

# RIBA MVVMLSL Feature Tutorial

---

Revision .....	2
Einführung .....	2
Voraussetzungen.....	2
Beispiele ausprobieren .....	3
Die Musteranwendung.....	4
Übersicht aller Views im fertigen Muster- Programm. ....	4
Erste Schritte .....	5
RelayCommand und RaiseShowMsgBox .....	5
Wie funktioniert das? .....	8
Message- Box mit Callback .....	9
MessageWindow.....	10
File- Dialog .....	11
Command mit InvokeCommandAction (Move mouse over) .....	12
Dialog mit dem NavigatorControl anzeigen .....	14
Wechsel zwischen Dialogen .....	16
Ein ChildWindow .....	19
Nachricht von einem ViewModel zu einem anderen ViewModel .....	24
Ein Wizard.....	27
IsDirty .....	30
Fortgeschrittene I: Details in ChildWindow.....	32
Data Transfer Objects.....	33
DomainService - serverseitig .....	35
DomainService – clientseitig (Model) .....	37
ViewModelLocator .....	42
MainPage .....	45
Autor- Dialog .....	47
Autor- Details .....	51
Bücher- Dialog .....	51
Unit Test.....	51
Eigene Animation implementieren .....	51

## Revision

Version 0.2	21.03.2011	C. Granwehr	Wizard, OpenFileDialog
Version 0.3	23.03.2011	C. Granwehr	Voraussetzungen
Version 0.4	23.03.2011	C. Granwehr	Beispiele ausprobieren
Version 0.5	28.03.2011	C. Granwehr	
Version 0.6	04.04.2011	C. Granwehr	MessageWindow, IsDirty, ChildWindowClosed
Version 0.7	14.04.2011	C. Granwehr	Generischer ViewModelLocator nach Idee von John Papa
Version 0.8	04.05.2011	C. Granwehr	ExchangeDataAvailableEventArgs
Version 0.9	17.05.2011	C. Granwehr	\n in MessageWindow

## Einführung

RIBA ist ein Beispiel für eine verteilte LOB- Anwendung in Silverlight in der agile Pattern der Softwareentwicklung exemplarisch verwendet werden.

Die Bibliothek RIBA.MVVMLSL dient der dem vereinfachten Einsatz des MVVM- Pattern in der GUI- Programmierung unserer Beispielanwendung.

RIBA.MVVMLSL unterstützt die Entwicklung mit MVVM durch eine einfache Implementierung von ICommand und einer Basisklasse für ViewModel. Zusätzlich werden folgende Features unterstützt:

- Einfache Verwendung von MessageBox und OpenFileDialog
- Nachrichten von ViewModel zu ViewModel
- Unterstützung von ChildWindow
- NavigatorControl für die animierte Anzeige von Dialog- Seiten oder Wizards

## Voraussetzungen

Die Beispiele (MVVMtutorial und MVVMtutorialAdvanced) setzt folgendes voraus:

- Visual Studio 2010 SP1
- Silverlight Toolkit April 2010
- WCF RIA Services SP1 for Silverlight 4
- WCF RIA Services Toolkit December 2010
- Microsoft Expression Blend Software Development Kit (SDK) for Silverlight 4 oder Expression Blend 4

Die vollständigen Solutions sind im „Feature & Advanced Tutorial (German)“- Download enthalten.

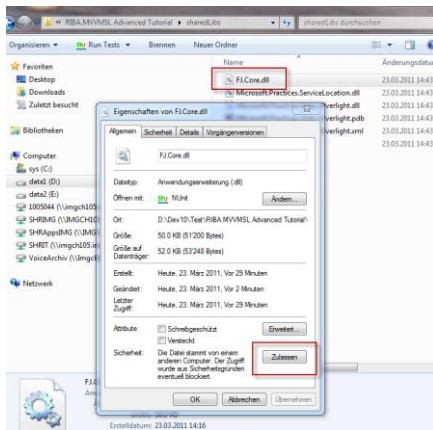
# Beispiele ausprobieren

## dll zulassen

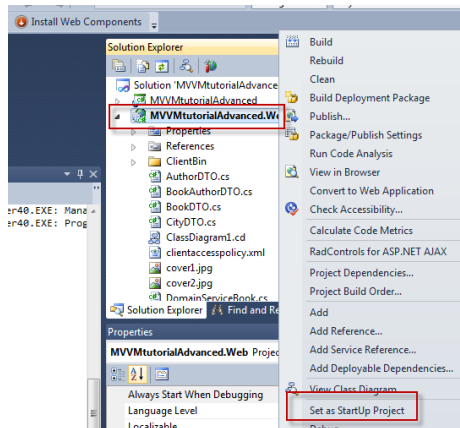
Wenn beim Kompilieren z.b. folgende Meldung erscheint:

Could not load the assembly file:///D:/Dev10/Test/RIBA.MVVMLSL Advanced Tutorial/sharedLibs\FJ.Core.dll. This assembly may have been downloaded from the Web. If an assembly has been downloaded from the Web, it is flagged by Windows as being a Web file, even if it resides on the local computer. ...

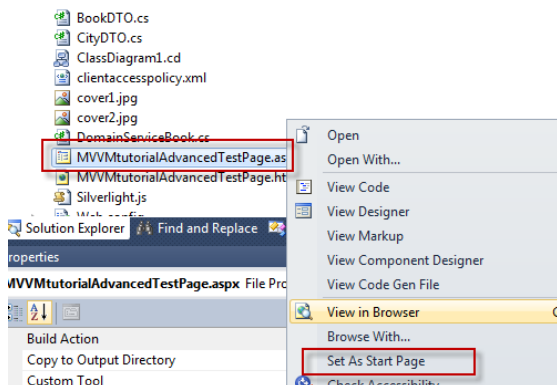
Bitte alle DLL im Unterordner sharedLibs zulassen:



## Webprojekt als Startup- Projekt setzen



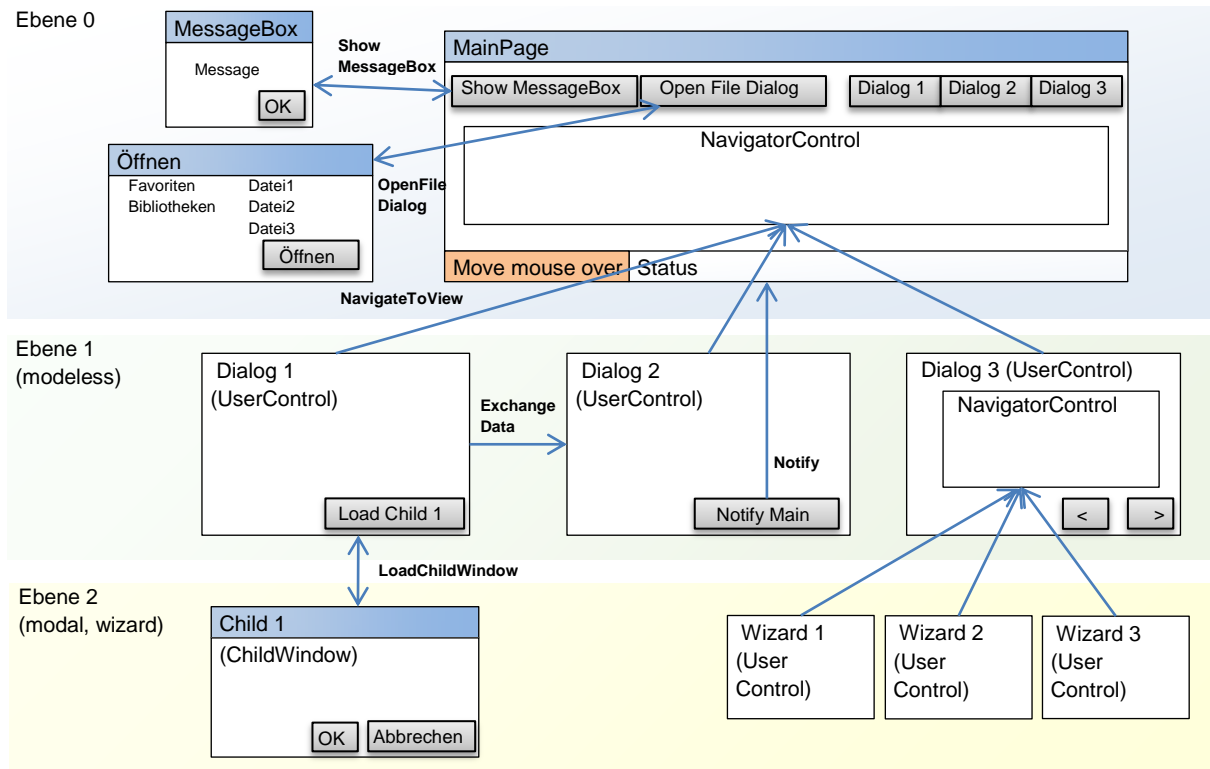
## MVVMtutorial...TestPage.aspx oder .html als Startseite setzen.



# Die Musteranwendung

Ziel der Musteranwendung ist es zu allen Features des Frameworks ein Beispiel aufzuzeigen.

## Übersicht aller Views im fertigen Muster- Programm.



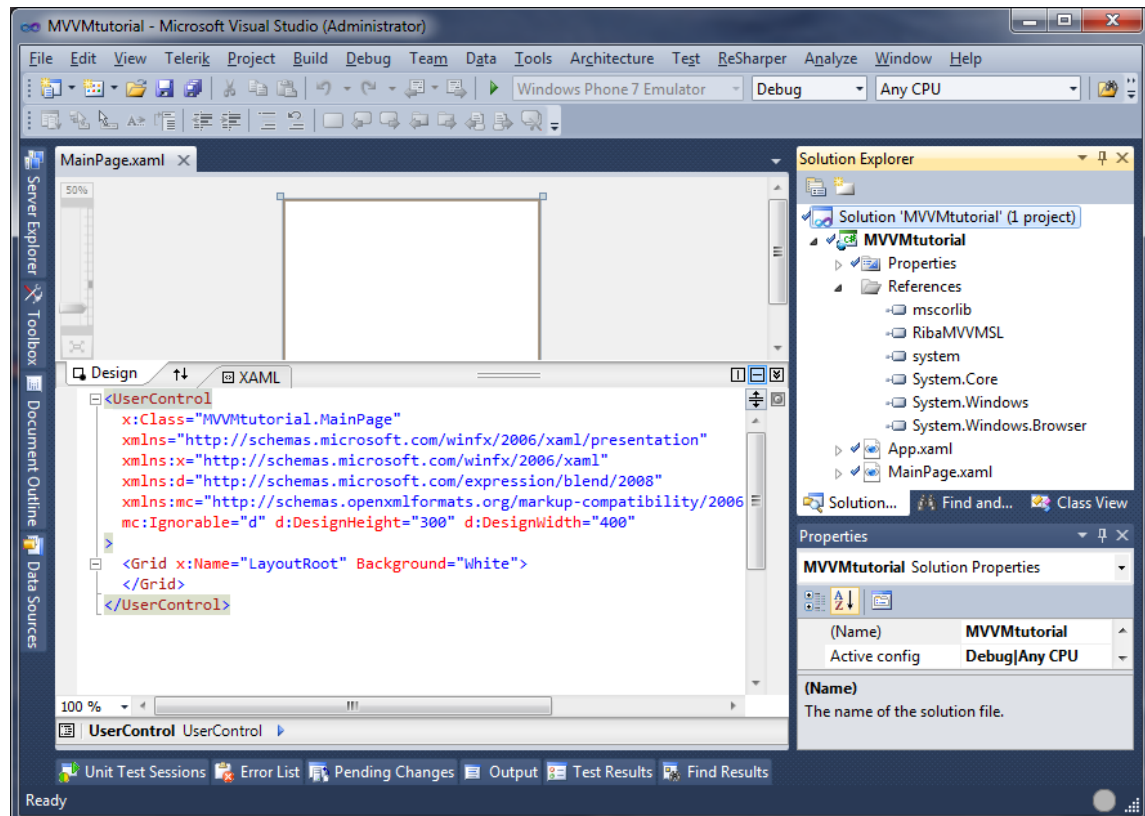
# Erste Schritte

## RelayCommand und RaiseShowMsgBox

### Musteranwendung erstellen

Erstellen Sie eine neue Silverlight 4 Application, welche nicht in einer Website gehostet wird und RIA Services nicht unterstützt.

Referenzieren Sie RibaMVMSL.dll



### Erstellen von MainPageViewModel und instanziiieren aus XAML

Fügen Sie eine neue Klasse MainPageViewModel zur Anwendung hinzu und leiten Sie diese von ViewModelBase ab.

```
using Riba.MVMSL;

namespace MVVMtutorial
{
    public class MainPageViewModel : ViewModelBase
    {
    }
}
```

Instanziiieren Sie MainPageViewModel in MainPage.xaml als DataContext.

```
<UserControl.DataContext>
    <me:MainPageViewModel />
</UserControl.DataContext>
```

Der Namespace für das ViewModel in MainPage.xaml muss noch angegeben werden:

```
xmlns:me="clr-namespace:MVVMtutorial"
```

### Behandlung der Basis- Events

Fügen Sie zusätzlich folgenden Namespace in MainPage.xaml hinzu:

```
xmlns:MVVMLSL="clr-namespace:Riba.MVVMLSL;assembly=RibaMVVMLSL"
```

Das referenzierte Assembly enthält ViewEventHandlerControl von dem eine Instanz in MainPage.xaml platziert werden muss. Wo das Control innerhalb der Seite platziert wird ist egal, die Seite muss sich aber kompilieren lassen!

