter { get; set; }

3: }

4: public interface ISupportParentViewModel {

5: object ParentViewModel { get; set; }

6: }

**ISupportParameter** passes parameters from the main ViewModel to the child ViewModel. There’s no restriction to what can serve as a parameter. For instance, a ViewModel editing a table row can be passed a row or key as a parameter.

**ISupportParentViewModel** makes it possible to access Services of the main ViewModel from the child ViewModel. I.e., all services defined by the main View are available in the child ViewModel.

With a ViewModel defined in XAML…

1: <dx:DialogService ...>

2: <dx:DialogService.ViewTemplate>

3: <DataTemplate>

4: <View:RegistrationView>

5: <View:RegistrationView.DataContext>

6: <ViewModel:RegistrationViewModel/>

7: </View:RegistrationView.DataContext>

8: </View:RegistrationView>

9: </DataTemplate>

10: </dx:DialogService.ViewTemplate>

11: ...

12: </dx:DialogService>

…call a Service method to pass the parameter and parent ViewModel:

1: DialogService.ShowDialog(

2: ...

3: parameter: "Parameter",

4: parentViewModel: this,

5: );

In this case, the DialogService picks up the ViewModel from the DataContext of the generated View and sets the Parameter and ParentViewModel properties on this ViewModel.

**Messenger**

Messenger also implements ViewModel interaction. This class exchanges messages between loosely-coupled parts of the application. We’ll discuss this pattern in detail in a future article.

### IDocumentManagerService

**IDocumentManagerService** is an abstraction of the document manager. With version 13.1, we introduced two implementations of this service: WindowedDocumentUIService and TabbedDocumentUIService.

**WindowedDocumentUIService**

WindowedDocumentUIService supports the display and control of Views in separate windows. Consider the simple sample. We have two Views -- **Document1View** and **Document2View** -- corresponding to ViewModels that are assigned in XAML (via DataContext).

The main View presents two buttons to open each document (View).

1: <UserControl x:Class="DocumentServiceTest.View.MainView"

2: xmlns:ViewModel="clr-namespace:DocumentServiceTest.ViewModel"

3: ...>

4: <UserControl.DataContext>

5: <ViewModel:MainViewModel/>

6: </UserControl.DataContext>

7: <StackPanel Orientation="Horizontal">

8: <Button Content="Show Document1 Window"

9: Command="{Binding ShowDocumentCommand}" CommandParameter="Document1View" />

10: <Button Content="Show Document3 Window"

11: Command="{Binding ShowDocumentCommand}" CommandParameter="Document2View" />

12: </StackPanel>

13: </UserControl>

1: public class MainViewModel : ViewModelBase {

2: public ICommand ShowDocumentCommand { get; private set; }

3: public MainViewModel() {

4: ShowDocumentCommand = new DelegateCommand<string>(OnShowDocumentCommandExecute);

5: }

6: void OnShowDocumentCommandExecute(string document) {

7: }

8: }

The main application window contains the main view.

1: <Window x:Class="DocumentServiceTest.WindowedDocumentUIServiceWindow"

2: xmlns:View="clr-namespace:DocumentServiceTest.View"

3: ...>

4: <Grid>

5: <View:MainView ...>

6: </View:MainView>

7: </Grid>

8: </Window>

To display the Document1View and Document2View from the main ViewModel, we will use the WindowedDocumentUIService. Add this service to the main view at the main window level.

1: <Window x:Class="DocumentServiceTest.WindowedDocumentUIServiceWindow"

2: xmlns:View="clr-namespace:DocumentServiceTest.View"

3: xmlns:dx="http://schemas.devexpress.com/winfx/2008/xaml/core"

4: xmlns:dxmvvm="http://schemas.devexpress.com/winfx/2008/xaml/mvvm"

5: ...>

6: <Grid>

7: <View:MainView>

8: <dxmvvm:Interaction.Behaviors>

9: <dx:WindowedDocumentUIService>

10: <dx:WindowedDocumentUIService.WindowStyle>

11: <Style TargetType="Window">

12: <Setter Property="Height" Value="300"/>

13: <Setter Property="Width" Value="400"/>

14: </Style>

15: </dx:WindowedDocumentUIService.WindowStyle>

16: </dx:WindowedDocumentUIService>

17: </dxmvvm:Interaction.Behaviors>

18: </View:MainView>

19: </Grid>

20: </Window>

The service is now available from the main ViewModel. We’ll use it to display the child views.

1: public class MainViewModel : ViewModelBase {

2: public ICommand ShowDocumentCommand { get; private set; }

3: IDocumentManagerService DocumentManager {

4: get { return GetService<IDocumentManagerService>(); }

5: }

6: public MainViewModel() {

7: ShowDocumentCommand = new DelegateCommand<string>(OnShowDocumentCommandExecute);

8: }

9: void OnShowDocumentCommandExecute(string document) {

10: IDocument doc = DocumentManager.CreateDocument(document, null, this);

11: doc.DestroyOnClose = true;

12: doc.Title = document;

13: doc.Show();

14: }

15: }

In the preceding code, we created a document via the service and call the document’s Show method. Our document supports the **IDocument** interface and provides access to a container with the appropriate View. While using the WindowedDocumentUIService, this container is a Window. IDocument is defined in the DevExpress.Xpf.Mvvm library as follows:

1: public interface IDocument {

2: void Show();

3: void Close(bool force = true);

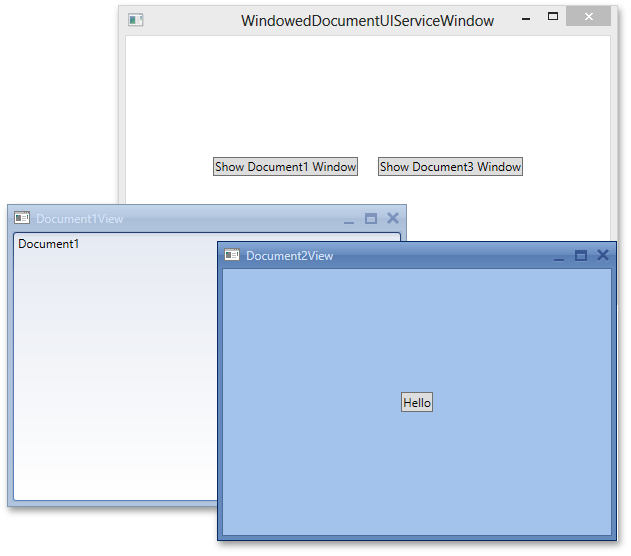
4: bool DestroyOnClose { get; set; }

5: object Title { get; set; }

6: object Content { get; }

7: }

We have the following result after running the program.

