PRISM 4.0 TRAINING KIT

Hands-On Lab

View-Based Navigation

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277958302)

[Exercise 1 – Implementing a Basic View-Based Navigation Scenario 4](#_Toc277958303)

[Task 1 – Adding the Code to Perform View-Based Navigation 4](#_Toc277958304)

[Task 2 – Configuring Your Application 6](#_Toc277958305)

[Verification 6](#_Toc277958306)

[Exercise 2 - Passing Parameters during Navigation 9](#_Toc277958307)

[Task 1 – Creating the query string to pass parameters to a target view 9](#_Toc277958308)

[Task 2 – Receiving the parameters in the target view 11](#_Toc277958309)

[Verification 12](#_Toc277958310)

[Exercise 3 – Confirming and Cancelling Navigation 16](#_Toc277958311)

[Task 1 – Handling Navigation Confirmation 16](#_Toc277958312)

[Task 2 – Setting up the View’s interaction behavior 18](#_Toc277958313)

[Exercise 4 - Using the Navigation Journal 21](#_Toc277958314)

[Task 1 – Adding the back and forward buttons 21](#_Toc277958315)

[Task 2 - Using the CanGoBack and CanGoForward 27](#_Toc277958316)

Overview

As the user interacts with a rich client application, its user interface (UI) will be continuously updated to reflect the current task and data that the user is working on. The UI may undergo considerable changes over time as the user interacts with and completes various tasks within the application. The process by which the application coordinates these UI changes is often referred to as navigation.

UI updates can be accomplished by adding or removing elements from the application's visual tree, or by applying state changes to existing elements within the visual tree. WPF and Silverlight are very flexible platforms, and it is often possible to implement a particular navigation scenario using either of these two approaches. However, the approach that will be most appropriate for your application depends on multiple factors.

Prism differentiates between the two styles of navigation: Navigation accomplished via state changes to existing controls in the visual tree is referred to as *state-based navigation*. Navigation accomplished via the addition or removal of elements from the visual tree is referred to as *view-based navigation*. Prism provides guidance on implementing both styles of navigation, focusing on the case where the application is using the Model-View-ViewModel (MVVM) pattern to separate the UI (encapsulated in the view) from the presentation logic and data (encapsulated in the view model). This Hands-On Lab will focus on explain the view-based navigation which make use of the Prism API.