Im Konstruktor von ViewEventHandlerControl wird die Behandlung folgender Events registriert: ShowMessageBox, LoadChildWindow und CloseChildWindowView.

```
<Grid>  
    <MVVMLSL:ViewEventHandlerControl />
```

### Behandlung der Basis- Events des ViewModels im Code- Behind

Alternativ kann die Behandlung der Events auch im Code- Behind der View realisiert werden. Rufen Sie dazu die Methode HandleViewModelBaseEvents aus der statischen Klasse ViewEventHandler auf.

```
using Riba.MVVMLSL;  
  
namespace MVVMtutorial  
{  
    public partial class MainPage  
    {  
        public MainPage()  
        {  
            InitializeComponent();  
  
            ViewEventHandler.HandleViewModelBaseEvents(DataContext as ViewModelBase);  
        }  
    }  
}
```

## Button und Command erstellen

Erstellen Sie im ViewModel einen RelayCommand, der die Nachricht eine Message-Box anzuzeigen versenden soll.

```
using System.Windows;
using Riba.MVVMLSL;

namespace MVVMtutorial
{
    public class MainPageViewModel: ViewModelBase
    {
        public RelayCommand CommandShowMessageBox { get; private set; }

        public MainPageViewModel()
        {
            CommandShowMessageBox =
                new RelayCommand(OnCommandShowMessageBox){ IsEnabled = true };
        }

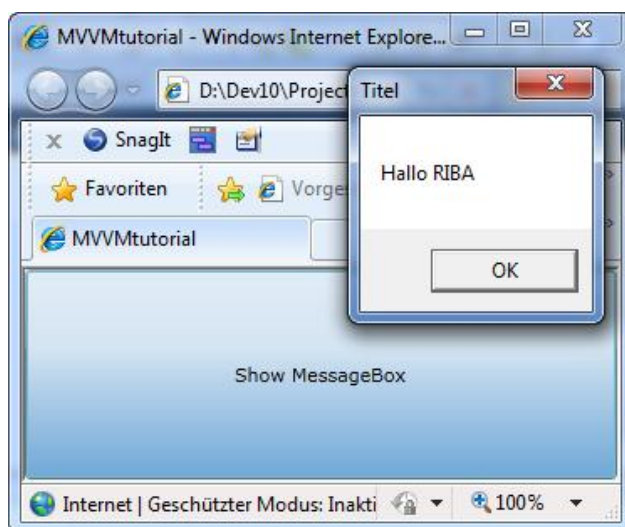
        private void OnCommandShowMessageBox(object commandParameter)
        {
            var args =
                new ShowMsgBoxEventArgs("Hallo RIBA", "Titel", MessageBoxButton.OK);

            RaiseShowMsgBox(args);
        }
    }
}
```

## Im XAML binden Sie einen Button an den Command

```
<Grid>
    <Button
        Content="Show MessageBox"
        Command="{Binding CommandShowMessageBox}"
    />
</Grid>
```

## Starten Sie die Anwendung, und Klicken Sie auf den Button



## Wie funktioniert das?

In folgendem Sequenz- Diagramm wird aufgezeigt, wie das vorherige Beispiel vereinfacht abläuft.

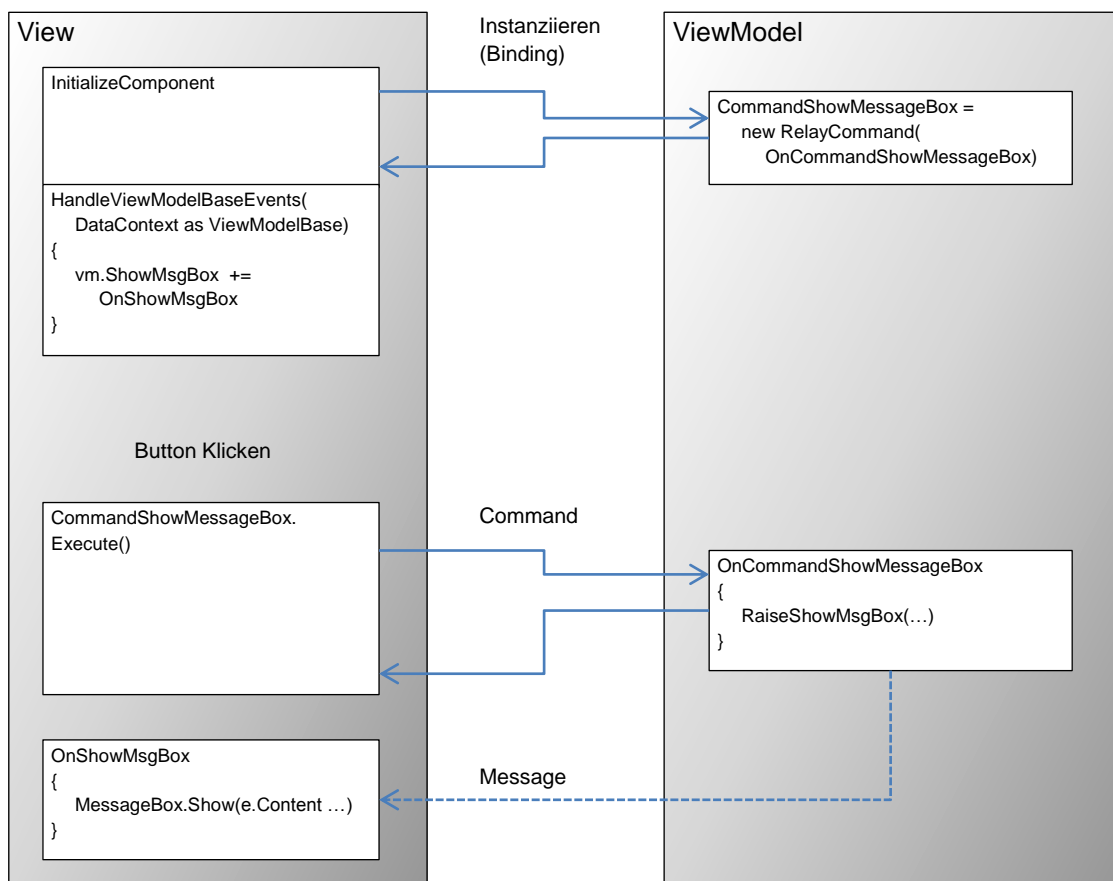
Der XAML- Parser erstellt eine Instanz des ViewModels und weist dieses dem DataContext (Binding) der View zu (Die View kennt das VM).

Das ViewModel instanziert den Command und stellt ihn als Property der View fürs Binding zur Verfügung.

Im der View wird mittels HandleViewModelBaseEvents der Event- Handler OnShowMsgBox für den ShowMessageBox- Event installiert.

Beim Klicken auf den Button ruft die View die Execute- Methode des gebundenen Command im ViewModel auf.

Die Execute- Methode des Commands (OnCommandShowMessageBox) sendet die Nachricht ShowMsgBox an alle, die diese abonniert haben. Die View bekommt die Nachricht und erstellt in OnShowMsgBox eine Standard- Message- Box mit den übergebenen Parametern.





## Message- Box mit Callback

Sie können die Message- Boxen auch zum Beantworten von Fragen verwenden. Die Antwort des Benutzers wird dabei in einem Callback ausgewertet.

### Callback erstellen

Der 4. Parameter von ShowMsgBoxEventArgs ist vom Typ Action<MessageBoxResult> und kann beispielsweise wie folgt definiert werden.

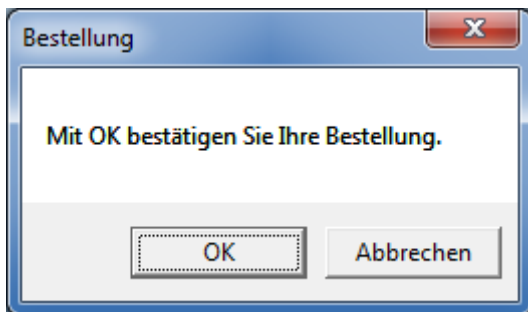
```
private void MsgCallback(MessageBoxResult result)
{
    if (result == MessageBoxResult.OK)
        RaiseShowMsgBox(new ShowMsgBoxEventArgs(
            "Bestellung aufgegeben", "Bestellung", MessageBoxButton.OK));
}
```

### Callback in den EventArgs verwenden

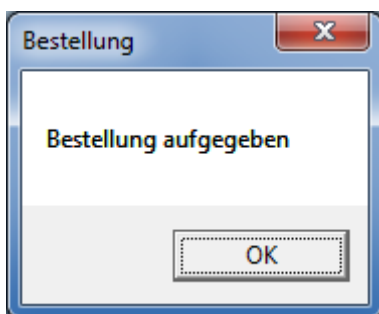
```
var args = new ShowMsgBoxEventArgs
(
    "Mit OK bestätigen Sie Ihre Bestellung.",
    "Bestellung",
    MessageBoxButton.OKCancel,
    MsgCallback
);

RaiseShowMsgBox(args);
```

### Und so sieht's aus



Und nach Klicken auf <OK>



## MessageWindow

Als Alternative zur MessageBox dient das MessageWindow. Dieses ist kein Systemelement sondern basiert auf einem ChildWindow.

Vorteile von MessageWindow:

- Mehrere Buttons möglich (Maximum 4)
- Callback möglich (null wenn nicht verwendet)
- Buttontext und Tag frei wählbar
- Auch Buttons sind einfach lokalisierbar
- Einsatz von Style (null, wenn nicht verwendet). Ein Beispiel zu einem Style finden Sie im Advanced Tutorial.

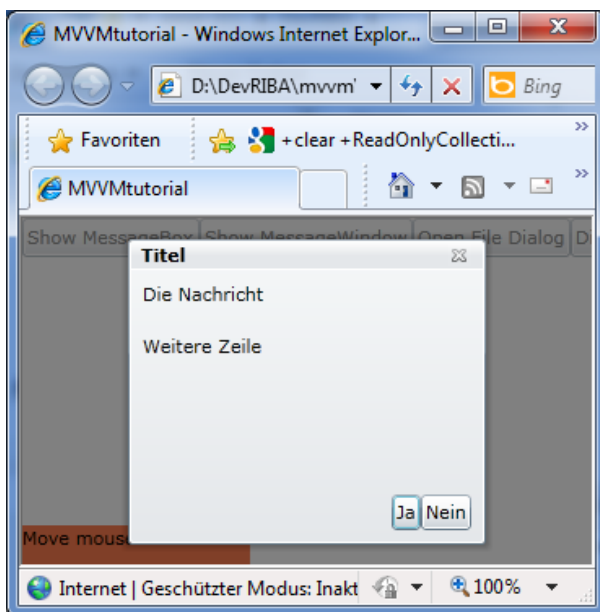
Beim Schliessen des Dialogs über den Standard Schliess- Button oben rechts wird derselbe Tag zurückgemeldet wie wenn der am weitesten links stehende Button gewählt wurde.

### Beispiel mit Callback

```
var buttons = new List<MessageWindowButton>();
buttons.Add(new MessageWindowButton { Caption = "Ja", Tag = "yes" });
buttons.Add(new MessageWindowButton { Caption = "Nein", Tag = "no" });
var eventArgs = new ShowMessageWindowEventArgs(
    "Die Nachricht\n\Weiterer Linie", "Titel", buttons, MessageWindow_Callback, null);
RaiseShowMessageWindow(eventArgs);
```

```
private void MessageWindow_Callback(object tag)
{
    object mode;
    if (tag.ToString() == "yes")
        mode = 1;
    ...
}
```

### Und so sieht's aus



## File- Dialog

Mit `RaiseOpenFileDialog` können Sie die View zur Anzeige eines File- Dialogs veranlassen.