[](http://community.devexpress.com/blogs/wpf/3.Blog.Services.001_66720096.png)

**TabbedDocumentUIService**

It’s a cool thing that the IDocumentManagerService allows switching UI styles while leaving the View and ViewModel code unchanged.

Let’s modify our application to use the Tabbed MDI UI. All our changes are at the main window code.

1: <Window x:Class="DocumentServiceTest.TabbedDocumentUIServiceWindow"

2: xmlns:View="clr-namespace:DocumentServiceTest.View"

3: xmlns:dx="http://schemas.devexpress.com/winfx/2008/xaml/core"

4: xmlns:dxmvvm="http://schemas.devexpress.com/winfx/2008/xaml/mvvm"

5: xmlns:dxdo="http://schemas.devexpress.com/winfx/2008/xaml/docking"

6: ...>

7: <Grid>

8: <dxdo:DockLayoutManager>

9: <dxdo:LayoutGroup Orientation="Vertical">

10: <dxdo:LayoutPanel Caption="Navigation" ItemHeight="Auto">

11: <View:MainView>

12: <dxmvvm:Interaction.Behaviors>

13: <dxdo:TabbedDocumentUIService

14: DocumentGroup="{Binding ElementName=documnetGroup}"/>

15: </dxmvvm:Interaction.Behaviors>

16: </View:MainView>

17: </dxdo:LayoutPanel>

18: <dxdo:DocumentGroup x:Name="documnetGroup"

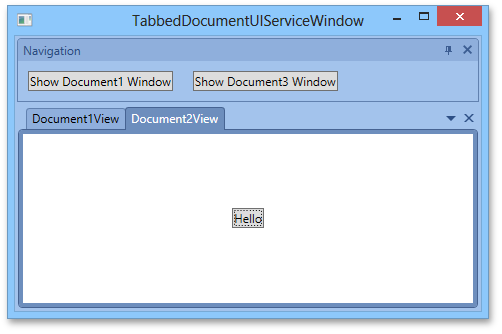
19: Caption="Documents" ItemHeight="\*"/>

20: </dxdo:LayoutGroup>

21: </dxdo:DockLayoutManager>

22: </Grid>

23: </Window>

[](http://community.devexpress.com/blogs/wpf/3.Blog.Services.002_186C31D2.png)

**CODE EXAMPLE:** Download an example of these two services available from the DevExpress documentation: <http://documentation.devexpress.com/#WPF/CustomDocument15745>.

In our upcoming 13.2, we’ll introduce the new IDocumentManagerService implementation for building a modern Windows 8 UI.

Thank you for your time. Feel free to sound off with your thoughts in the comments below.

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. **THIS POST:** DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Introduction to POCO ViewModels.

[[](https://community.devexpress.com/blogs/wpf/v4_51E2B016.png)](https://community.devexpress.com/blogs/wpf/v4_51E2B016.png)Traditionally, MVVM development means writing significant ViewModel boilerplate for bindable properties and commands. Even after extending the **DevExpress.Xpf.Mvvm.ViewModelBase** class, you need at least five lines of code to declare a single bindable property and several more to define a command:

***NOTE:*** Refer to the following topic to explore the ViewModelBase class: [*Getting Started with DevExpress MVVM Framework. Commands and View Models.*](https://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)

1: public class LoginViewModel : ViewModelBase {

2: string userName;

3: public string UserName {

4: get { return userName; }

5: set { SetProperty(

6: ref userName, value, () => UserName);

7: }

8: }

9: public DelegateCommand<string>

10: SaveAccountCommand { get; private set; }

11:

12: public LoginViewModel() {

13: SaveAccountCommand =

14: new DelegateCommand<string>(

15: SaveAccount, CanSaveAccount);

16: }

17: void SaveAccount(string fileName) {

18: //...

19: }

20: bool CanSaveAccount(string fileName) {

21: return !string.IsNullOrEmpty(fileName);

22: }

23: }

With many bindable properties declared, it becomes a real nightmare if an error is raised in a property setter: as when passing the incorrect field to a setter or setting the incorrect property from a lambda expression. Worse, it is just not beautiful.

Now imagine, instead of the preceding code, writing this:

1: public class LoginViewModel {

2: public virtual string UserName { get; set; }

3: public void SaveAccount(string fileName) {

4: //...

5: }

6: public bool CanSaveAccount(string fileName) {

7: return true;

8: }

9: }

The 13.2 release makes it possible, with support for [POCO](http://en.wikipedia.org/wiki/Plain_Old_CLR_Object) ViewModels. Generate a full-fledged ViewModel from your POCO class with **ViewModelSource.**

In code:

1: public class LoginViewModel {

2: protected LoginViewModel() { }

3: public static LoginViewModel Create() {

4: return ViewModelSource.Create(() => new LoginViewModel());

5: }

6:

7: public virtual string UserName { get; set; }

8: public void SaveAccount(string fileName) {

9: //...

10: }

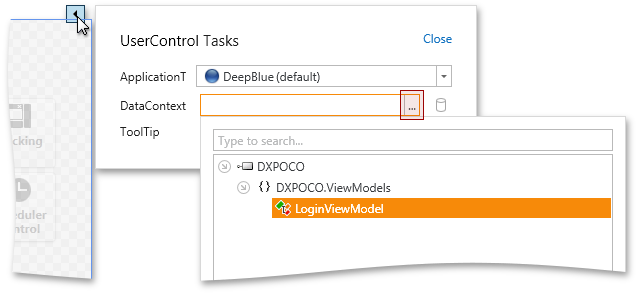
11: public bool CanSaveAccount(string fileName) {

12: return !string.IsNullOrEmpty(fileName);

13: }

14: }

In XAML:

[](https://community.devexpress.com/blogs/wpf/4.Blog.Messaging.003_727874A1.png)

1: <UserControl x:Class="DXPOCO.Views.LoginView"