Exercise 1 – Implementing a Basic View-Based Navigation Scenario

Task 1 – Adding the Code to Perform View-Based Navigation

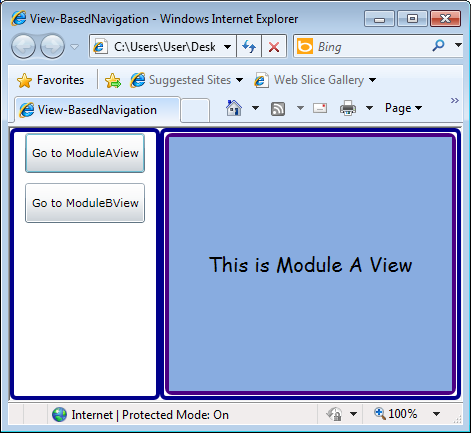
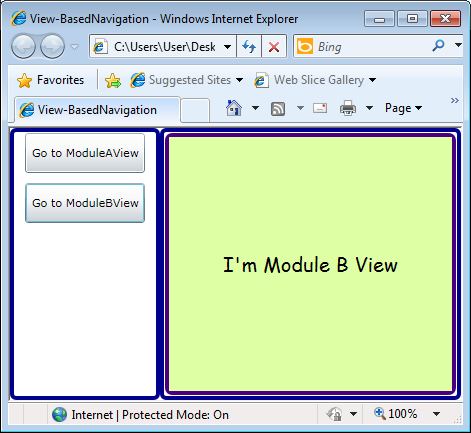
* 1. In this task, you will create some views, which will contain a button with navigation logic.
  2. Open the **View-BasedNavigation.sln** solution, located at the **\Navigation\Exercise 1\Begin\** folder of this Training Kit**.**
  3. In **ModuleA**, add a UserControl named **ModuleANavigationItemView** inside the **Views** folder.
  4. Add a button to navigate to **ModuleAView** view, as shown in the following code.
     1. XAML
     2. <Button Width="120" Height="40" Click="Button\_Click">Go to ModuleAView</Button>
  5. In the code behind for **ModuleANavigationItemView**, add the following using statements.
     1. C#
     2. using System.ComponentModel.Composition;
     3. using Microsoft.Practices.Prism.Regions;
     4. using Infrastructure;
  6. Decorate the **ModuleANavigationItemView** classwith the **Export** attribute, as shown in the following code.
     1. C#
     2. **[Export]**
     3. public partial class ModuleANavigationItemView : UserControl
     4. {
     5. ...
     6. }
  7. Add the following code in the body of the class.
     1. C#
     2. private static Uri moduleAViewUri = new Uri(ViewNames.ModuleAView, UriKind.Relative);
     3. [Import]
     4. public IRegionManager regionManager;
  8. In the handler for the Button's click event, add the following code. The **Region**.**RequestNavigate** method is responsible for navigating into a view. The **RegionManager.RequestNavigate** method calls **RequestNavigate** in the specified region.
     1. C#
     2. private void Button\_Click(object sender, RoutedEventArgs e)
     3. {
     4. regionManager.RequestNavigate(RegionNames.MainRegion, moduleAViewUri);
     5. }
  9. In **ModuleB**, add a new UserControl named **ModuleBNavigationItemView** inside the **Views** folder.
  10. Add a button to navigate to **ModuleBView**
      1. XAML
      2. <Button Width="120" Height="40" Click="Button\_Click">Go to ModuleBView</Button>
  11. In the code behind for **ModuleBNavigationItemView**, repeat the same procedures you’ve done in steps 4-7, but for ModuleBView. The resulting code should look like this.
      1. C#
      2. using System.Windows.Controls;
      3. using System.ComponentModel.Composition;
      4. using Microsoft.Practices.Prism.Regions;
      5. using Infrastructure;
      6. using System;
      7. using System.Windows;
      8. namespace ModuleB.Views
      9. {
      10. [Export]
      11. public partial class ModuleBNavigationItemView : UserControl
      12. {
      13. private static Uri moduleBViewUri = new Uri(ViewNames.ModuleBView, UriKind.Relative);
      14. [Import]
      15. public IRegionManager regionManager;
      17. public ModuleBNavigationItemView()
      18. {
      19. InitializeComponent();
      20. }
      21. private void Button\_Click(object sender, RoutedEventArgs e)
      22. {
      23. regionManager.RequestNavigate(RegionNames.MainRegion, moduleBViewUri);
      24. }
      25. }
      26. }

Task 2 – Configuring Your Application

In this task you will configure your application to perform the navigation scenario. You will add the views created in the previous task to a Menu region, and add AssemblyCatalogs to the bootstrapper for the involved modules.

* 1. In the Initialize method of **ModuleA**, add the following code to register the **ModuleANavigationItemView** in the MenuRegion.
     1. C#
     2. this.RegionManager.RegisterViewWithRegion(RegionNames.MenuRegion, typeof(ModuleANavigationItemView));
  2. In the Initialize method of **ModuleB**, add the following code to register the **ModuleBNavigationItemView** in the MenuRegion.
     1. C#
     2. this.RegionManager.RegisterViewWithRegion(RegionNames.MenuRegion, typeof(ModuleBNavigationItemView));
  3. In the **WorkshopBootstrapper** class, add the following code inside the **ConfigureAggregateCatalog** method.
     1. C#
     2. this.AggregateCatalog.Catalogs.Add(new AssemblyCatalog(typeof(ModuleA.ModuleA).Assembly));
     3. this.AggregateCatalog.Catalogs.Add(new AssemblyCatalog(typeof(ModuleB.ModuleB).Assembly));

Verification

* 1. Press F5 to run the solution.
  2. Click the **Go to ModuleAView** button. Verify that ModuleAView is displayed in the MainRegion.
     1. 
     2. ModuleAView being displayed in the MainRegion
  3. Click the **Go to ModuleBView** button. Verify that ModuleBView is displayed in the MainRegion.
     1. 
     2. ModuleBView being displayed in the MainRegion

Exercise 2 - Passing Parameters during Navigation

* 1. In this exercise you will learn how to pass parameters during navigation from a view to another. Parameters are passed using query strings when specifying the URL that you are navigating to. These parameters are used to provide the target view with the necessary information to populate it with data.