### Command erstellen

Im Execute- Handler des Commands wird `RaiseOpenFileDialog` mit den gewünschten Parametern aufgerufen. Die Parameter `filter`, `filterIndex` und `multiselect` entsprechen ihren Pendanten im Konstruktor von `System.Windows.Controls.OpenFileDialog`.

```
CommandOpenFileDialog = new RelayCommand(
    (
        cmdPar =>
        {
            var openFileDialogEventArgs = new OpenFileDialogEventArgs(
                "JPG Files (*.jpg;*.png)|*.jpg;*.png | All Files (*.*)|*.*",
                1, false, OpenFileDialogCallback);
            RaiseOpenFileDialog(openFileDialogEventArgs);
        }
    );
CommandOpenFileDialog.IsEnabled = true;
```

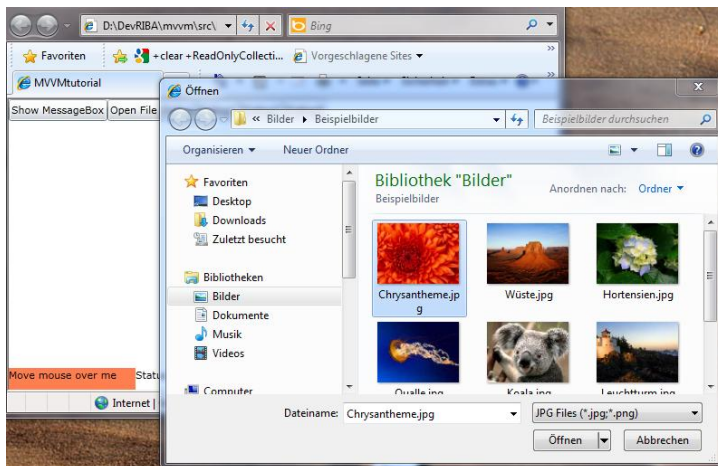
### Callback

Im Callback können die gewählten Dateien ausgewertet werden. Der Callback wird nur aufgerufen, wenn der Benutzer mindestens eine vorhandene Datei ausgewählt hat.

```
private void OpenFileDialogCallback(IEnumerable<FileInfo> files)
{
    if (files.Count() == 0)
        return;

    var file = files.FirstOrDefault();
    RaiseShowMessageBox(new ShowMsgBoxEventArgs(file.Name, "Gewählte Datei"));
}
```

### Und so sieht's aus



## Command mit InvokeCommandAction (Move mouse over)

In Silverlight 4 kann die Command- Eigenschaft nur für das Click- Event eines Button oder eines Hyperlinks verwendet werden. Wenn man auf die Events anderer Controls oder auf andere Events reagieren will, bieten sich die Interaction Trigger aus der Expression Interactivity- DLL an.

### Command erstellen

Erstellen Sie einen neuen RelayCommand in ViewModel der MainPage. Im folgenden Beispiel wird dem RelayCommand eine anonyme Methode übergeben. Diese soll aufgerufen werden, wenn ein Anwender mit der Maus über einen Text fährt.

```
namespace MVVMtutorial
{
    public class MainPageViewModel: ViewModelBase
    {
        public RelayCommand CommandShowMessageBox { get; private set; }
        ...

        public MainPageViewModel()
        {
            CommandMouseEnter = new RelayCommand
            (
                cmdPar =>
                    RaiseShowMsgBox(new ShowMsgBoxEventArgs(
                        cmdPar.ToString(), "MouseEnter", MessageBoxButton.OK))
            );
            ...
        }
    }
}
```

### System.Windows.Interactivity

Erstellen Sie eine Referenz auf System.Windows.Interactivity. Die DLL wird beispielsweise mit dem Microsoft Expression Blend Software Development Kit (SDK) for Silverlight 4 installiert.

Fügen Sie den Namespace im XAML der MainPage ein

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
```

### TextBlock mit EventTrigger für MouseEnter

Fügen Sie in der MainPage 2 RowDefinition zum Grid hinzu. Platzieren Sie einen TextBlock in der zweiten Zeile. Dem Textblock wird nun ein Trigger zum Behandeln des MouseEnter- Events beigefügt. Dabei wird der zuvor erstellte Command in InvokeCommandAction gebunden.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>

  <Button ... />

  <Border Grid.Row="1" Width="170" Background="Coral">
    <TextBlock Text="Move mouse over me">
      <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseEnter">
          <i:InvokeCommandAction
            Command="{Binding CommandMouseEnter}"
            CommandParameter="123"
          />
        </i:EventTrigger>
      </i:Interaction.Triggers>
    </TextBlock>
  </Border>
</Grid>
```

### Command reagiert noch nicht

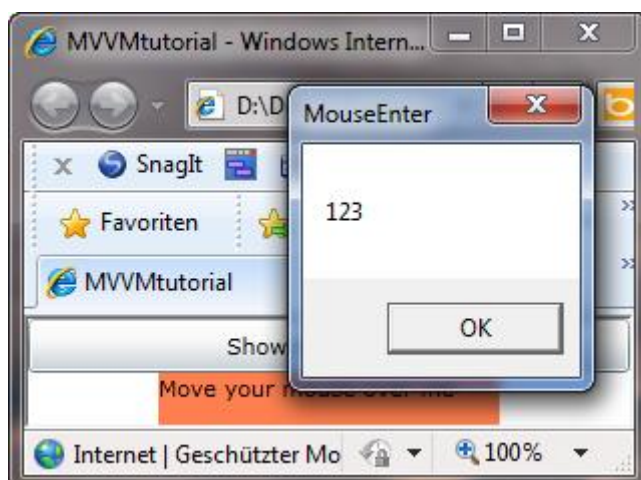
Starten Sie die Anwendung. Wenn Sie jetzt mit der Maus über die TextBox fahren passiert noch nichts. Wieso ?

Sie müssen den Command zuerst noch aktivieren. Fügen Sie folgenden Code nach der Instanziierung des Commands hinzu.

```
CommandMouseEnter.IsEnabled = true;
```

### Das Resultat

Starten Sie das Programm erneut und fahren Sie mit der Maus über den Text. Eine Message- Box in der der Inhalt des CommandParameter angezeigt wird sollte nun erscheinen.



## Dialog mit dem NavigatorControl anzeigen

### ViewModel des Dialogs

Fügen Sie die Klasse Dialog1ViewModel zum Projekt hinzu und leiten Sie diese von ViewModelBase ab.

```
using Riba.MVMSL;

namespace MVVMtutorial
{
    public class Dialog1ViewModel : ViewModelBase
    {
    }
}
```

### View des Dialogs

Fügen Sie ein neues Silverlight User Control zur Anwendung hinzu und nennen Sie es Dialog1. Setzen Sie dessen DataContext auf Dialog1ViewModel. Platzieren Sie im neuen Dialog einen Textblock mit dem Text „Dialog1“.

```
<UserControl ...
>
    <UserControl.DataContext>
        <me:Dialog1ViewModel />
    </UserControl.DataContext>

    <Grid>
        <TextBlock Text="Dialog1" />
    </Grid>
</UserControl>
```

### Laden des Dialogs aus dem View- Model von MainPage.

Fügen Sie in MainPage einen weiteren Command hinzu. In dessen executeHandler wird RaiseNavigateToView aufgerufen.

```
CommandDialog = new RelayCommand
(
    cmdPar =>
    {
        var p = new NavigateToViewEventArgs(typeof(Dialog1));
        RaiseNavigateToView(p);
    }
) { IsEnabled = true };
```

## NavigatorControl und Command

Nun muss nur noch die View auf die im ViewModel versendete Nachricht reagieren und den Dialog anzeigen.

Fügen Sie eine zusätzliche Zeile zum Grid hinzu und verschieben Sie den Textblock aus dem vorherigen Schritt in die letzte Zeile.

Dann fügen Sie einen Button hinzu und binden Sie dessen Command- Eigenschaft an den CommandDialog aus dem ViewModel.

Fügen Sie nun noch ein NavigatorControl in die zweite Zeile ein. Dieses soll den ganzen mittleren Raum in unserer MainPage übernehmen.

```
<UserControl
    ...
>
<UserControl.DataContext>
    <me:MainPageViewModel />
</UserControl.DataContext>

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="24" />
        <RowDefinition />
        <RowDefinition Height="24" />
    </Grid.RowDefinitions>

    <StackPanel Orientation="Horizontal">
        <Button .../>
        <Button
            Content="Dialog1"
            Command="{Binding CommandDialog}"
        />
    </StackPanel>

    <MVVMLSL:NavigatorControl Grid.Row="1" />

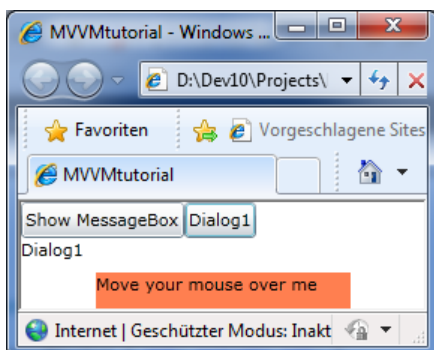
    ...

    <TextBlock Text="Move your mouse over me">

        ...
    </TextBlock>
</Grid>
</UserControl>
```

## Das Resultat

Starten Sie die Anwendung und klicken Sie auf <Dialog1>. Der Dialog wird angezeigt.



## Wechsel zwischen Dialogen

### Pimp my Dialog

Damit wir die Animationen und den Datenaustausch zwischen Dialogen in den folgenden Schritten besser visualisieren können, stattdessen wir unseren Dialog weiter aus.

### PrivateData

Eine erste TextBox wird an das Property PrivateData gebunden. Alle Klassen, welche von ViewModelBase ableiten haben diese Eigenschaft. PrivateData steht erst nach dem Erhalt von ExchangeDataAvailable zur Verfügung. Die privaten Daten eines Dialogs bleiben auch nach zwischenzeitlicher Navigation zu anderen Dialogen erhalten.

### ExchangeData und ExchangeDataAvailable

Eine weitere TextBox wird an das Property ExchangeData gebunden. Alle Klassen, welche von ViewModelBase ableiten haben diese Eigenschaft. In ExchangeData steht nach dem Laden einer View mit dem NavigatorControl in dessen ViewModel der Wert welcher im vorhergehenden ViewModel gesetzt wurde. ExchangeData ist im Konstruktor des ViewModel noch nicht gesetzt! Die Daten stehen erst nach dem Erhalt von ExchangeDataAvailable zur Verfügung. ExchangeData bietet einen einfachen Mechanismus, Daten an einen folgenden Dialog weiterzugeben.

Passen Sie den Dialog wie folgt an:

```
<Border Background="AliceBlue">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Text="Dialog1" />

    <TextBlock Text="PrivateData" Grid.Row="1" />
    <TextBox
      Text="{Binding PrivateData, Mode=TwoWay}"
      Grid.Row="1"
      Grid.Column="1"
    />

    <TextBlock Text="ExchangeData" Grid.Row="2" />
    <TextBox
      Text="{Binding ExchangeData, Mode=TwoWay}"
      Grid.Row="2" Grid.Column="1"
    />
  </Grid>
</Border>
```

Erstellen Sie nun 2 weitere Dialoge 2 und 3 gemäss Dialog 1 (View und ViewModel). Passen Sie in den Kopien jeweils den Text des ersten TextBlocks an (Dialog2, 3). Geben Sie jedem Dialog eine eigene Hintergrundfarbe. Achten Sie darauf, dass dem DataContext jeweils das richtige ViewModel zugewiesen wird.



## Dialoge animiert laden

Passen Sie den `executeHandler` von `CommandDialog` wie folgt an. In den `NavigateToViewEventArgs` können Sie angeben ob die Animation für den Wechsel von einem Dialog zum anderen vorwärts oder rückwärts ablaufen soll, oder ob keine Animation zur Anwendung kommen soll.

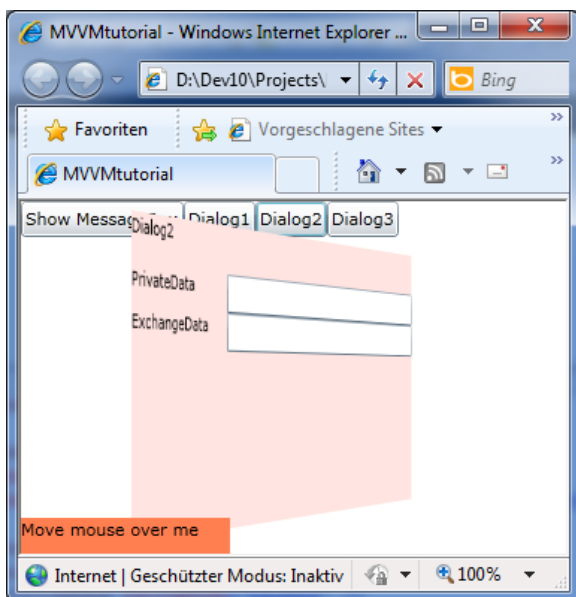
```
CommandDialog = new RelayCommand
(
    cmdPar =>
    {
        NavigateToViewEventArgs p;

        if (cmdPar.ToString() == "Dialog2")
            p = new NavigateToViewEventArgs(typeof(Dialog2), AnimationMode.Forward);
        else if (cmdPar.ToString() == "Dialog3")
            p = new NavigateToViewEventArgs(typeof(Dialog3), AnimationMode.Forward);
        else
            p = new NavigateToViewEventArgs(typeof(Dialog1), AnimationMode.Forward);

        RaiseNavigateToView(p);
    }
) { IsEnabled = true };
```

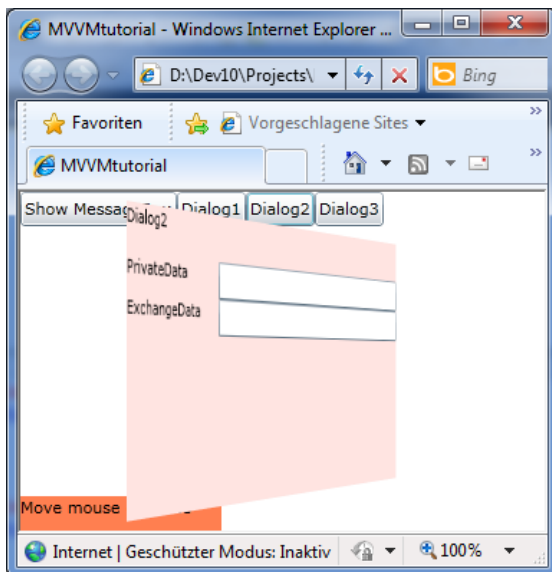
## Das Resultat

Ein bereits angezeigter Dialog wird um 90° gedreht; er verschwindet. Der neu anzuzeigende Dialog wird ebenfalls aus seiner Ausgangslage gedreht, bis er vollständig sichtbar ist. Im `AnimationMode.Forward` läuft die Animation im Uhrzeigersinn, bei `Backward` im Gegenuhrzeigersinn ab.