2: xmlns:dxmvvm="http://schemas.devexpress.com/winfx/2008/xaml/mvvm"

3: xmlns:ViewModels="clr-namespace:DXPOCO.ViewModels"

4: DataContext="{dxmvvm:ViewModelSource Type=ViewModels:LoginViewModel}"

5: ...>

6: <Grid>

7: <!--...-->

8: </Grid>

9: </UserControl>

ViewModelSource uses **System.Reflection.Emit** to dynamically create and return an instance of a descendant of the passed ViewModel. The approximate code of the descendant is:

1: public class LoginViewModelBindable : LoginViewModel, INotifyPropertyChanged {

2: public override string UserName {

3: get { return base.UserName; }

4: set {

5: if(base.UserName == value) return;

6: base.UserName = value;

7: RaisePropertyChanged("UserName");

8: }

9: }

10: DelegateCommand<string> saveAccountCommand;

11: public DelegateCommand<string> SaveAccountCommand {

12: get {

13: return saveAccountCommand ??

14: (saveAccountCommand =

15: new DelegateCommand<string>(SaveAccount, CanSaveAccount));

16: }

17: }

18:

19: //INotifyPropertyChanged Implementation

20: }

Now let’s examine how to control ViewModel generation.

###### Bindable Properties

The rules for generating bindable properties are simple: ViewModelSouce creates bindable properties for public virtual properties and automatic properties with a public getter and, at the least, a protected setter.

You can define functions to invoke when a property is changed. These functions need a special name: **On<PropertyName>Changed**, **On<PropertyName>Changing**:

1: public class LoginViewModel {

2: public virtual string UserName { get; set; }

3: protected void OnUserNameChanged() {

4: //...

5: }

6: }

1: public class LoginViewModel {

2: public virtual string UserName { get; set; }

3: protected void OnUserNameChanged(string oldValue) {

4: //...

5: }

6: protected void OnUserNameChanging(string newValue) {

7: //...

8: }

9: }

Use the **BindableProperty** attribute to set a function not matching the convention:

1: public class LoginViewModel {

2: [BindableProperty(isBindable: false)]

3: public virtual bool IsEnabled { get; set; }

4:

5: [BindableProperty(OnPropertyChangedMethodName = "Update")]

6: public virtual string UserName { get; set; }

7: protected void Update() {

8: //...

9: }

10: }

The disadvantage of this approach is renaming a function or property causes its handler to stop working. This is why we also implemented support for fluent declaration. Use the [Fluent API](http://en.wikipedia.org/wiki/Fluent_interface) as follows:

1: [MetadataType(typeof(Metadata))]

2: public class LoginViewModel {

3: class Metadata : IMetadataProvider<LoginViewModel> {

4: void IMetadataProvider<LoginViewModel>.BuildMetadata

5: (MetadataBuilder<LoginViewModel> builder) {

6:

7: builder.Property(x => x.UserName).

8: OnPropertyChangedCall(x => x.Update());

9: builder.Property(x => x.IsEnabled).

10: DoNotMakeBindable();

11: }

12: }

13: public virtual bool IsEnabled { get; set; }

14: public virtual string UserName { get; set; }

15: protected void Update() {

16: //...

17: }

18: }

This approach avoids errors during refactoring.

***NOTE:*** We will return to the Fluent API in an upcoming post to examine other tasks to solve with it.

###### Commands

A command is generated for each parameterless and single parameter public method.

1: public class LoginViewModel {

2: //DelegateCommand<string> SaveAccountCommand =

3: // new DelegateCommand<string>(SaveAccount, CanSaveAccount);

4: public void SaveAccount(string fileName) {

5: //...

6: }

7: public bool CanSaveAccount(string fileName) {

8: return !string.IsNullOrEmpty(fileName);

9: }

10:

11: //DelegateCommand Close = new DelegateCommand(Close);

12: public void Close() {

13: //...

14: }

15: }

Likewise, command generation can be controlled with the **Command** attribute or Fluent API:

1: public class LoginViewModel {

2: [Command(isCommand: false)]

3: public void SaveCore() {

4: //...

5: }

6:

7: [Command(CanExecuteMethodName = "CanSaveAccount",

8: Name = "SaveCommand",

9: UseCommandManager = true)]

10: public void SaveAccount(string fileName) {

11: //...

12: }

13: public bool CanSaveAccount(string fileName) {

14: return !string.IsNullOrEmpty(fileName);

15: }

16: }

1: [MetadataType(typeof(Metadata))]

2: public class LoginViewModel {

3: class Metadata : IMetadataProvider<LoginViewModel> {

4: void IMetadataProvider<LoginViewModel>.BuildMetadata

5: (MetadataBuilder<LoginViewModel> builder) {

6:

7: builder.CommandFromMethod(x => x.SaveCore()).

8: DoNotCreateCommand();

9: builder.CommandFromMethod(x => x.SaveAccount(default(string))).

10: CommandName("SaveCommand").

11: CanExecuteMethod(x => x.CanSaveAccount(default(string)));

12: }

13: }

14:

15: public void SaveCore() {

16: //...

17: }

18:

19: public void SaveAccount(string fileName) {

20: //...

21: }

22: public bool CanSaveAccount(string fileName) {

23: return !string.IsNullOrEmpty(fileName);

24: }

25: }

The extension methods of the **DevExpress.Xpf.Mvvm.POCO.POCOViewModelExtensions** class support manually raising a PropertyChanged event or updating a command:

1: public static class POCOViewModelExtensions {

2: public static bool IsInDesignMode(this object viewModel);

3: public static void RaiseCanExecuteChanged<T>(

4: this T viewModel, Expression<Action<T>> methodExpression);

5: public static void RaisePropertyChanged<T, TProperty>(

6: this T viewModel, Expression<Func<T, TProperty>> propertyExpression);

7: }

For instance:

1: public class LoginViewModel {

2: public void Update() {

3: this.RaisePropertyChanged(x => x.UserName);

4: this.RaiseCanExecuteChanged(x => x.SaveAccount(default(string)));

5: }

6:

7: public virtual string UserName { get; set; }

8: public void SaveAccount(string fileName) {

9: //...