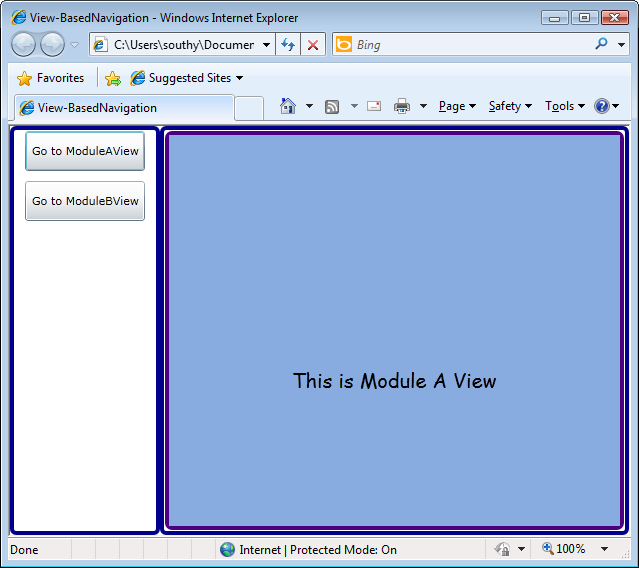
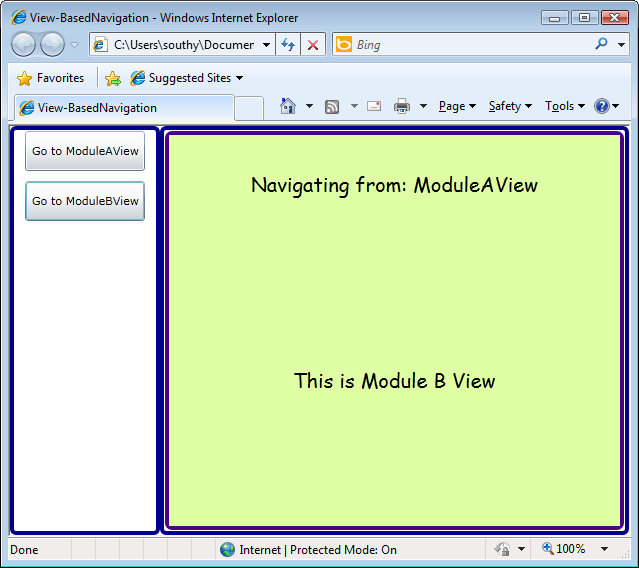
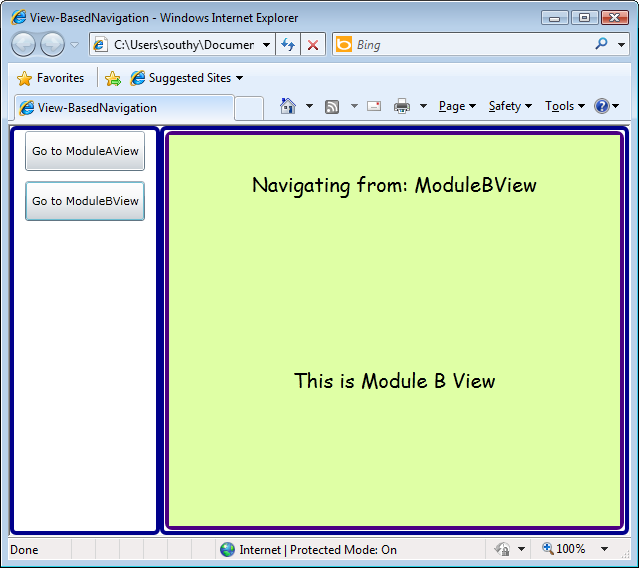
Task 1 – Creating the query string to pass parameters to a target view