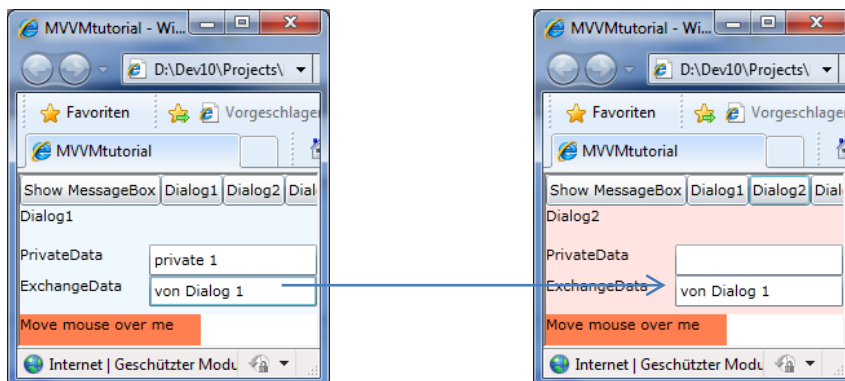
## Z- Order des NavigatorControl

Wenn Sie das NavigatorControl am Schluss im XAML von MainPage setzten, wird sich der animierte Dialog über den orange hinterlegten Text bewegen!



## ExchangeData

Geben Sie in einem Dialog einen Text in der TextBox für ExchangeData ein und wählen Sie einen anderen Dialog an. Im neu geladenen Dialog wird der vorher erfasste Wert ebenfalls in der entsprechenden TextBox angezeigt. Er kann vor der Wahl eines weiteren Dialogs wiederum verändert werden ...



## Ein ChildWindow

### ViewModel

Erstellen Sie eine neue Klasse ChildWindow1ViewModel und leiten Sie diese von ViewModelBase ab.

Im Konstruktor abonnieren Sie den Event ExchangeDataAvailable. Dieser wird aufgerufen, wenn die Daten und der Callback des Aufrufers bereit stehen.

Im Event-Handler von ExchangeDataAvailable können die in ExchangeData bereitgestellten Daten ausgewertet werden. Hier kann auch der Command für den OK-Button enabled werden, da nun der Callback verfügbar ist.

Beim Klicken auf den OK-Button wird parallel zum Schliessen des Dialogs der executeHandler des entsprechenden Commands ausgelöst.

Im Handler werden die geänderten Daten wieder an den Aufrufer übermittelt, falls ein Callback angegeben wurde. Danach wird der View die Close-Nachricht übermittelt.

ChildWindowClosed wird immer nach dem Schliessen aufgerufen.

```
using System;
using Riba.MVVMSL;

namespace MVVMtutorial
{
    public class ChildWindow1ViewModel : ViewModelBase
    {
        public RelayCommand CommandOk { get; private set; }

        public ChildWindow1ViewModel()
        {
            ExchangeDataAvailable += ChildWindow1ViewModel_ExchangeDataAvailable;

            CommandOk = new RelayCommand(
                (
                    handler =>
                    {
                        if (Callback != null)
                            Callback(this, ExchangeData);

                        RaiseCloseChildWindowView();
                    }
                );

            ChildWindowClosed += ChildWindow1ViewModel_ChildWindowClosed;
        }

        void ChildWindow1ViewModel_ExchangeDataAvailable(
            object sender, ExchangeDataAvailableEventArgs e)
        {
            CommandOk.IsEnabled = true;
        }
    }

    void ChildWindow1ViewModel_ChildWindowClosed(object sender, ChildWindowClosedEventArgs e)
    {
        RaiseShowMsgBox(
            new ShowMsgBoxEventArgs(string.Format("Result={0}", e.DialogResult), ""));
    }
}
```

## View

Fügen Sie ein Silverlight Child Window zur Anwendung hinzu und nennen Sie es ChildWindow1.

Der XAML- Code für das ChildWindows im folgenden Beispiel wurde leicht angepasst. Die Button- Höhe wird nun über die RowDefinition mit Auto gesteuert.

Geben Sie MVVMtutorial als Namespace an und setzen Sie wie gewohnt das ViewModel als DataContext.

Binden Sie das Text- Property an ExchangeData.

Referenzieren Sie zusätzlich RibaMVVMLSL und fügen Sie ein ViewEventHandlerControl hinzu.

Der Event- Handler des OK- Buttons wird entfernt, dafür wird ein CommandOk gebunden.

```
<controls:ChildWindow
  xmlns:me="clr-namespace:MVVMtutorial"
  xmlns:MVVMLSL="clr-namespace:Riba.MVVMLSL;assembly=RibaMVVMLSL"
  ...
>
<controls:ChildWindow.DataContext>
  <me:ChildWindow1ViewModel />
</controls:ChildWindow.DataContext>

<Grid Margin="2">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="130" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <MVVMLSL:ViewEventHandlerControl />

  <TextBlock Text="ExchangeData" />
  <TextBox Text="{Binding ExchangeData , Mode=TwoWay}" Grid.Column="1" />

  <StackPanel Grid.Row="2" Grid.ColumnSpan="2" Orientation="Horizontal"
    HorizontalAlignment="Right">
    <Button
      x:Name="CancelButton" Content="Cancel"
      Click="CancelButton_Click"
      Width="60" Margin="0, 0, 2, 0"
    />
    <Button
      x:Name="OkButton" Content="OK"
      Command="{Binding CommandOk, Mode=OneWay}"
      Click="OkButton_Click"
      Width="60"
    />
  </StackPanel>
</Grid>
</controls:ChildWindow>
```

### CodeBehind der View

Belassen Sie den Eventhandler für den Cancel- Button im CodeBehind. Durch das Setzen von DialogResult wird das ChildWindow geschlossen. Der Wert in DialogResult wird aber nicht ausgewertet!

Entfernen Sie den Eventhandler des OK- Button.

```
using System.Windows;

namespace MVVMtutorial
{
    public ChildWindow1(object exchangeData, Action<ViewModelBase, object> callback)
    {
        public ChildWindow1()
        {
            InitializeComponent();
        }

        private void CancelButton_Click(object sender, RoutedEventArgs e)
        {
            DialogResult = false;
        }
    }
}
```

### Aufruf des ChildWindows aus dem Dialog

In Dialog1ViewModel definieren Sie einen Command, welcher RaiseLoadChildWindow aufruft.

Übergeben Sie die Daten welche das Child- Window erhalten soll und den Callback mittels LoadChildWindowEventArgs. In unserem Beispiel wird das Property ExchangeData aus dem ViewModel mitgegeben.

Im Call- Back werten Sie die im Child- Window gesetzten Austausch- Daten wieder aus. Der Call- Back wird nur aufgerufen, wenn der Benutzer OK gewählt hat!

```
using Riba.MVVMLSL;

namespace MVVMtutorial
{
    public class Dialog1ViewModel : ViewModelBase
    {
        public RelayCommand CommandLoadChild { get; private set; }

        public Dialog1ViewModel()
        {
            CommandLoadChild = new RelayCommand(OnLoadChild) { IsEnabled = true };
        }

        private void OnLoadChild(object commandParameter)
        {
            var args = new LoadChildWindowEventArgs(
                typeof(ChildWindow1), ExchangeData, OnLoadViewCallback);

            RaiseLoadChildWindow(args);
        }

        public void OnLoadViewCallback(ViewModelBase vm, object exchangeData)
        {
            ExchangeData = exchangeData;
        }
    }
}
```

### CodeBehind von Dialog1

Damit aufgrund der Nachricht LoadChildWindow, welche via RaiseLoadChildWindow ausgelöst wird, die View des Kind- Fensters auch geladen wird, muss im Code- Behind von Dialog1 der passende Eventhandler mit HandleViewModelBaseEvents registriert werden. Alternativ dazu kann auch ein ViewEventHandlerControl im XAML der Seite platziert werden.

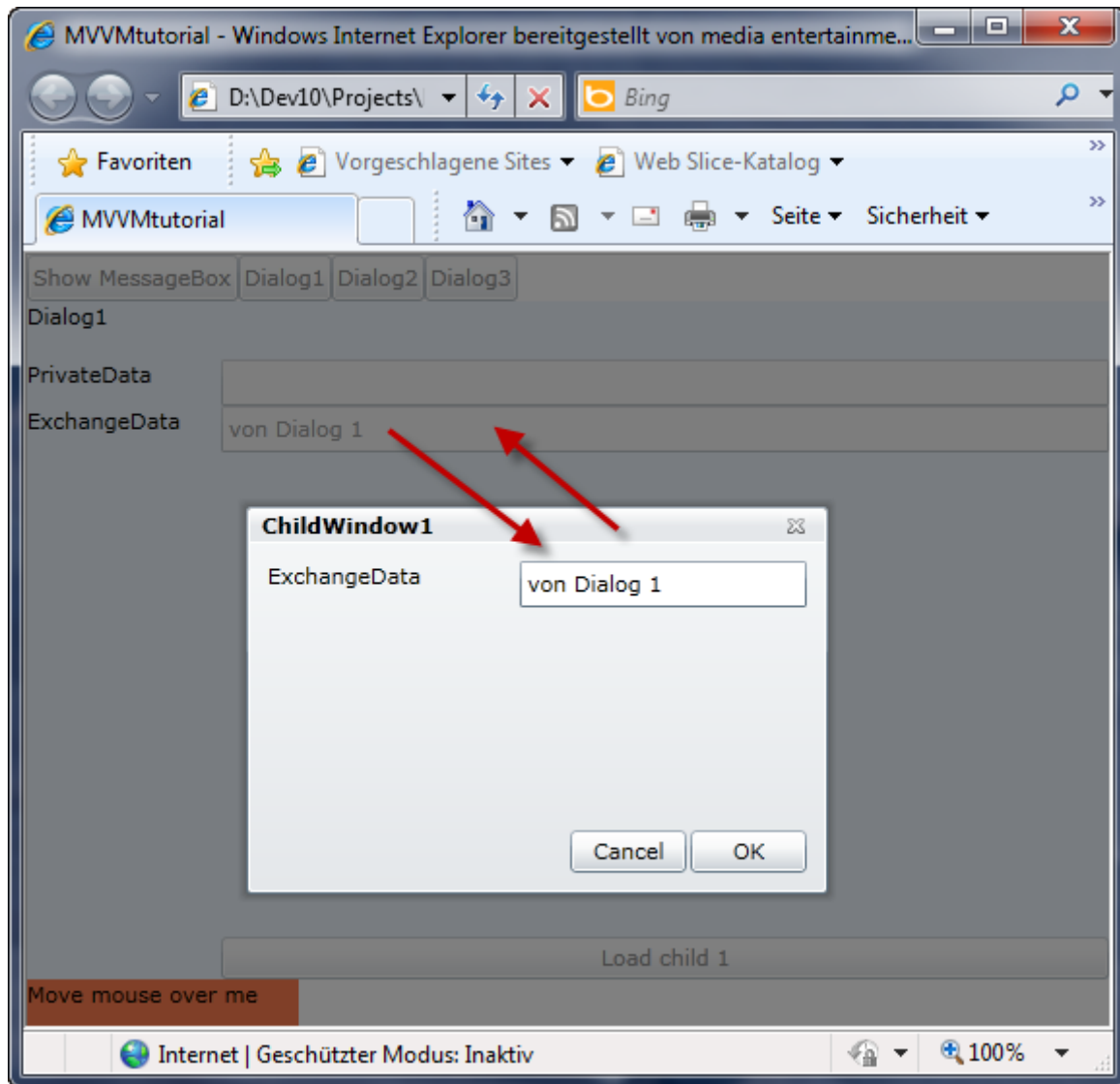
```
public Dialog1()
{
    InitializeComponent();

    ViewEventHandler.HandleViewModelBaseEvents(DataContext as ViewModelBase);
}
```

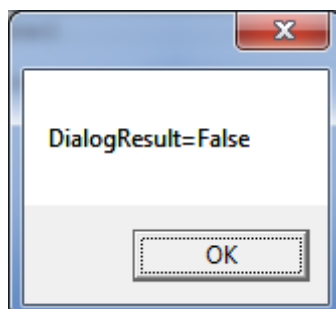
## Button in XAML von Dialog1

```
<Button
  Content="Load child 1"
  Command="{Binding CommandLoadChild}"
  Grid.Row="4" Grid.Column="2"
/>
```

## Das Resultat



Danach wird der DialogResult aus ChildWindowClosed angezeigt.



## Nachricht von einem ViewModel zu einem anderen ViewModel

### ViewModel von Dialog2

Erstellen Sie einen Command in dessen executeHandler die Nachricht versendet wird.