10: }

11: public bool CanSaveAccount(string fileName) {

12: return !string.IsNullOrEmpty(fileName);

13: }

14: }

###### Services

As you likely know, the DevExpress MVVM Framework provides a [service mechanism](https://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx). Previously, accessing a service required inheriting from ViewModelBase and implementing the following construction:

1: public IMessageBoxService MessageBoxService {

2: get { return GetService<IMessageBoxService>(); }

3: }

You can now write:

1: public virtual IMessageBoxService MessageBoxService { get { return null; } }

Use the **ServiceProperty** attribute or the Fluent API, as earlier described, to control service property generation.

###### Is there a performance penalty for working with POCO?

The ViewModel created by ViewModelSource is a descendant of the passed class. This descendant is generated using System.Reflection.Emit to implement INotifyPropertyChanged, override virtual properties, create commands, etc. Although these operations are handled at runtime, performance degradation is not an issue because the mechanism uses a cache -- generation is performed once only for each class and not for each instance of a class. By the way, Entity Framework 5.0+ uses the same mechanism.

ViewModelSource supports several approaches to create ViewModels.

1. For a ViewModel with a public parameterless constructor, the following approach is simple and fast:

ViewModelSource.Create<LoginViewModel>();

1. Since lambda expressions are not comparable, they remain uncached and compiled anew with each method call. While this approach is the most powerful and beautiful, it is also the slowest:

ViewModelSource.Create(() => new LoginViewModel(caption: "Login") {  
 UserName = "John Smith"  
});

1. Since compiled delegate instances can be cached, this is a fast approach for passing parameters to the ViewModel constructor:

var factory = ViewModelSource.Factory((string caption) => new LoginViewModel(caption));  
factory("Login");

###### Is there a live example of POCO ViewModels?

With 13.2, we introduce a new real-life demo – Sales. This demo is built completely on the POCO technology, so you can examine its code.

Moreover, DevExpress Scaffolding Wizards now generate ViewModels as POCO. [Here](http://documentation.devexpress.com/#WPF/CustomDocument15282) is a tutorial describing how to build an application with Scaffolding Wizards. You can also scaffold Views based on your POCO ViewModels - simply add the **DevExpress.Xpf.Mvvm.DataAnnotations.POCOViewModel** attribute to the ViewModels.

P.S. What could be better than the previous approach? Right, dropping the requirements of virtual properties and creating ViewModel instances via a special factory. We are currently working on this and will introduce a solution in the near future.

P.P.S. In fact, the POCO ViewModels functionality is a good example of the [Aspect-oriented Programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming) paradigm. Previously, inheriting from ViewModelBase meant implementing [cross-cutting concerns](http://en.wikipedia.org/wiki/Cross-cutting_concern) across ViewModels (i.e. similar code in the definitions of the bindable properties and commands). Now, you can get rid of this redundant code with aspects (e.g., conventions for defining properties and methods, attributes, and Fluent API).

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. **THIS POST:** DevExpress MVVM Framework. Introduction to POCO ViewModels.
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Interaction of ViewModels. Messenger.

An application’s architecture depends on the degree of connection between its modules. Loosely-coupled systems are suited for large applications. This usually means many scattered modules, operating without awareness of each other. Ideally, modules are the building blocks of an adaptive design. Loosely-coupled architectures are easy to support and improve because adding or removing functionality simply means registering or unregistering a specific module without concern towards the others.

To facilitate interaction between modules, we implemented aclass to exchange messages regardless of which module is sending or receiving the message. This class is **Messenger.**

Let’s examine a simple sample to get a clearer understanding.

Imagine we have a database which can be modified from several modules. One module will need to be notified when modifications happen (e.g., adding, removing, and changing a record). We’ll first need to create a message:

1: public enum MessageType { Added, Deleted, Changed }

2: public class Message {

3: public MessageType MessageType { get; private set; }

4: public object RecordID { get; private set; }

5: public Message(object recordID, MessageType messageType) {

6: RecordID = recordID;

7: MessageType = messageType;

8: }

9: }

We can then subscribe to the message from anywhere in the application. For instance:

1: public class Module1 {

2: public Module1() {

3: Messenger.Default.Register<Message>(this, OnMessage);

4: }

5: void OnMessage(Message message) {

6: switch(message.MessageType) {

7: case MessageType.Added:

8: //...

9: break;

10: case MessageType.Changed:

11: //...

12: break;

13: case MessageType.Deleted:

14: //...

15: break;

16: default:

17: throw new NotImplementedException();

18: }

19: }

20: }

Sending a message is even easier:

1: public class Module2 {

2: void SendMessage() {

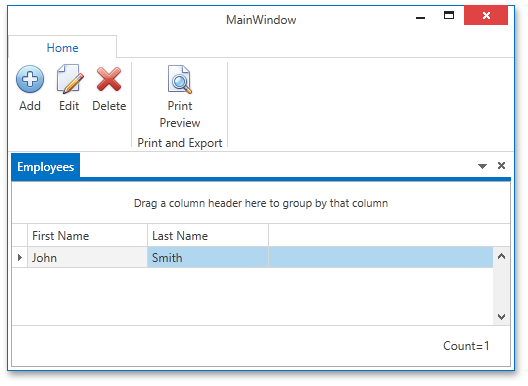
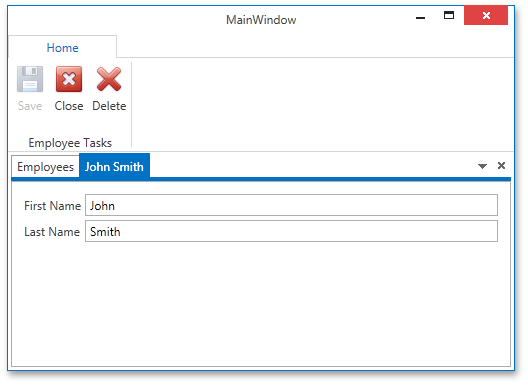
3: Messenger.Default.Send(new Message(0, MessageType.Added));

4: }

5: }

As you can see, this approach implements module interaction without reference to a module’s code. Even if you remove Module1 or Module2 while developing the application, it will not cause errors and the entire system will continue to function.

We also prepared a real example with this architecture. It can be found [here](https://www.devexpress.com/Support/Center/Example/Details/E5001). Screenshots of the example are below.