* 1. In this task, you will create the query string that will be used to pass parameters to the target view.
  2. Open the **View-BasedNavigation.sln** solution, located at the **\Navigation\Exercise 2\Begin\** folder of this Training Kit**.**
  3. Open the ModuleANavigationItemView.xaml.cs file, located at ModuleA\Views.
  4. In the **ModuleANavigationItemView** class, remove the declaration of the moduleAViewUri variable, and create a new variable that will hold the navigation service. This variable will be of **INavigationService** type. The resulting code will look like the following.
     1. C#
     2. [Export]
     3. public partial class ModuleANavigationItemView : UserControl
     4. {
     5. [Import]
     6. public IRegionManager regionManager;
     7. **private IRegionNavigationService navigationService;**
     8. public ModuleANavigationItemView()
     9. {
     10. InitializeComponent();
     11. }
     12. private void Button\_Click(object sender, RoutedEventArgs e)
     13. {
     14. regionManager.RequestNavigate(RegionNames.MainRegion, moduleAViewUri);
     15. }
     16. }
  5. Edit the **Button\_Click** method, by getting a reference to the navigation service. This is shown in the following code.
     1. C#
     2. **this.navigationService = regionManager.Regions[RegionNames.MainRegion].NavigationService;**
  6. Add the following using statement at the top of the class. This namespace is needed to use the **UriQuery** class, provided by Prism to help specify and retrieve navigation parameters.
     1. C#
     2. using Microsoft.Practices.Prism;
  7. Create a new UriQuery instance, named query, which will be used for passing the parameters to the target view. As shown in the following code.
     1. C#
     2. **UriQuery query = new UriQuery();**
  8. We will pass the actual URI to the target view to display which was the previous URI. To do this, we will use the Add method of the UriQuery class that will receive the parameter name and its value, as arguments. In this case, the parameter name will be "previous" and the value (the current Uri) will be obtained from the CurrentEntry property of the Journal. The **Journal** property of the Navigation Service provides access to the navigation journal associated with the region. The Journal stores the navigation history of the corresponding region. The following code shows this.
     1. C#
     2. **if (this.navigationService.Journal.CurrentEntry != null)**
     3. **{**
     4. **query.Add("previous", this.navigationService.Journal.CurrentEntry.Uri.ToString());**
     5. **}**
     6. Notice that in the preceding code, the if clause is used for adding the parameter only if the journal contains an entry. This means that when the application is loaded and there is not any entry in the journal, the parameter will not be passed and the previous view text will not be displayed.
  9. Construct a new URI by appending the UriQuery to the name of the view. This is done in the following code.
     1. C#
     2. var uri = new Uri(ViewNames.ModuleAView + query.ToString(), UriKind.Relative);
  10. Finally, in the RequestNavigate method invocation, replace the Uri parameter with the new one created in the previous step. The final code of the Button\_Click method should look like the following.
      1. C#
      2. private void Button\_Click(object sender, RoutedEventArgs e)
      3. {
      4. this.navigationService = regionManager.Regions[RegionNames.MainRegion].NavigationService;
      5. UriQuery query = new UriQuery();
      6. if (this.navigationService.Journal.CurrentEntry != null)
      7. {
      8. query.Add("previous", this.navigationService.Journal.CurrentEntry.Uri.ToString());
      9. }
      10. var uri = new Uri(ViewNames.ModuleAView + query.ToString(), UriKind.Relative);
      11. regionManager.RequestNavigate(RegionNames.MainRegion, uri);
      12. }
  11. Repeat steps 2 to 8 of this task for **ModuleBNavigationItemView**.

Task 2 – Receiving the parameters in the target view

* 1. In this task, you will implement the **INavigationAware** interface, to be able to receive the
  2. Open the ModuleAView.xaml.cs file, located at ModuleA\Views.
  3. At the top of the file, add the following using statement.
     1. C#
     2. using Microsoft.Practices.Prism.Regions;
  4. Modify the class signature to make it implement the **INavigationAware** interface. This is shown in the following code.
     1. C#
     2. public partial class ModuleAView : UserControl, **INavigationAware**
  5. Implement the interface by clicking INavigationAware, pressing **CTRL+.** and then clicking **Implement Interface 'INavigationAware'** in the menu.
  6. Replace the existing code in the **IsNavigationTarget** method to return a true Boolean value. This method is called during navigation on all views in a region that are of the same type as the target view. The ability to navigate to an existing view is useful for a variety of reasons. It is often more efficient to update an existing view instead of replace it with a new instance of the same type. Similarly, activating an existing view, instead of creating a duplicate view, provides a more consistent user experience. In this case, when navigating from a view to the same one, will just update the label and do not create a new instance of the view.This is shown in the following code.
     1. C#
     2. public bool IsNavigationTarget(NavigationContext navigationContext)
     3. {
     4. return true;
     5. }
  7. Comment the existing code in the **OnNavigatedFrom** method. This method is used to execute code when navigating away from this view.
  8. Update the code of the **OnNavigatedTo** method, which is used to execute code when navigating to this view. In this method you will receive the parameters passed through the query string and perform the necessary tasks. following code shows how to receive the parameters.
     1. C#
     2. string previousPage = navigationContext.Parameters["previous"];
  9. Update the **Text** property of the **txtNavigatedFrom** text box in the view with the name of the previous view. Keep in mind that you have to update the Text property, only if the previous page is not null or empty. The resulting code of the OnNavigatedTo method is shown in the following code.
     1. C#
     2. public void OnNavigatedTo(NavigationContext navigationContext)
     3. {
     4. string previousPage = navigationContext.Parameters["previous"];
     5. if (!String.IsNullOrEmpty(previousPage))
     6. {
     7. this.txtNavigatedFrom.Text = string.Format("Navigating from: {0}", previousPage.Split('?')[0]);
     8. }
     9. }
     10. Notice that the previous parameter contains the query string, which is removed to retrieve only the view name.
  10. Repeat steps 1 to 8 for **ModuleBView**.