Da die Nachrichten nicht gezielt an ein bestimmtes ViewModel gerichtet werden können, wir der Nachricht ein Tag mitgegeben, diesen kann ein weiteres ViewModel beim Empfangen der Nachricht auswerten und so bestimmen, ob die Nachricht für es bestimmt ist.

Die Nachricht wird mittels RaiseNotify der statischen Klasse Notifier versendet.

Im untenstehenden Beispiel wir der Nachricht neben den zu übermittelnden Daten noch ein Callback mitgegeben. Dieser kann vom Empfänger aufgerufen werden.

Als Daten wird im folgenden Beispiel die Anzahl der versendeten Nachrichten übermittelt.

Das Property CallBackData wird zum Anzeigen der vom Empfänger zurückgesendeten Daten verwendet.

```
using Riba.MVVMLSL;

namespace MVVMtutorial
{
    public class Dialog2ViewModel : ViewModelBase
    {
        private int _calls;
        public RelayCommand CommandNotifyMain { get; private set; }

        private object _callBackData;
        public object CallBackData
        {
            get { return _callBackData; }
            set { SetPropertyValue(ref _callBackData, value, () => CallBackData); }
        }

        public Dialog2ViewModel()
        {
            CommandNotifyMain =
                new RelayCommand(OnCommandNotifyMain) { IsEnabled = true };
        }

        private void OnCommandNotifyMain(object commandParameter)
        {
            _calls++;

            var n = new Notifier.NotificationEventArgs(
                "MyTag",
                string.Format("Aufruf {0} aus Dialog2ViewModel", _calls),
                CallbackFromMainPage);

            Notifier.RaiseNotify(n);
        }

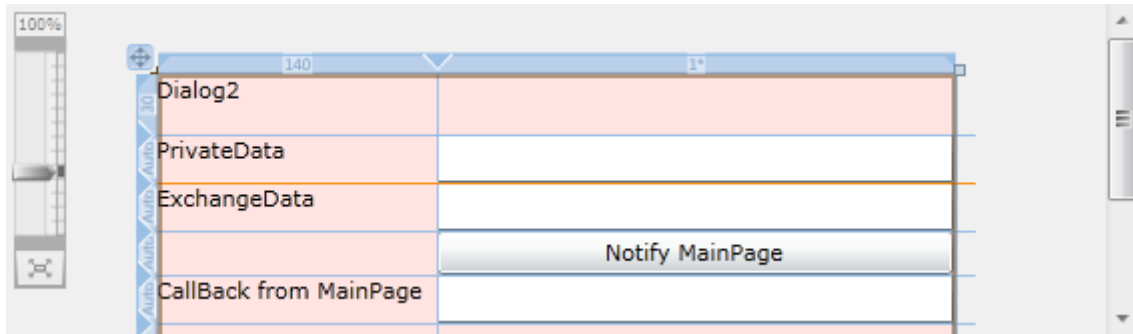
        private void CallbackFromMainPage(object data)
        {
            CallBackData = data;
        }
    }
}
```



## View von Dialog2

Fügen Sie der View wie im folgenden XAML einen Button zum Auslösen der Nachricht hinzu.

Die Daten aus dem Callback werden in einer TextBox angezeigt.



```
<Button
    Content="Notify MainPage"
    Command="{Binding CommandNotifyMain}"
    Grid.Row="3" Grid.Column="1"
/>

<TextBlock Text="CallBack from MainPage" Grid.Row="4" />
<TextBox
    Text="{Binding CallBackData, Mode=TwoWay}"
    Grid.Row="4"
    Grid.Column="1"
/>
```

## ViewModel von MainPage

Die Nachricht wird in MainPageViewModel über den Notifier abonniert.

Die Nachricht wird über das Property NotificationFromDialog2 visualisiert.

```
private string _notificationFromDialog2;
public string NotificationFromDialog2
{
    get { return _notificationFromDialog2; }
    set { SetPropertyValue(
        ref _notificationFromDialog2, value, () => NotificationFromDialog2); }
}

public MainPageViewModel()
{
    Notifier.Notify +=
        (sender, e) =>
        {
            if (e.Tag == "MyTag")
            {
                NotificationFromDialog2 = e.Data.ToString();

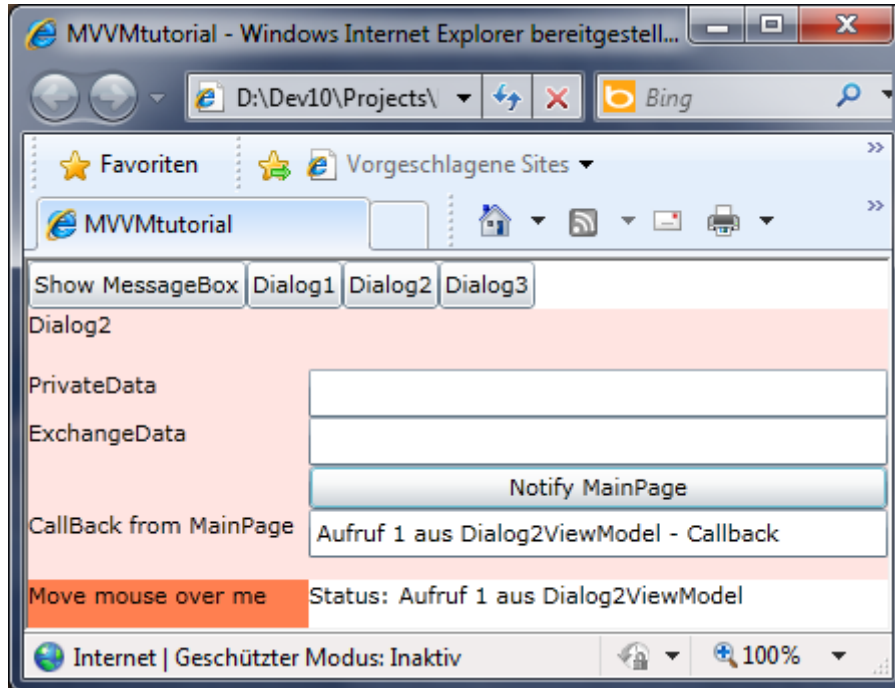
                if (e.Callback != null)
                    e.Callback(NotificationFromDialog2 + " - Callback");
            }
        };
    ...
}
```

## View von MainPage

Die Daten welche mit der Nachricht übermittelt wurden müssen nun noch in der View angezeigt werden.

```
<TextBlock Text="Status: " />
<TextBlock Text="{Binding NotificationFromDialog2}" />
</StackPanel>
```

## Resultat



## Ein Wizard

### ViewModel der Wizard- Seiten

Fügen Sie die Klassen Wizard1ViewModel und Wizard2ViewModel zum Projekt hinzu. Diese leiten von ViewModelBase ab.

```
using Riba.MVVMLSL;

namespace MVVMtutorial
{
    public class Wizard1ViewModel: ViewModelBase
    {
    }
}
```

### Views der Wizard- Seiten

Fügen Sie dem Projekt 2 JPG- Bilder Ihrer Wahl hinzu und nennen Sie diese picture1.jpg und picture2.jpg. Die Build Action der 2 Bilder muss Resource sein.

Fügen Sie 2 UserControls Wizard1 und Wizard2 zum Projekt hinzu. Der DataContext wird auf Wizard1/2ViewModel gesetzt.

Beiden UserControls fügen Sie ein ViewEventHandlerControl hinzu, es stellt die Event- Handler für ShowMessageBox, OpenFileDialog und LoadChildWindow zur Verfügung.

Beiden UserControls fügen Sie eine an ExchangeData und eine an PrivateData gebundene TextBox hinzu. Mode ist TwoWay.

Beiden UserControls fügen Sie das entsprechende Bild, welches aus der lokalen Ressource geladen wird hinzu.

```
<UserControl
    x:Class="MVVMtutorial.Wizard1"
    ...
    xmlns:MVVMtutorial="clr-namespace:MVVMtutorial"
    xmlns:MVVMLSL="clr-namespace:Riba.MVVMLSL;assembly=RibaMVVMLSL"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400"
>
    <UserControl.DataContext>
        <MVVMtutorial:Wizard1ViewModel />
    </UserControl.DataContext>

    <Grid x:Name="LayoutRoot" Background="Beige">
        <MVVMLSL:ViewEventHandlerControl />

        <Border BorderBrush="PaleGoldenrod" BorderThickness="20">
            <StackPanel>
                <TextBlock Text="Wizard Page 1" />
                <TextBox Text="{Binding ExchangeData, Mode=TwoWay}" />
                <TextBox Text="{Binding PrivateData, Mode=TwoWay}" />
                <Image
                    Source="/MVVMtutorial;component/picture1.jpg"
                    Stretch="Fill"
                />
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

## Beispiel einer View



## Die Logik des Wizard im View- Model

Ergänzen Sie Dialog3ViewModel mit einem CommandBack und einem CommandForward. Im Execute-Handler der Commands wird abwechselungsweise Wizard1 beziehungsweise Wizard2 mittels RaiseNavigateToView geladen. Initial wird Wizard1 angezeigt.

```
using System;
using Riba.MVVMLSL;

namespace MVVMtutorial
{
    public class Dialog3ViewModel : ViewModelBase
    {
        private Type _currentViewType;
        public RelayCommand CommandBack { get; private set; }
        public RelayCommand CommandForward { get; private set; }

        public Dialog3ViewModel()
        {
            CommandBack = new RelayCommand(OnNavigateBack);
            CommandForward = new RelayCommand(OnNavigateForward);

            NavigatorLoaded += Dialog3ViewModel_NavigatorLoaded;
        }

        void Dialog3ViewModel_NavigatorLoaded(object sender, EventArgs e)
        {
            CommandForward.Execute(null);

            CommandBack.IsEnabled = true;
            CommandForward.IsEnabled = true;
        }

        private void OnNavigateBack(object handler)
        {
            _currentViewType =
                _currentViewType == typeof(Wizard1) ? typeof(Wizard2) : typeof(Wizard1);

            RaiseNavigateToView(
                new NavigateToViewEventArgs(_currentViewType, AnimationMode.Backward));
        }

        private void OnNavigateForward(object handler)
        {
            _currentViewType =
                _currentViewType == typeof(Wizard1) ? typeof(Wizard2) : typeof(Wizard1);
            RaiseNavigateToView(
                new NavigateToViewEventArgs(_currentViewType, AnimationMode.Forward));
        }
    }
}
```

### Wizard in Dialog3- View einbetten

Fügen Sie in Dialog3.xaml zwei weitere Zeilen hinzu. In die erste neue Zeile wird ein NavigatorControl platziert, in die letzte Zeile 2 Buttons. Einer für Zurück und einer für Weiter.

```
<Grid>
  <Grid.RowDefinitions>
    ...

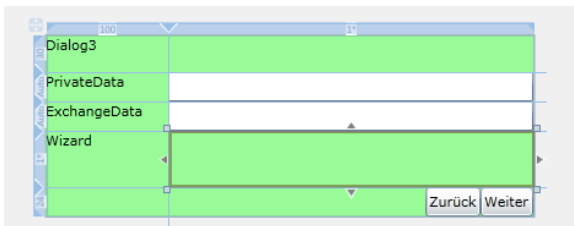
    <RowDefinition Height="*" />
    <RowDefinition Height="24" />
  </Grid.RowDefinitions>

  ...

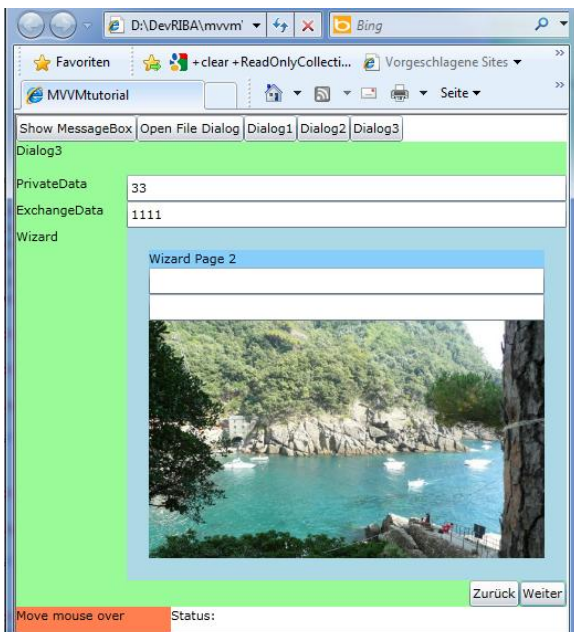
  <TextBlock Text="Wizard" Grid.Row="3" />
  <MVVMLSL:NavigatorControl Grid.Row="3" Grid.Column="1" />

  <StackPanel
    Orientation="Horizontal"
    HorizontalAlignment="Right"
    Grid.Row="4" Grid.Column="1"
  >
    <Button Content="Zurück" Command="{Binding CommandBack}" />
    <Button Content="Weiter" Command="{Binding CommandForward}" />
  </StackPanel>
</Grid>
```

Der Dialog sieht danach in etwa wie folgt aus.



Der fertige Wizard



## IsDirty

Um zu verhindern, dass ein Benutzer ungewollt seine im Dialog erfassten Daten verliert, stellt ViewModelBase das Property IsDirty zur Verfügung. Dieses muss bei Bedarf bei jeder Änderung der Daten (Model) auf true gesetzt werden. Wenn IsDirty = true ist, wird der Benutzer bei der Wahl eines weiteren Dialogs (RaiseNavigateToView) automatisch gefragt, ob er seine Änderungen zuerst noch speichern will.

### PropertyChanged

Am einfachsten ist es IsDirty im PropertyChanged- Event zu setzen. Dieser tritt bei jeder Änderung des ViewModel- Contexts auf. Erweitern Sie Dialog1ViewModel wie folgt.