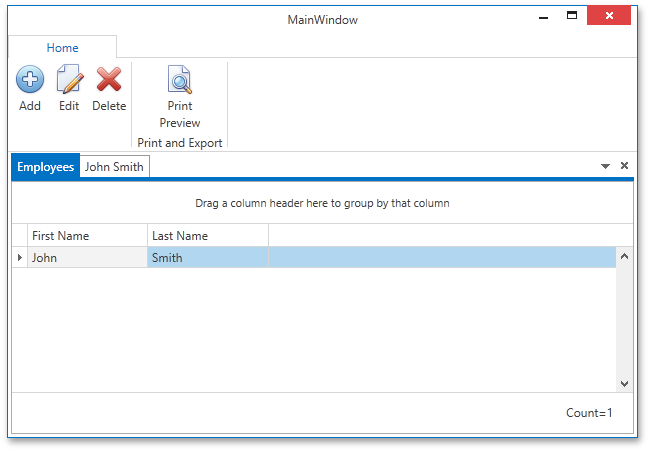
[](https://community.devexpress.com/blogs/wpf/4.Blog.Messaging.001_2A0B907F.png)[](https://community.devexpress.com/blogs/wpf/4.Blog.Messaging.002_4B211DCA.png)

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. **THIS POST:** DevExpress MVVM Framework. Interaction of ViewModels. Messenger.
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

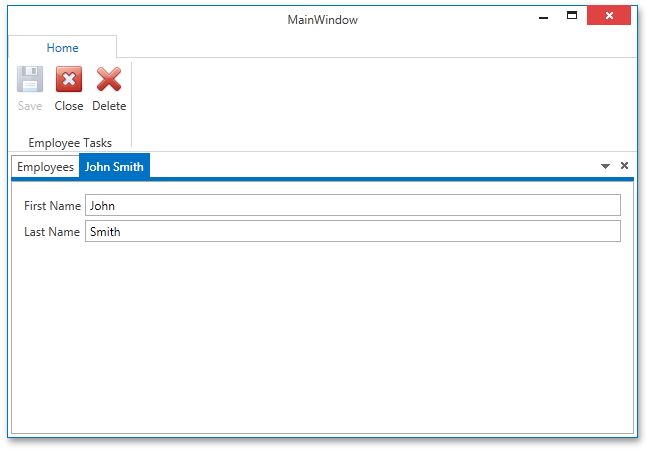
## DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.

Today, we examine using ViewModels to automatically generate Views. Our application looks as follows.

[](https://community.devexpress.com/blogs/wpf/001_2D37ACAF.png)

When the application starts, the MainWindow shows the GridControl. The Ribbon provides some basic functionality to *Add*, *Remove*, or *Modify* a record, or *Print* the view.

Clicking *Add* or *Modify* opens a View in the detail tab.

[](https://community.devexpress.com/blogs/wpf/002_13D23A26.png)

While the second View is open, the RibbonControl displays its buttons: *Save*, *Close*, and *Delete*. Each save operation is specific to its tab. Changes are committed separately in each of the available tabs.

This simple layout can be automatically generated from ViewModels. You can try this yourself from the sample we’ve prepared. Download and open it from Code Example: [E5001](https://www.devexpress.com/Support/Center/Example/Details/E5001).

Notice the provided code of the sample for the model and ViewModels.

**The model layer is three classes:** *Employee*, *EmployeeContext*, and *EmployeeContextInitializer*.

*Employee* is a database entity. [**Entity Framework Code First**](http://msdn.microsoft.com/en-us/data/jj193542.aspx) creates the database during the application’s initial load.

*EmployeeContext* operates on the database with EF Code First.

*EmployeeContextInitializer* populates the database with predefined records.

**There are two ViewModels.**

The first is for the Employee table – *EmployeeCollectionViewModel*. Since it is a [**POCO**](http://documentation.devexpress.com/#WPF/CustomDocument16454) class, its public and virtual properties are bindable and its public methods are available as commands. This ViewModel provides an Employees property for the database records. The available commands add, remove, or modify a record.

*EmployeeViewModel* is also a POCO class. A value for the *Parameter* property is necessary for each *EmployeeViewModel* instance. The parameter describes how the ViewModel operates on records: by creating a new record or modifying an existing one. The *Save* command commits changes. *EmployeeViewModel* also contains commands to remove a record or to close the corresponding View.

ViewModel to ViewModel interactions are mediated by **Messenger**. A message is sent when changes are saved in *EmployeeViewModel*. *EmployeeCollectionViewModel* catches this message and updates its View.

That’s it. Now, let’s generate the layout. Remove all the Views and the MainViewModel to follow the steps demonstrated in the video below.

[Video](http://youtu.be/0WUSdGtqUSw)

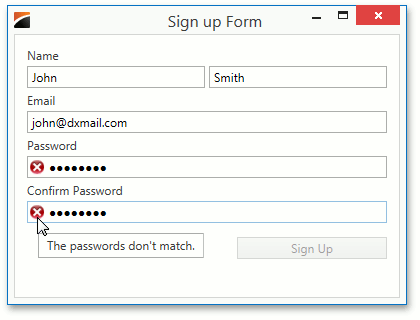
**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. **THIS POST:** DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.

The IDataErrorInfo interface is the standard mechanism for data validation in WPF and Silverlight. IDataErrorInfo can provide validation rules for each property in isolation or, alternatively, on the entire object. This post presents three distinct IDataErrorInfo implementations.

As an illustrative example, I will use the simple registration form below:

[](https://community.devexpress.com/blogs/wpf/initial_256color_21D12E2C.png)

A complete sample is available from <http://www.devexpress.com/example=E5151>.

Version 13.2.8 extends our POCO to support an automatic IDataErrorInfo implementation. See [our post](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx) for additional information on POCO View Model support.

### Attributes

Attributes are the most straightforward IDataErrorInfo validation approach. Attributes for specifying validation constraints including Required, Range and CustomValidation are available in the System.ComponentModel.DataAnnotations andDevExpress.Mvvm.Native namespaces.