Verification

* 1. Press F5 to run the solution.
  2. Click the **Go to ModuleAView** button. Verify that ModuleAView is displayed in the MainRegion.
     1. 
     2. ModuleAView being displayed in the MainRegion
  3. Click the **Go to ModuleBView** button. Verify that ModuleBView is displayed in the MainRegion and that the "Navigating from: ModuleAView" message is displayed.
     1. 
     2. ModuleBView being displayed in the MainRegion, showing the navigating from message
  4. Click the **Go to ModuleBView** button again. The view is updated to show the "Navigating from: ModuleBView" message.
     1. 
     2. ModuleBView is updated, showing the new message

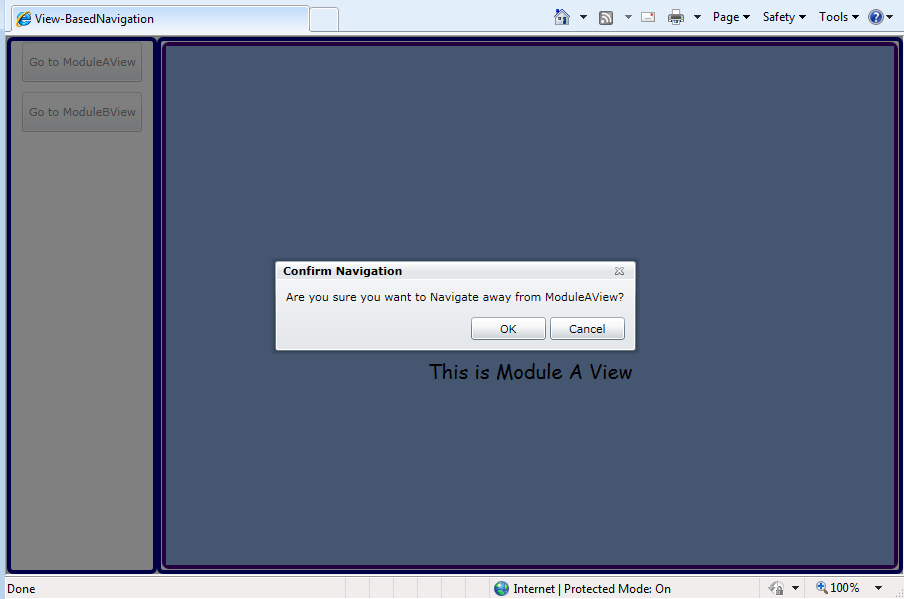
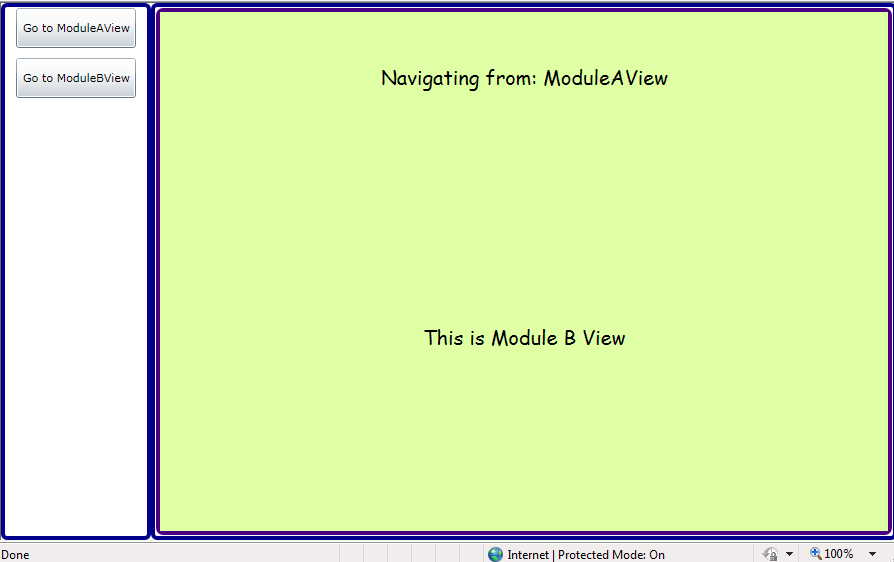
Exercise 3 – Confirming and Cancelling Navigation

* 1. Often, applications need to interact with users during a navigation operation to confirm or cancel said operation. In this exercise, you will add the necessary logic to prompt the user for confirmation before navigating away from a view.