```
public Dialog1ViewModel()
{
    ...
    CommandResetIsDirty = new RelayCommand
    (
        cmdPara =>
        {
            PropertyChanged -= Dialog1ViewModel_PropertyChanged;
            IsDirty = false;
            PropertyChanged += Dialog1ViewModel_PropertyChanged;
        }
    ){IsEnabled = true};

    PropertyChanged += Dialog1ViewModel_PropertyChanged;
}

void Dialog1ViewModel_PropertyChanged(
    object sender, System.ComponentModel.PropertyChangedEventArgs e)
{
    IsDirty = true;
}
```

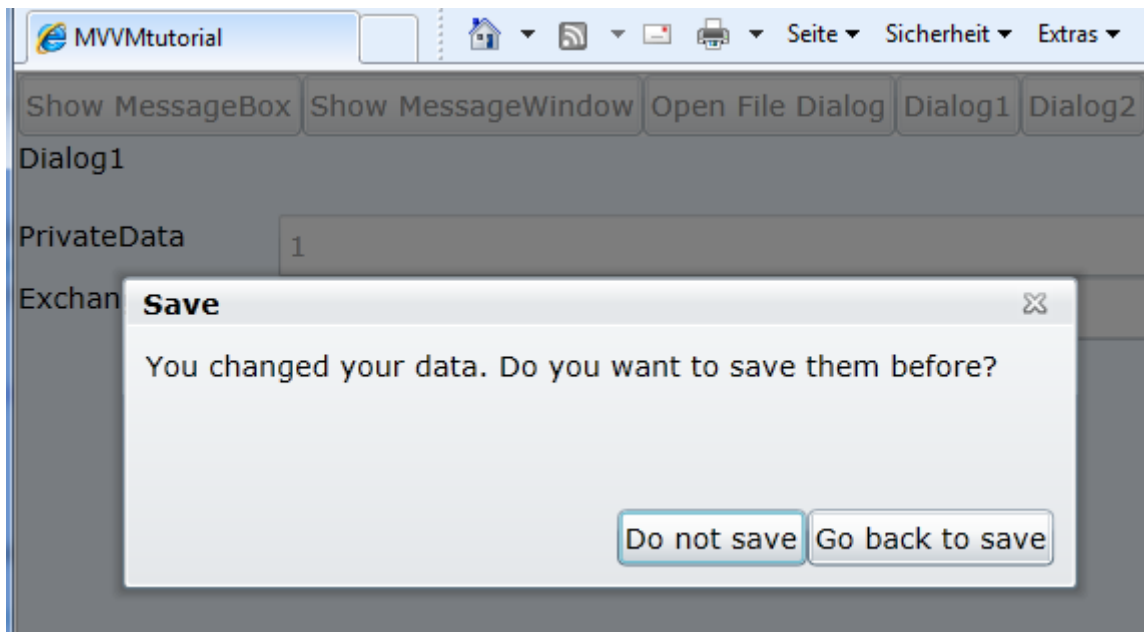
### Zurücksetzen

IsDirty wird beispielsweise nach dem Speichern der geänderten Daten zurückgesetzt. In unserem Beispiel haben wir einen Command der das simuliert. Damit IsDirty im Beispiel nicht gleich wieder auf true gesetzt wird, da jedes Ändern eine Property's einen PropertyChanged- Event zur Folge hat, wird der Event-Handler von PropertyChanged zuvor entfernt und nach dem Reset wieder installiert.

In RIA.NET Anwendungen wird beispielsweise an eine Subklasse von System.ServiceModel.DomainServices.Client.Entity gebunden. Diese hat ebenfalls ein passendes PropertyChanged- Event. (siehe auch Advanced Tutorial)

### Und so sieht's aus

Öffnen Sie Dialog1 und geben Sie im Feld PrivateData etwas ein. Danach klicken Sie auf <Dialog2>. Es erscheint ein MessageWindow mit der Frage, ob Sie die Daten zuerst speichern wollen.



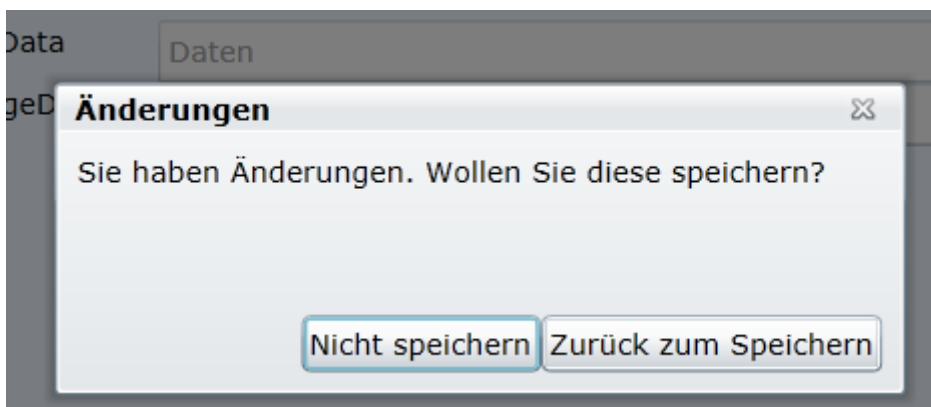
Drücken Sie <Do not save> um zum nächsten Dialog zu kommen, ohne die geänderten Daten zu speichern. Drücken Sie <Go back to save> um im aktuellen Dialog die geänderten Daten noch zu speichern.

### Eigene Texte

Im folgenden Beispiel sehen Sie wie Titel, Nachricht und Caption der Buttons angepasst werden können. Die Verwendung des Attributes `IsDirtyQuestionWindowState` über das der `MessageBox` ein beliebiger Style zugewiesen werden kann finden Sie im Advanced Tutorial.

```
<MVVMLSL:NavigatorControl
  Grid.Row="1"
  IsDirtyQuestion="Sie haben Änderungen. Wollen Sie diese speichern?"
  IsDirtyQuestionCaption="Änderungen"
  ContinueNavigationButtonCaption="Nicht speichern"
  CancelNavigationButtonCaption="Zurück zum Speichern"
/>
```

### Und so sieht's aus



## Fortgeschrittene I: Details in ChildWindow

Für die folgenden Beispiele erstellen Sie eine neue Silverlight- Anwendung mit der Bezeichnung MVVMtutorialAdvanced.

Über einen Menu- Button <Autoren> soll ein Dialog in ein NavigatorControl geladen werden.

Im Dialog werden Autoren mit Nachname und Id aufgelistet.

Der Benutzer kann Autoren ändern, hinzufügen und löschen.

Die Details eines Autoren der geändert oder hinzugefügt wird werden in einem ChildWindow bearbeitet und gespeichert.

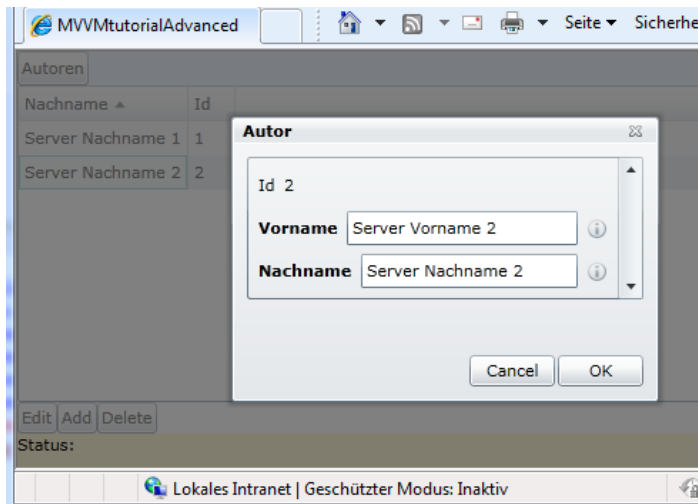
Die Anwendung holt sich die Daten (**Model**) im Design- Modus via Attrappen (Mock) und sonst über RIA-NET Services.

Das zu verwendende Model (DomainService) wird den **ViewModel** via Dependency Injection (Unity) zur Verfügung gestellt.

Das Binding der View an das ViewModel geschieht mittels eines View- Model- Locators.

Die Daten werden nicht in eine Datenbank sondern in einfache Collections des DomainServices gespeichert. Beim Start des DomainServices werden Dummy- Daten geladen. Dieser Ansatz ist ein erster Schritt in Richtung Repository Pattern. Der DomainService soll sich nicht um das Persistieren kümmern.

### Ansicht der fertigen Anwendung

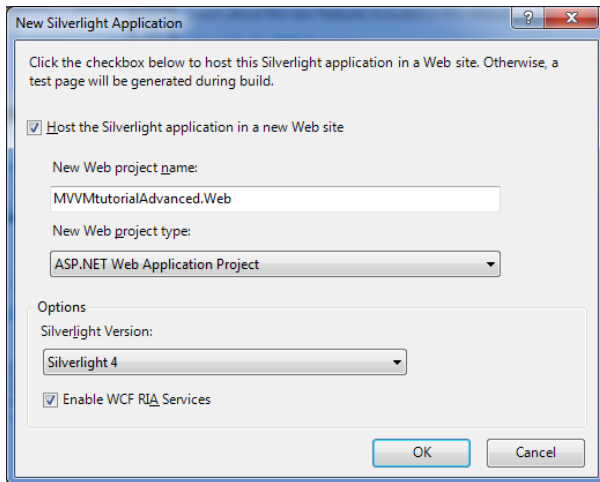




## Projekt

Erstellen Sie ein neues Silverlight Application Projekt.

Die Anwendung soll diesmal in einer Website gehostet werden und die RIA- Service werden benötigt.



Referenzieren Sie RibaMVVMLSL.

## Data Transfer Objects

Die Entität AuthorDTO wird im Verlauf des Beispiels verwendet. Mit der Endung DTO wird verdeutlicht, dass es sich nicht um die Entität aus der Datenbank handelt, sondern um eine Kopie derselben für den Datenaustausch zwischen GUI und Server, ein Data Transfer Object.

### AuthorDTO

Fügen Sie dem Webprojekt eine Klasse AuthorDTO hinzu mit einem Property Id und den Eigenschaften FirstName und LastName.

Für die Verwendung mit RIA.NET muss Id mit dem Attribut [Key] dekoriert werden.

FirstName und LastName werden als Muss- Felder von maximal 20 Buchstaben deklariert (Required).

Gültige Eingaben sind nur A-Z und Zahlen (RegularExpression).

Als Kolonnenüberschrift soll Vorname bzw. Nachname verwendet werden (Display.Name).

Der Vorname soll nicht automatisch angezeigt werden (Display.AutoGenerateField).

Bei automatischer Anzeige in einem DataGrid, soll die Kolonne ‚Id‘ am Schluss aufgeführt werden (Display.Order).

```
using System.ComponentModel.DataAnnotations;

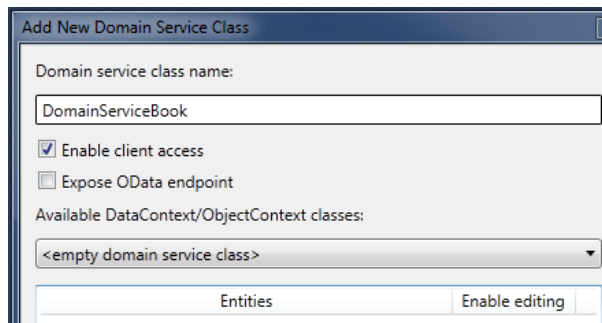
namespace MVVMtutorialAdvanced.Web
{
    public class AuthorDTO
    {
        [Key]
        [Display(Order = 3)]
        public int Id { get; set; }

        [Required(AllowEmptyStrings = false, ErrorMessage = "Muss- Feld")]
        [StringLength(20, ErrorMessage = "Maximal 20 Buchstaben")]
        [RegularExpression("[A-Za-z0-9 ]*", ErrorMessage = "Nur A-z und Zahlen")]
        [Display(
            Name = "Vorname",
            Description = "Dies ist der Vorname",
            Prompt = "Prompt Vorname",
            Order = 2,
            AutoGenerateField = false)]
        public string FirstName { get; set; }

        [Required(AllowEmptyStrings = false, ErrorMessage = "Muss- Feld")]
        [StringLength(20, ErrorMessage = "Maximal 20 Buchstaben")]
        [RegularExpression("[A-Za-z0-9 ]*", ErrorMessage = "Nur A-z und Zahlen")]
        [Display(
            Name = "Nachname",
            Description = "Dies ist der Nachname",
            Prompt = "Prompt Nachname",
            Order = 1)]
        public string LastName { get; set; }
    }
}
```

## DomainService - serverseitig

Fügen Sie dem Webprojekt eine ‚Domain Service Class‘ hinzu und nennen Sie diese DomainServiceBook.



## Repository

Die Daten werden im folgenden Beispiel nicht aus einer Datenbank sondern aus dem flüchtigen Repository \_authors geladen. Das Repository wird im Konstruktor des Domain- Services erstellt und mit 2 Autoren bestückt.