The sign up form ViewModel with validation attributes is as follows:

[POCOViewModel(ImplementIDataErrorInfo = true)]   
public class SignupViewModel {   
 [Required(ErrorMessage = "Please enter the first name.")]   
 public virtual string FirstName { get; set; }   
 [Required(ErrorMessage = "Please enter the last name.")]   
 public virtual string LastName { get; set; }   
 [EmailAddress]   
 public virtual string Email { get; set; }   
 [Required(ErrorMessage = "Please enter the password.")]   
 [MinLength(8, ErrorMessage = "The password must be at least 8 characters long.")]   
 [MaxLength(20, ErrorMessage = "The password must not exceed the length of 20.")]   
 [CustomValidation(typeof(SignupViewModel), "CheckPassword")]   
 public virtual string Password { get; set; }   
 [Required(ErrorMessage = "Please confirm the password.")]   
 [MinLength(8, ErrorMessage = "The password must be at least 8 characters long.")]   
 [MaxLength(20, ErrorMessage = "The password must not exceed the length of 20.")]   
 [CustomValidation(typeof(SignupViewModel), "CheckPassword")]   
 public virtual string ConfirmPassword { get; set; }   
 public static ValidationResult CheckPassword(object value, ValidationContext context) { ... }   
}

**NOTE:** The MinLength, MaxLength and EmailAddress attributes became available in .NET 4.5.

The POCO will implement the IDataErrorInfo interface for you. Make note of the POCOViewModel attribute and its ImplementIDataErrorInfo parameter, which need to be explicitly specified.

### Fluent API

Although attributes are convenient in simple scenarios, they quickly become awkward with custom validation logic. The validating method name (CheckPassword in our case) is passed to the attribute as a string, which makes refactoring less pleasant and more error prone.

The problem is even more pronounced when doing localization. Compare passing an error message through an attribute

[Required(ErrorMessageResourceName = "PleaseConfirmPasswordError",  
 ErrorMessageResourceType = typeof(Resources))]  
public virtual string ConfirmPassword { get; set; }

to the same done via the Fluent API

builder.Property(x => x.ConfirmPassword)  
 .Required(() => Resources.PleaseConfirmPasswordError);

The Fluent API allows specifying metadata properties and constraints while preserving compile-time checks. The previous ViewModel can be rewritten as the following:

[POCOViewModel(ImplementIDataErrorInfo = true)]   
public class SignupViewModel : ViewModelBase {   
 static PropertyMetadataBuilder<SignupViewModel, string> AddPasswordCheck(  
 PropertyMetadataBuilder<SignupViewModel, string> builder) {   
 return builder.MatchesInstanceRule(vm => vm.Password == vm.ConfirmPassword,  
 () => "The passwords don't match.")   
 .MinLength(8, () => "The password must be at least 8 characters long.")   
 .MaxLength(20, () => "The password must not exceed the length of 20.");   
 }   
 public static void BuildMetadata(MetadataBuilder<SignupViewModel> builder) {   
 builder.Property(x => x.FirstName)   
 .Required(() => "Please enter the first name.");   
 builder.Property(x => x.LastName)   
 .Required(() => "Please enter the last name.");  
 builder.Property(x => x.Email)   
 .EmailAddressDataType(() => "Please enter a correct email address.");   
 AddPasswordCheck(builder.Property(x => x.Password))   
 .Required(() => "Please enter the password.");   
 AddPasswordCheck(builder.Property(x => x.ConfirmPassword))   
 .Required(() => "Please confirm the password.");   
 }  
 public virtual string FirstName { get; set; }   
 public virtual string LastName { get; set; }   
 public virtual string Email { get; set; }   
 public virtual string Password { get; set; }   
 public virtual string ConfirmPassword { get; set; }   
}

Now, when you change a property name the compiler will require updating the metadata.

### Custom implementation

If you need greater control over validation or you can’t use POCO View Models for some reason, you can use the following code generated by the POCO as a basis for your own implementation.

public class SignupViewModel : IDataErrorInfo {   
 ...   
 string IDataErrorInfo.Error {   
 get { return string.Empty; }   
 }   
 string IDataErrorInfo.this[string columnName] {   
 get { return IDataErrorInfoHelper.GetErrorText(this, columnName); }   
 }   
}

The default Error implementation here returns an empty string. Change it to provide a custom error message.

### Consuming the ViewModel

To use the resulting ViewModel, we need a View to be aware of our IDataErrorInfo implementation. Out-of-the-box, DevExpress controls support data validation and visual notifications when a user encounters validation errors. Our registration form example illustrates validation in [Data Editors](https://documentation.devexpress.com/#WPF/CustomDocument6190), but the [Data Grid](https://documentation.devexpress.com/#WPF/CustomDocument6084) or [Property Grid](https://documentation.devexpress.com/#WPF/CustomDocument15640) would do equally well – all support IDataErrorInfo validation. The View is the following:

<dx:DXWindow x:Class="DataValidationSample.MainWindow"   
 ...  
 DataContext="{dxmvvm:ViewModelSource Type=local:SignupViewModel}">  
 <dxlayout:LayoutControl dxe:ValidationService.IsValidationContainer="True" x:Name="validationContainer">  
 ...  
 <dxlayout:LayoutItem Label="Email" ...>  
 <dxe:TextEdit Text="{Binding Email, ValidatesOnDataErrors=True,   
 UpdateSourceTrigger=PropertyChanged}"/>  
 </dxlayout:LayoutItem>  
 ...  
 <Button ... IsEnabled="{Binding Path=(dxe:ValidationService.HasValidationError),  
 ElementName=validationContainer, Converter={dx:NegationConverter}}"/>  
 </dxlayout:LayoutControl>  
</dx:DXWindow>

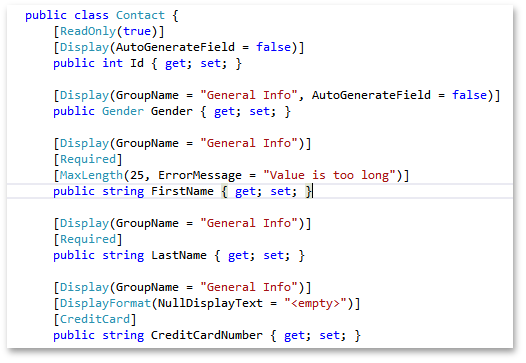
That is all that’s necessary. Feel free to comment if you have any questions or suggestions.