Task 1 – Handling Navigation Confirmation

* 1. Open the **View-BasedNavigation.sln** solution located under the **\Navigation\Exercise 3\Begin\** directory of this Training Kit**.**
  2. Open the code behind file for **ModuleAView**, located inside the **Views** folder of the **ModuleA** project.
  3. Add a reference to the following assemblies, located in the **Lib** folder, to the **ModuleA** project.
     1. System.Windows.Interactivity.dll
     2. Microsoft.Practices.Prism.Interactivity.dll
     3. Microsoft.Expressions.Interactions.dll
  4. Add the following using statement to at the top of the file.
     1. C#
     2. using Microsoft.Practices.Prism.Interactivity.InteractionRequest;
  5. Update the class’ signature to implement the **IConfirmNavigationRequest** interface. As this interface also implements the **INavigationAware** interface, you can remove that interface from the class’ signature. The following code fragment shows the results.
     1. C#
     2. public partial class ModuleAView : UserControl, **IConfirmNavigationRequest**
  6. Add a property of type **IInteractionRequest**, which will be exposed to trigger the mechanism that prompts the user for confirmation on whether to navigate or not.
     1. C#
     2. private InteractionRequest<Confirmation> navigationInteractionRequest;
     3. public IInteractionRequest NavigationInteractionRequest
     4. {
     5. get
     6. {
     7. return this.navigationInteractionRequest;
     8. }
     9. }
  7. Instantiate the **InteractionRequest** in the class’ constructor and set the property as the view’s **DataContext**, as shown in the following code.
     1. C#
     2. public ModuleAView()
     3. {
     4. **this.navigationInteractionRequest = new InteractionRequest<Confirmation>();**
     5. **this.DataContext = this.NavigationInteractionRequest;**
     6. InitializeComponent();
     7. }
     8. **Note:** It is important to set only the property as the **DataContext** and not the view itself. This is because Prism’s navigation mechanism asks for confirmation from both the **View** and the **ViewModel** to perform the navigation. So, if the **DataContext** is the View itself, Prism considers that both the **View** and the **ViewModel** (view’s DataContext) implement the **IConfirmNavigationRequest** interface and the confirmation dialog is shown twice.
  8. Implement the **ConfirmNavigationRequest** method, from the **IConfirmNavigationRequest** interface, to raise the interaction request when navigating to a different view. This is shown in the following code.
     1. C#
     2. **public void ConfirmNavigationRequest(NavigationContext navigationContext, Action<bool> continuationCallback)**
     3. **{**
     4. **string targetPage = navigationContext.Uri.ToString().Split('?')[0];**
     5. **if (!String.IsNullOrEmpty(targetPage) && !targetPage.Contains("ModuleAView"))**
     6. **{**
     7. **this.navigationInteractionRequest.Raise**
     8. **(**
     9. **new Confirmation()**
     10. **{**
     11. **Content = "Are you sure you want to Navigate away from ModuleAView?",**
     12. **Title = "Confirm Navigation"**
     13. **},**
     14. **c =>**
     15. **{**
     16. **continuationCallback(c.Confirmed);**
     17. **}**
     18. **);**
     19. **}**
     20. **else**
     21. **{**
     22. **continuationCallback(true);**
     23. **}**
     24. **}**