```
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel.DomainServices.Hosting;
using System.ServiceModel.DomainServices.Server;

namespace MVVMtutorialAdvanced.Web
{
    [EnableClientAccess]
    public class DomainServiceBook : DomainService
    {
        private static List<AuthorDTO> _authors;

        public DomainServiceBook()
        {
            if (_authors == null)
            {
                _authors = new List<AuthorDTO>();

                var author = new AuthorDTO {
                    Id = 1,
                    FirstName = "Michael C",
                    LastName = "Feathers" };
                _authors.Add(author);

                author = new AuthorDTO {
                    Id = 2,
                    FirstName = "Martin",
                    LastName = "Fowler" };
                _authors.Add(author);
            }
        }
    }
}
```

## GetAuthors

Über die Methode GetAuthors werden alle Autoren aufgelistet. Die Autoren werden mittels select aus dem Repository in AuthorForListDTOs gelesen und als Array zurückgegeben.

```
public ICollection<AuthorDTO> GetAuthors()
{
    var q =
        from aut in _authors
        select aut;

    return q.ToArray();
}
```

## InsertAuthor

Sofern sich ein Autor nicht bereits im Repository befindet wird er zu diesem hinzugefügt. Ein neuer Autor erhält als Id eine Id um 1 höher als die bisher höchste Id.

```
public void InsertAuthor(AuthorDTO author)
{
    if (_authors.Contains(author))
        return;

    if (_authors.Count == 0)
        author.Id = 1;
    else
        author.Id = _authors.Max(a => a.Id) + 1;

    _authors.Add(author);
}
```

## GetAuthorById

Ein einzelner Autor wird für Updates und Delete mittels der privaten Methode GetAuthorById gelesen. Diese könnte bei Bedarf durchaus auch öffentlich gemacht werden.

```
private static AuthorDTO GetAuthorById(int id)
{
    var q =
        from aut in _authors
        where aut.Id == id
        select aut;

    return q.FirstOrDefault();
}
```

## DeleteAuthor

Der übergebene Autor wird im Repository gesucht und dann entfernt.

```
public void DeleteAuthor(AuthorDTO author)
{
    var a = GetAuthorById(author.Id);
    _authors.Remove(a);
}
```

## UpdateAuthor

Der übergebene Autor wird im Repository gesucht und dann angepasst.

```
public void UpdateAuthor(AuthorDTO author)
{
    var a = GetAuthorById(author.Id);
    a.FirstName = author.FirstName;
    a.LastName = author.LastName;
}
```

## DomainService – clientseitig (Model)

Damit wir das Model in unserem GUI sowohl bei Aufrufen des DomainServices als auch im Design- Mode beispielsweise in Blend verwenden können, erstellen wir zuerst ein Interface das alle Methoden aus dem DomainService abdeckt. Das Interface wird dann 2 Mal implementiert. Einmal mit dem RIA.NET DomainService und einmal als Mock für den Design- Mode.

Welches der Implementierungen zur Verwendung kommt, wird in einer noch zu erstellenden Klasse ViewModelLocator bestimmt. Dort wird der passende DomainService via Unity registriert.

Via Dependency- Injection wird beim Erstellen eines View- Models dann der jeweilige DomainService im Konstruktor übergeben.

Platzieren Sie das folgende Interface und die Klasse DomainServiceMock im Unterordner DomainServices im GUI.

### IDomainServiceBook

```
using System;
using System.Collections.Generic;
using MVVMtutorialAdvanced.Web;

namespace MVVMtutorialAdvanced.DomainServices
{
    public interface IDomainServiceBook
    {
        void GetAuthors(Action<IEnumerable<AuthorDTO>, Exception> callback);
        void InsertAuthor(AuthorDTO author, Action<Exception> callback);
        void DeleteAuthor(AuthorDTO author, Action<Exception> callback);
        void UpdateAuthor(AuthorDTO author, Action<Exception> callback);
    }
}
```

## DomainServiceBookMock

Eine erste Implementierung des Interfaces `IDomainServiceBook` erstellen wir für den Design- Mode. Hier werden beim Aufruf von `GetAuthors` zwei vorgegebenen Autoren zurückgegeben. Fügen Sie folgende Klasse ebenfalls in den Unterordner `DomainServices` ein.

```
using System;
using System.Collections.Generic;
using System.Linq;
using MVVMtutorialAdvanced.Web;

namespace MVVMtutorialAdvanced.DomainServices
{
    public class DomainServiceBookMock : IDomainServiceBook
    {
        public void GetAuthors(Action<IEnumerable<AuthorDTO>, Exception> callback)
        {
            var data = new List<AuthorDTO>();

            data.Add(new AuthorDTO {
                Id = 1, FirstName = "First 1", LastName = "Last 1"});
            data.Add(new AuthorDTO {
                Id = 2, FirstName = "First 2", LastName = "Last 2"});

            callback(data.AsEnumerable(), null);
        }

        public void InsertAuthor(AuthorDTO author, Action<Exception> callback)
        {
            if (callback != null)
                callback(null);
        }

        public void DeleteAuthor(AuthorDTO author, Action<Exception> callback)
        {
            if (callback != null)
                callback(null);
        }

        public void UpdateAuthor(AuthorDTO author, Action<Exception> callback)
        {
            if (callback != null)
                callback(null);
        }
    }
}
```

## DomainServiceBookLive

Die RIA-NET- Variante nennen wir DomainServiceBookLive. Auch diese fügen Sie in den Unterordner DomainServices ein.

Im Konstruktor wird zuerst ein neuer DomainServiceBook instanziierte. Die Instanz weisen wir einer privaten Instanz- Variablen zu.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel.DomainServices.Client;
using MVVMtutorialAdvanced.Web;

namespace MVVMtutorialAdvanced.DomainServices
{
    public class DomainServiceBookLive : IDomainServiceBook
    {
        private readonly DomainServiceBook _service;

        public DomainServiceBookLive()
        {
            _service = new DomainServiceBook();
        }
    }
}
```

## GetAuthors

Die Methode zum Auflisten aller Autoren ruft Load auf GetAuthorsQuery, welches vom DomainService zur Verfügung gestellt wird auf.

Auf der zurückgegebenen LoadOperation kann der Completed- Event abonniert werden.

Im Completed- Event muss der sender als LoadOperation vom Typ AuthorDTO gecasted werden.

Falls der Aufrufer von GetAuthors einen Callback mitgegeben hat, wird dieser mit den Daten oder allenfalls mit dem aufgetretenen Fehler aufgerufen.

```
private Action<IEnumerable<AuthorDTO>, Exception> _getAuthorsCallback;

public void GetAuthors(Action<IEnumerable<AuthorDTO>, Exception> callback)
{
    _getAuthorsCallback = callback;

    var query = _service.GetAuthorsQuery();
    var loadOperation = _service.Load(query);
    loadOperation.Completed += getAuthorsLoadOp_Completed;
}

void getAuthorsLoadOp_Completed(object sender, EventArgs e)
{
    if (_getAuthorsCallback == null)
        return;

    var lo = sender as LoadOperation<AuthorDTO>;

    if (lo == null)
        _getAuthorsCallback(null, new NullReferenceException("LoadOperation is null"));
    else if (lo.HasError)
        _getAuthorsCallback(null, lo.Error);
    else
        _getAuthorsCallback(lo.Entities.AsEnumerable(), null);
}
```

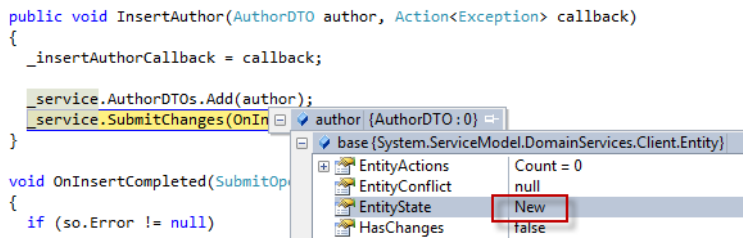
## InsertAuthor

Damit ein neuer Autor ins Repository eingefügt wird, muss er dem EntitySet AuthorDTOs des DomainServices hinzugefügt werden, dem neuen Autor wird dabei der EntityState ‚New‘ zugeordnet.

```
public void InsertAuthor(AuthorDTO author, Action<Exception> callback)
{
    _insertAuthorCallback = callback;

    _service.AuthorDTOs.Add(author);
    _service.SubmitChanges(OnInsertCompleted);
}

void OnInsertCompleted(SubmitOperation so)
{
    if (so.Error != null)
```

The image shows a code editor with the InsertAuthor method. The line `_service.SubmitChanges(OnInsertCompleted);` is highlighted. A tooltip for the `author` parameter is visible, showing its type as `base (System.ServiceModel.DomainServices.Client.Entity)`. The tooltip also displays properties: `EntityActions`, `EntityConflict`, `EntityState` (highlighted with a red box and showing the value `New`), and `HasChanges` (showing `false`).

EntityActions	Count = 0
EntityConflict	null
EntityState	New
HasChanges	false

Danach muss die Änderung via SubmitChanges übermittelt werden. Für alle hinzugefügten Entitäten wird dabei auf dem DomainService InsertAuthor aufgerufen.

Falls der Aufrufer von InsertAuthor einen Callback mitgegeben hat, wird dieser mit dem allenfalls aufgetretenen Fehler aufgerufen.

```
private Action<Exception> _insertAuthorCallback;

public void InsertAuthor(AuthorDTO author, Action<Exception> callback)
{
    _insertAuthorCallback = callback;

    _service.AuthorDTOs.Add(author);
    _service.SubmitChanges(OnInsertCompleted, null);
}

void OnInsertCompleted(SubmitOperation so)
{
    if (_insertAuthorCallback == null)
        return;

    _insertAuthorCallback(so.Error);
}
```



## DeleteAuthor

Löscht einen Autor aus dem Repository. Dazu muss er aus dem EntitySet AuthorDTOs des DomainServices mittels Remove entfernt werden.

Die Änderung wird via SubmitChanges übermittelt. Für alle gelöschten Entitäten wird dabei auf dem DomainService DeleteAuthor aufgerufen.

Falls der Aufrufer von DeleteAuthor einen Callback mitgegeben hat, wird dieser mit dem allenfalls aufgetretenen Fehler aufgerufen.

```
private Action<Exception> _deleteAuthorCallback;

public void DeleteAuthor(AuthorDTO author, Action<Exception> callback)
{
    _deleteAuthorCallback = callback;

    _service.AuthorDTOs.Remove(author);
    _service.SubmitChanges(OnDeleteCompleted, null);
}

void OnDeleteCompleted(SubmitOperation so)
{
    if (_deleteAuthorCallback == null)
        return;

    _deleteAuthorCallback(so.Error);
}
```

## UpdateAuthor

Speichert die Änderungen an einem Autor im Repository.

Die Änderung wird via SubmitChanges übermittelt. Für alle geänderten Entitäten wird dabei auf dem DomainService UpdateAuthor aufgerufen.

Falls der Aufrufer von UpdateAuthor einen Callback mitgegeben hat, wird dieser mit dem allenfalls aufgetretenen Fehler aufgerufen.

```
private Action<Exception> _updateAuthorCallback;

public void UpdateAuthor(AuthorDTO author, Action<Exception> callback)
{
    _updateAuthorCallback = callback;

    _service.SubmitChanges(OnSubmitCompleted, null);
}

void OnSubmitCompleted(SubmitOperation so)
{
    if (_updateAuthorCallback == null)
        return;

    _updateAuthorCallback(so.Error);
}
```

## ViewModelLocator

Erstellen Sie im Ordner VMLocator eine Klasse ViewModelLocator. Diese soll eine Instanz des UnityContainers in einer statischen Variable erhalten.

Im Konstruktor wird, falls sich die Anwendung im Designer (Blend oder Visual- Studio) befindet, der Mock- Domain- Service im Unity- Container registriert, ansonsten der Live- Domain- Service.

Über ein Property Find vom Typ Indexer wird über einen Value- Converter (IValueConverter) das im ConverterParameter angegebenen ViewModel zurückgegeben.

Wie im folgenden Beispiel für Author\_VM soll das Binding im XAML angegeben werden können:

```
DataContext="{Binding Source={StaticResource App_ViewModelLocator},  
    Converter={StaticResource App_ViewModelNameToViewModelConverter},  
    ConverterParameter=MVVMtutorialAdvanced.Author_VM}"
```

### ViewModelLocator

Die Idee zum ViewModelLocator stammt von John Papa. Anstelle von MEF verwende ich aber Unity, damit geht die Registrierung des zur Verwendung kommenden Domain- Services sehr einfach. Unity übergibt dann im Konstruktor des View- Models automatisch den entsprechenden Domain- Service.