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](http://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/24/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. THIS POST: DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.
8. [DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.](https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx)

## DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.

Imagine configuring editing and validation settings from one place – eliminating the need to separately configure each data view (XAML). DataAnnotation attributes allow you to do exactly that.   
Beginning with version 13.2, a consistent set of functionalities were implemented across all our controls. Originally, GridControl provided limited support for attributes and cell editors could not be configured by attributes specifying an editor data type. Bands could not be set at the data source level. Moreover, DataAnnotation attributes did not exist for PropertyGridControl. Now, all of our components provide similar functionality at design time, runtime, and the Scaffolding Wizard. We also introduced a flexible alternative in version 13.2 – the DevExpress Fluent API.



### Which attributes are supported?

DevExpress editors, GridControl, DataLayoutControl, Scaffolding Wizards and other components support nearly every standard attribute in the [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace.

We also implemented a number of our own attributes for defining masks. These attributes are available from the [DevExpress.Xpf.Mvvm.DataAnnotations](https://documentation.devexpress.com/#WPF/DevExpressXpfMvvmDataAnnotations) namespace.

To learn more about the available attributes, see the following help topic: [Data Annotation Attributes](https://documentation.devexpress.com/#WPF/CustomDocument16863).

### Common decoration patterns

#### In a model class

The simplest way to assign an attribute is to explicitly declare it on a property of a model class:

public class Point {  
 [Display(Name = "Abscissa")]  
 public double X { get; set; }  
 [Display(Name = "Ordinate")]  
 public double Y { get; set; }  
}

Although this approach is simple and descriptive, your code may become difficult to read when there are many properties. You may also find your model class does not allow attribute decorations. For example, a WCF Data Service or the Entity Framework Database First approach may automatically generate model classes.

#### In a metadata class

Declare a metadata class and assign it to a model via the [MetadataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.metadatatypeattribute%28v=vs.110%29.aspx) attribute:

[MetadataType(typeof(PointMetadata))]  
public class Point {  
 public double X { get; set; }  
 public double Y { get; set; }  
}  
public class PointMetadata {  
 [Display(Name = "Abscissa")]  
 public double X { get; set; }  
 [Display(Name = "Ordinate")]  
 public double Y { get; set; }  
}

You can extend this approach to automatically generated classes using partial classes.

The obvious disadvantage is the metadata class is both difficult to support and error prone. For instance, you will not receive an error at compile time if you rename a model property and forget to do the same for its metadata class.

#### DevExpress Fluent API

Considering these disadvantages, we developed a flexible solution – the **DevExpress Fluent API**. The following code is written with the DevExpress Fluent API:

[MetadataType(typeof(PointMetadata))]  
public class Point {  
 public double X { get; set; }  
 public double Y { get; set; }  
}  
public static class PointMetadata {  
 public void BuildMetadata(MetadataBuilder<Point> builder) {  
 builder.Property(x => x.X).DisplayName("Abscissa");  
 builder.Property(x => x.Y).DisplayName("Ordinate");  
 }  
}

Since the DevExpress Fluent API does not allow setting attributes on non-existent properties, renaming properties becomes easy.

It’s sometimes convenient to arrange properties in groups. The classic approach for this is to set a Display Attribute on the grouped properties. With the DevExpress Fluent API, properties can be grouped as follows:

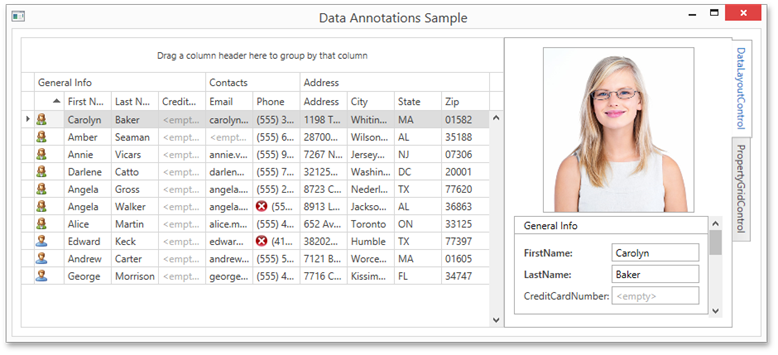
builder  
 .Group("General Info")  
 .ContainsProperty(x => x.FirstName)  
 .ContainsProperty(x => x.LastName)  
 .EndGroup()  
 .Group("Contacts")  
 .ContainsProperty(x => x.Email)  
 .ContainsProperty(x => x.Phone)  
 .EndGroup()  
 .Group("Address")  
 .ContainsProperty(x => x.City)  
 .ContainsProperty(x => x.Zip)  
 .EndGroup();

While the Fluent API is valuable in complex scenarios, in simple cases it still makes sense to use one of the standard approaches.

### Using attributes with DevExpress controls

We have a few demos that clearly demonstrate how to use attributes with our controls: ***“Data Grid - Smart Columns Generation”***, ***“Layout Manager - Data Layout Control”***, ***“Property Grid - DataAnnotation Attributes”***, ***“Property Grid - DataAnnotation Attributes (Fluent API)”***. Please refer to them to see our controls in action.

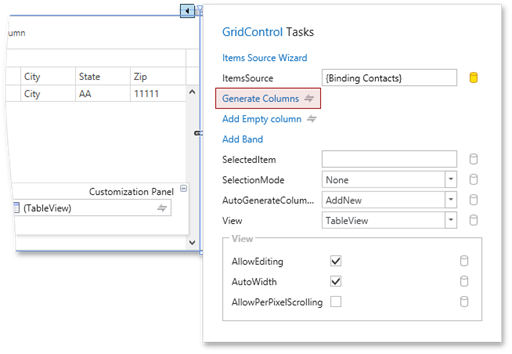
We prepared two samples. The first sample ([E5179](https://www.devexpress.com/Support/Center/CodeCentral/ViewExample.aspx?exampleId=E5179)) illustrates how to use attributes with Entity Framework Code First, the second ([E5180](https://www.devexpress.com/Support/Center/CodeCentral/ViewExample.aspx?exampleId=E5180)) uses the DevExpress Fluent API. The samples are very similar; below you will see a screenshot of how they look:

[](https://community.devexpress.com/blogs/wpf/2014-03-31_1501---Copy_7406DB04.png)

#### GridControl

To configure the GridControl layout in for the underlying data types and attributes, set the [GridControl.EnableSmartColumnsGeneration](https://documentation.devexpress.com/#WPF/DevExpressXpfGridDataControlBase_EnableSmartColumnsGenerationtopic) property to True. In this case, GridControl will automatically generates columns at runtime and customize them according to DataAnnotation and DevExpress Fluent API usage.