Task 2 – Setting up the View’s interaction behavior

* 1. Open **ModuleAView’s** designer.
  2. Add the following namespace declarations.
     1. XAML
     2. xmlns:i=http://schemas.microsoft.com/expression/2010/interactivity
     3. xmlns:prism="http://www.codeplex.com/prism"
  3. Add a behavior to the view to prompt the user for confirmation when the interaction request is raised. The behavior should be added inside the as a child of the border control. Note that the SourceObject property (highlighted), binds to the instance of the **NavigationInteractionRequest** property created in the View.
     1. C#
     2. <Border BorderBrush="Indigo" BorderThickness="4" CornerRadius="5">
     3. **<i:Interaction.Triggers>**
     4. **<prism:InteractionRequestTrigger SourceObject="{Binding}">**
     5. **<prism:PopupChildWindowAction />**
     6. **</prism:InteractionRequestTrigger>**
     7. **</i:Interaction.Triggers>**
     8. ...
     9. **Note:** In case the ViewModel “owns” the **InteractionRequest**, you can simply bind to a named ViewModel property, as in regular MVVM.
  4. In the **View-BasedNavigation** project, add a reference to the **Microsoft.Expression.Interactions.dll** assembly, located inside the **Lib** folder.
     1. **Important:** The assembly you just added is not used by the project. The reason why you add it is because the assembly must be inside the .xap file that contains the view that prompts the confirmation dialog. You can try removing the assembly for the Shell project to see the error you will get.
  5. In Visual Studio, press **F5** to run the application.
  6. Click **Go to ModuleAView**. **ModuleAView** appears in the **MainRegion**.
  7. Click **Go to ModuleBView**. A confirmation message appears, as shown in the following figure.
     1. 
     2. Confirmation dialog before navigating away from ModuleAView
  8. Click **Cancel**. **ModuleAView** will continue to be displayed.
  9. Again, click **Go to ModuleBView**, and in the confirmation dialog click **OK**. You will navigate to **ModuleViewB**.
     1. 
     2. Navigating to ModuleViewB

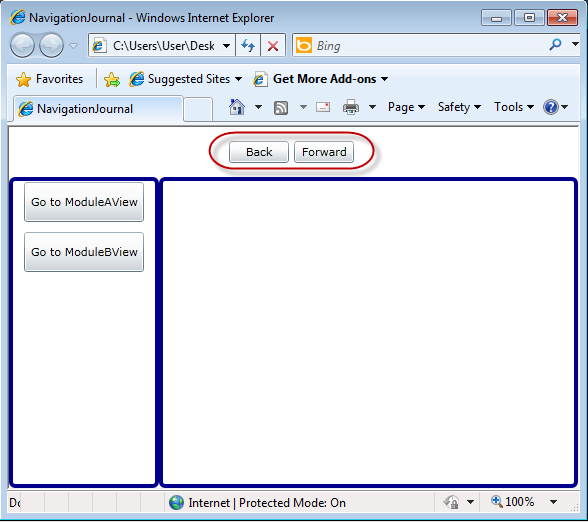
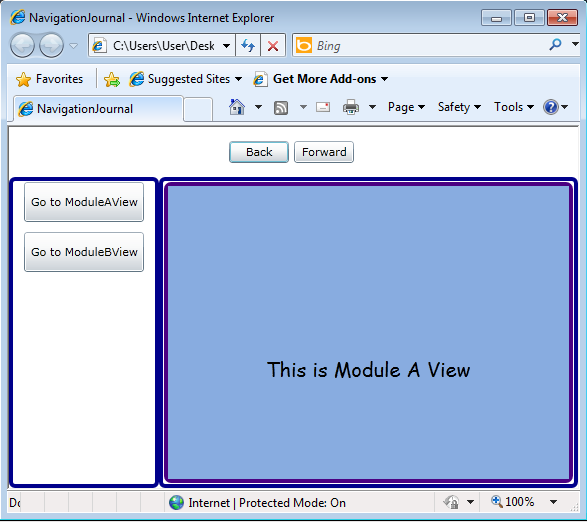
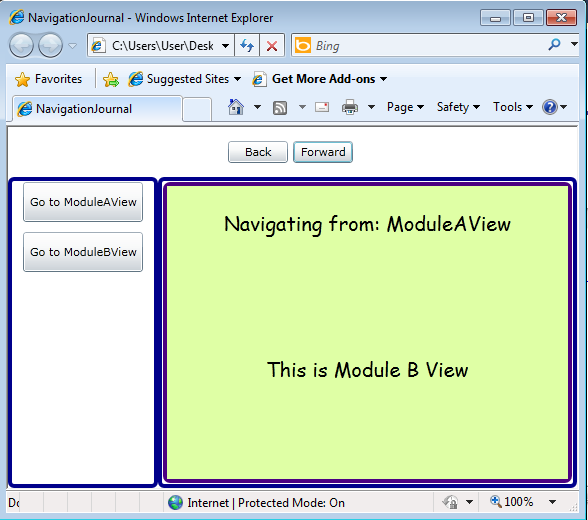
Exercise 4 - Using the Navigation Journal

* 1. In this exercise you will add a back and a forward button to the shell that will let you navigate backwards and forwards. To achieve this functionality Prism's Journal APIs will be used. For this exercise, the back and forward buttons will be placed on the top shell layer, but they could be easily located on any view.