URL: <http://johnpapa.net/silverlight/simple-viewmodel-locator-for-mvvm-the-patients-have-left-the-asylum/>

Der Domain- Service wird im folgenden Beispiel mit einem ContainerControlledLifetimeManager registriert, es wird also immer dieselbe Instanz verwendet.

```
using Microsoft.Practices.Unity;  
using MVVMtutorialAdvanced.DomainServices;  
using Riba.MVVMLSL;  
  
namespace MVVMtutorialAdvanced.VMLocator  
{  
    public class ViewModelLocator  
    {  
        private static IUnityContainer _container;  
  
        public Indexer Find { get; private set; }  
  
        public ViewModelLocator()  
        {  
            _container = new UnityContainer();  
  
            if (ViewModelBase.IsInDesignTool)  
                _container.RegisterType<IDomainServiceBook, DomainServiceBookMock>  
                    (new ContainerControlledLifetimeManager()); //Singleton  
            else  
                _container.RegisterType<IDomainServiceBook, DomainServiceBookLive>  
                    (new ContainerControlledLifetimeManager()); //Singleton  
  
            Find = new Indexer { IsShared = false, UnityContainer = _container};  
        }  
    }  
}
```

## Converter

Der Converter erhält den Namen des View- Models als Parameter vom Typ object. Im Parameter value ist eine als Anwendungs- Ressource abgelegte Instanz von ViewModelLocator angegeben. Auf dieser Instanz kann die Methode Find mit dem Namen des View- Models aufgerufen werden. Wegen eines Bugs in Silverlight 4 kann nicht direkt an einen String Indexer gebunden werden. (siehe John Papa)

```
using System;
using System.Windows.Data;

namespace MVVMtutorialAdvanced.VMLocator
{
    public class ViewModelNameToViewModelConverter : IValueConverter
    {
        public object Convert(
            object value, Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
        {
            var locator = (ViewModelLocator)value;
            var index = parameter.ToString(); //ViewModel- Name
            return locator.Find[index];
        }

        public object ConvertBack(
            object value, Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

## Indexer

Der Indexer gibt bei jeder Abfrage eine neue Instanz des gewünschten View- Models zurück. Wenn Sie immer dieselbe Instanz verwenden wollen, müssen Sie diese zuvor beispielsweise im Konstruktor des ViewModelLocator mit einem ContainerControlledLifetimeManager registrieren.

```
using System;
using Microsoft.Practices.Unity;

namespace MVVMtutorialAdvanced.VMLocator
{
    public class Indexer
    {
        public IUnityContainer UnityContainer { get; set; }

        private object GetViewModel(string viewModelTypeName)
        {
            var type = Type.GetType(viewModelTypeName);
            return UnityContainer.Resolve(type, viewModelTypeName);
        }

        public object this[string viewModel]
        {
            get { return GetViewModel(viewModel); }
        }
    }
}
```

## Ressource in App.xaml

Eine Instanz von ViewModelLocator und IndexerConverter werden in App.xaml als Ressource erstellt. Da wir später noch ein externes Dictionary als Ressource angeben, wird bereits ein MergedDictionary verwendet.

```
>
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary>
        <VMLocator:ViewModelLocator
          x:Key="App_ViewModelLocator" />
        <VMLocator:ViewModelNameToViewModelConverter
          x:Key="App_IndexerConverter" />
      </ResourceDictionary>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
</Application>
```

## MainPage

### ViewModel

Fügen Sie eine neue Klasse MainPage\_VM zum Projekt hinzu, diese leitet von ViewModelBase ab.

In der Main- Page wird ein Command erstellt, der den Dialog zur Anzeige der Autoren lädt.

Das Laden des Dialogs wird durch den Versand einer NavigateToView- Meldung an die View veranlasst.

Der Command soll von Anfang an benutzbar sein (IsEnabled).

```
using Riba.MVVMLSL;

namespace MVVMtutorialAdvanced
{
    public class MainPageVM : ViewModelBase
    {
        public RelayCommand Commandment { get; private set; }

        public MainPageViewModel()
        {
            CommandMenu = new RelayCommand
            (
                cmdPar =>
                {
                    NavigateToViewEventArgs p;

                    if (cmdPar.ToString() == "Authors")
                        p = new NavigateToViewEventArgs(
                            typeof(Author), AnimationMode.Forward);
                    else
                        p = null;

                    RaiseNavigateToView(p);
                }
            ) { IsEnabled = true };
        }
    }
}
```

## View

Der DataContext der View wird mittels der Application Ressource App\_ViewModelLocator geladen.

Die View beinhaltet einen Menu- Bar mit dem Button 'Autoren' dem der Command CommandMenu zugewiesen wird.

Der Dialog für die Autoren soll via ein NavigatorControl geladen werden. Damit das NavigatorControl zuoberst in der Z- Order liegt, wird es am Schluss des Grids definiert.

```
<UserControl
  x:Class="MVVMtutorialAdvanced.MainPage"
  ...
  xmlns:MVVMLSL="clr-namespace:Riba.MVVMLSL;assembly=RibaMVVMLSL"
  mc:Ignorable="d" d:DesignHeight="200" d:DesignWidth="400"
  DataContext= "{Binding
    Source={StaticResource App_ViewModelLocator},
    Converter={StaticResource App_IndexerConverter},
    ConverterParameter=MVVMtutorialAdvanced.MainPage_VM}"
>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="24" />
      <RowDefinition />
      <RowDefinition Height="24" />
    </Grid.RowDefinitions>

    <StackPanel Orientation="Horizontal" Background="AliceBlue">
      <Button
        Content="Autoren"
        Command="{Binding CommandMenu}"
        CommandParameter="Authors"
      />
    </StackPanel>

    <StackPanel Grid.Row="2" Orientation="Horizontal" Background="Cornsilk">
      <TextBlock Text="Status: " />
    </StackPanel>

    <MVVMLSL:NavigatorControl Grid.Row="1" />
  </Grid>
</UserControl>
```

## Autor- Dialog

### ViewModel

Erstellen Sie eine neue Klasse Author\_VM welche von ViewModelBase ableitet.

Die Klasse publiziert zwei Properties welche bei Änderungen das PropertyChanged- Event auslösen. Verwenden Sie dazu im Setter SetPropertyValue. SetPropertyValue wird eine Referenz auf den bisherigen Wert, der neue Wert und eine Expression welche auf das entsprechende Property verweist übergeben.

Die Eigenschaft EntityList wird einem DataGrid im Dialog als Datenquelle (DataSoure) dienen. Sie ist vom Typ PagedCollectionView, welcher sehr einfach für Sortieren und Gruppieren verwendet werden kann.

Die Eigenschaft SelectedEntity soll an SelectedItem des DataGrids gebunden werden. Hier steht der aktuell selektierte Author zur Verfügung.

Für die spätere Verwendung wir der im Konstruktor via Dependency Injection (Unity) übergebene DomainService in einer privaten Variablen gesichert.

```
using System;
using System.ComponentModel;
using System.Windows.Data;
using MVVMtutorialAdvanced.DomainServices;
using MVVMtutorialAdvanced.Web;
using Riba.MVVMLSL;

namespace MVVMtutorialAdvanced
{
    public class Author_VM : ViewModelBase
    {
        private PagedCollectionView _entityList;
        public PagedCollectionView EntityList
        {
            get { return _entityList; }
            set { SetPropertyValue(ref _entityList, value, () => EntityList); }
        }

        private AuthorDTO _selectedEntity;
        public AuthorDTO SelectedEntity
        {
            get { return _selectedEntity; }
            set { SetPropertyValue(
                ref _selectedEntity, value, () => SelectedEntity); }
        }

        private readonly IDomainServiceBook _domainService;

        public Author_VM(IDomainServiceBook domainService)
        {
            _domainService = domainService;

            GetItems();
        }
    }
}
```

## GetItems

In GetItems werden die Autoren gelesen.

Das Resultat von GetAuthors aus dem Domain- Service wird in einem anonymen Callback zuerst nach dem Nachnamen sortiert und dann EntityList zugeordnet.

Sofern Autoren gefunden werden, wird der erste Autor in der Liste selektiert.

```
private void GetItems()
{
    _domainService.GetAuthors
    (
        (data, error) =>
        {
            if (error != null)
            {
                RaiseShowMsgBox(new ShowMsgBoxEventArgs(error.Message, "Fehler"));
                return;
            }

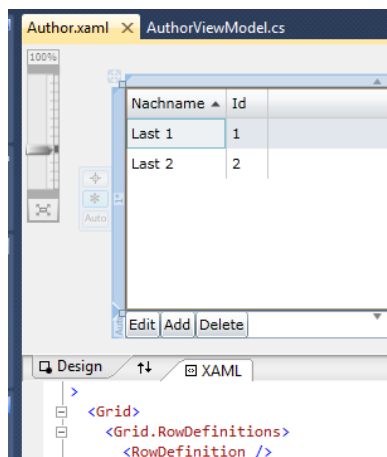
            var pagedCollectionView = new PagedCollectionView(data);
            pagedCollectionView.SortDescriptions.Add(
                new SortDescription("LastName", ListSortDirection.Ascending));

            EntityList = pagedCollectionView;

            if (EntityList.Count > 0)
                SelectedItem = EntityList[0] as AuthorDTO;
        }
    );
}
```

## View

Fügen Sie ein neues UserControl mit dem Namen Author.xaml hinzu. Der Dialog soll wie folgt aussehen.



Im Designer sehen Sie bereits die Mock- Daten, welche über DomainServiceBookMock zur Verfügung gestellt werden.

Auch die Sortierung nach dem Nachnamen ist bereits erfolgt.

Der Vorname wird wegen dem Display- Attribute ‚AutoGenerateField = false‘ nicht angezeigt!



## Author.Xaml

Fügend Sie für das DataGrid den Namespace

,http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk' hinzu.

Der DataContext der View wird mittels der Application Ressource App\_ViewModelLocator geladen.

Den Buttons für Edit, Add und Delete wird später noch der passende Command zugeordnet.

```
<UserControl
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  x:Class="MVVMtutorialAdvanced.Author"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignHeight="200" d:DesignWidth="400"
  DataContext="{Binding
    Source={StaticResource App_ViewModelLocator},
    Converter={StaticResource App_IndexerConverter},
    ConverterParameter=MVVMtutorialAdvanced.Author_VM}"
>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

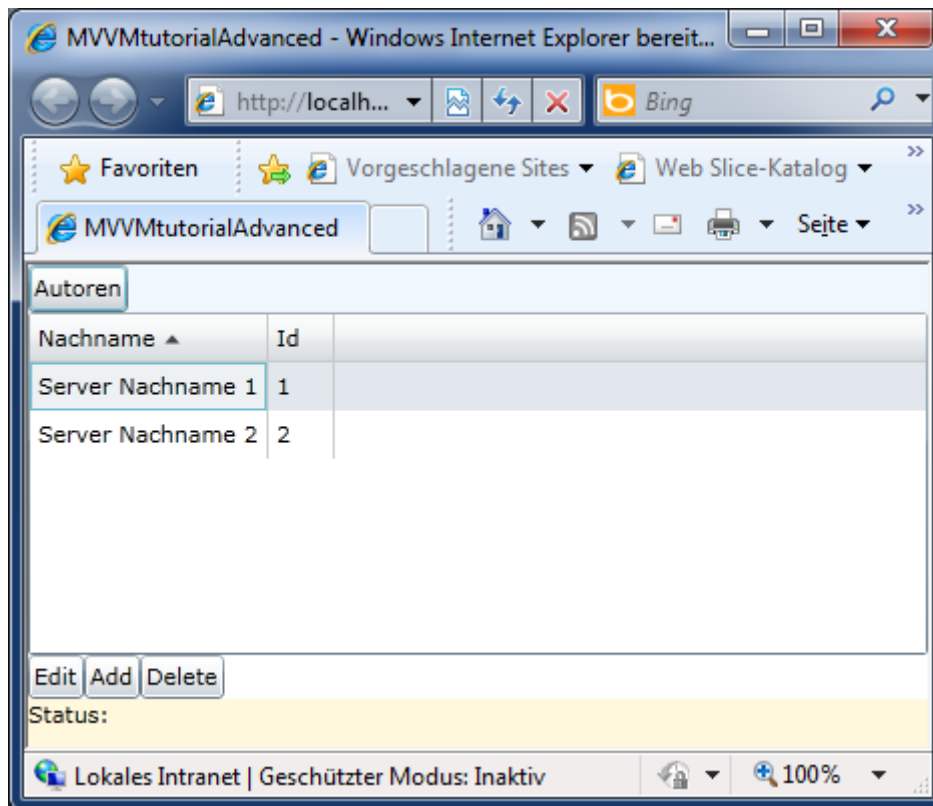
    <sdk:DataGrid
      ItemsSource="{Binding EntityList}"
      SelectedItem="{Binding SelectedEntity, Mode=TwoWay}"
      AutoGenerateColumns="True"
      RowDetailsVisibilityMode="VisibleWhenSelected"
      AIsReadOnly="True"
    />

    <StackPanel Orientation="Horizontal" Grid.Row="1">
      <Button Content="Edit" />
      <Button Content="Add" />
      <Button Content="Delete" ./>
    </StackPanel>
  </Grid>
</UserControl>
```

## Test

Starten Sie die Anwendung und Klicken Sie auf den Button zum Laden der Autoren.

Das Resultat sollte wie folgt aussehen.



## Autor- Details

TODO

## Bücher- Dialog

TODO

## Unit Test

TODO

## Eigene Animation implementieren

TODO

Laden einer View und animierte Anzeige im NavigationControl

### MainPage