You can also generate columns at design time based on the data source objects. GridControl’s SmartTag provides the “Generate Columns” action for this purpose:

[](https://community.devexpress.com/blogs/wpf/2014-03-31_1505_1_230916DC.png)

Generated columns have their [IsSmart](https://documentation.devexpress.com/#WPF/DevExpressXpfGridColumnBase_IsSmarttopic) property set to True. It means that these columns are configured automatically. In the meantime, you are free to set any property manually – your settings will override the automatic settings.

You can call the [TableLayout](https://documentation.devexpress.com/#WPF/DevExpressXpfMvvmDataAnnotationsMetadataBuilder%7ET%7E_TableLayouttopic) method of the Fluent API to define a grouping applicable only for GridControls:

builder.TableLayout()  
 .Group("Personal Data")  
 .ContainsProperty(x => x.FirstName)  
 .ContainsProperty(x => x.LastName)  
 .EndGroup();

The DevExpress Scaffolding Wizard also generates columns with IsSmart=True. To avoid generating columns for some properties, set the ScaffoldColumn(false) attribute for them:

[ScaffoldColumn(false)]  
public string LastName { get; set; }

#### DataLayoutControl

With the Fluent API, you can configure groups by calling the Group method as we did for GridControl. If you like, it's also possible to define a grouping exclusively for DataLayoutControl using the [DataFormLayout](https://documentation.devexpress.com/#WPF/DevExpressXpfMvvmDataAnnotationsMetadataBuilder%7ET%7E_DataFormLayouttopic) method. The following code snippet demonstrates this:

builder.DataFormLayout()  
 .GroupBox("General Info")  
 .ContainsProperty(x => x.FirstName)  
 .ContainsProperty(x => x.LastName)  
 .ContainsProperty(x => x.CreditCardNumber)  
 .EndGroup()  
 .GroupBox("Contacts")  
 .ContainsProperty(x => x.Email)  
 .ContainsProperty(x => x.Phone)  
 .EndGroup()  
 .GroupBox("Address")  
 .ContainsProperty(x => x.Address)  
 .ContainsProperty(x => x.City)  
 .ContainsProperty(x => x.State)  
 .ContainsProperty(x => x.Zip)  
 .EndGroup();

The Scaffolding Wizard generates a simple LayoutControl with LayoutGroup and LayoutItem objects. You can customize them in the designer as you like.

#### PropertyGridControl

PropertyGridControl automatically generates rows for the defined attributes.

PropertyGridControl provides the capability to initialize properties at runtime. In this scenario, you could opt for a custom item initializer. For this, assign the [InstanceInitializer](https://documentation.devexpress.com/#WPF/clsDevExpressXpfMvvmDataAnnotationsInstanceInitializerAttributetopic) attribute to the required property.

For example:

[InstanceInitializer(typeof(Item1), "Item1")]  
public object Item { get; set; }

At runtime, a user will see an additional “Item1” button in the [Property Menu](https://documentation.devexpress.com/#WPF/CustomDocument15631). When s/he presses this button, the Item property will be set to a new Item1 class instance.

### Some tricks

* It is possible to not create a metadata builder class; instead, place the **BuildMetadata** method directly in a data class:

public class Point {  
 public double X { get; set; }  
 public double Y { get; set; }  
 public static void BuildMetadata(MetadataBuilder<Point> builder) {  
 builder.Property(x => x.X).DisplayName("Abscissa");  
 builder.Property(x => x.Y).DisplayName("Ordinate");  
 }  
}

* If you have a library which you can’t modify, use **MetadataLocator** to register metadata classes on startup:

MetadataLocator.Default = MetadataLocator.Create()  
 .AddMetadata<Metadata>();  
public class Metadata {  
 public static void BuildMetadata(MetadataBuilder<Employee> builder) {  
 builder.Property(x => x.FullName).ReadOnly();  
 }  
 public static void BuildMetadata(MetadataBuilder<Team> builder) {  
 builder.Property(x => x.Id).ReadOnly();  
 }  
}

* Metadata for generic classes:

[MetadataType(typeof(BaseGenericClassMetadata<>))]  
public class BaseGenericClass<T1> {  
 public int BaseProperty1 { get; set; }  
 public int BaseProperty2 { get; set; }  
 public int BaseProperty3 { get; set; }  
 public int BaseProperty4 { get; set; }  
}  
public class BaseGenericClassMetadata<T1> {  
 public static void BuildMetadata(MetadataBuilder<BaseGenericClass<T1>> builder) {  
 builder.Property(x => x.BaseProperty2).ReadOnly();  
 }  
 public static void BuildBaseMetadata<T>(MetadataBuilder<T>   
builder) where T : BaseGenericClass<T1> {  
 builder.Property(x => x.BaseProperty4).ReadOnly();  
 }  
}

That’s all. Thank you for your time.

**OTHER RELATED ARTICLES:**

1. [Getting Started with DevExpress MVVM Framework. Commands and View Models.](http://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx)
2. [DevExpress MVVM Framework. Introduction to Services, DXMessageBoxService and DialogService.](https://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx)
3. [DevExpress MVVM Framework. Interaction of ViewModels. IDocumentManagerService.](http://community.devexpress.com/blogs/wpf/archive/2013/10/09/devexpress-mvvm-framework-interaction-of-viewmodels-idocumentmanagerservice.aspx)
4. [DevExpress MVVM Framework. Introduction to POCO ViewModels.](https://community.devexpress.com/blogs/wpf/archive/2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx)
5. [DevExpress MVVM Framework. Interaction of ViewModels. Messenger.](https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx)
6. [DevExpress MVVM Framework. Using Scaffolding Wizards for building Views.](https://community.devexpress.com/blogs/wpf/archive/2013/12/25/devexpress-mvvm-framework-using-scaffolding-wizards-for-building-views.aspx)
7. [DevExpress MVVM Framework. Data validation. Implementing IDataErrorInfo.](https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx)
8. **THIS POST:** DevExpress MVVM Framework. Using DataAnnotation attributes and DevExpress Fluent API.