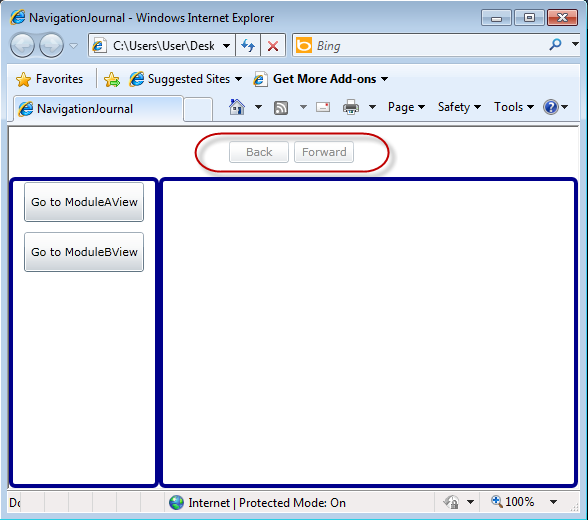
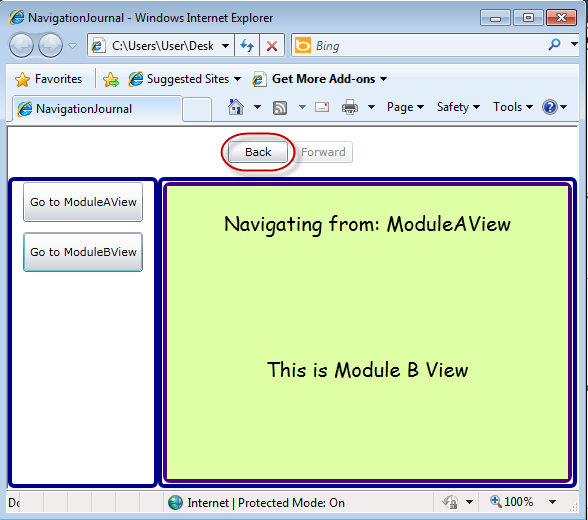
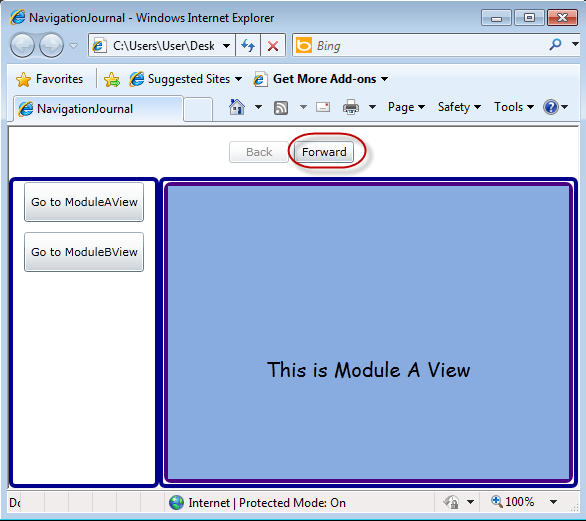
Task 1 – Adding the back and forward buttons

* 1. In this task, you will update the Shell view to show the Back and Forward buttons. Additionally, you will implement the commands that will handle the navigation in the **Shell** class.
  2. Open the **NavigationJournal.sln** solution located under the **\7-Navigation\Exercise 4\Begin\** folder of this Training Kit**.**
  3. Open the Shell.xaml.cs file, and add the following using statements.
     1. C#
     2. **using Microsoft.Practices.Prism.Modularity;**
     3. **using Microsoft.Practices.Prism.Regions;**
     4. **using Infrastructure;**
     5. **using Microsoft.Practices.Prism.Commands;**
     6. **using System.Collections.Specialized;**
  4. To implement the back and forward button behavior, you will use Commands. Add the following command members to the **Shell** class as shown on the following code.
     1. C#
     2. public partial class Shell : UserControl
     3. {
     4. **public DelegateCommand JournalBackCommand { get; set; }**
     5. **public DelegateCommand JournalForwardCommand { get; set; }**
     6. …
     7. }
  5. Create a private member in the **Shell** class to hold the instance of the region manager, as shown in the following code.
     1. C#
     2. public DelegateCommand JournalBackCommand { get; set; }
     3. public DelegateCommand JournalForwardCommand { get; set; }
     5. **private readonly IRegionManager regionManager;**
     6. In the following steps, the instance of the region manager will be used to retrieve the MainRegion and its navigation service.
  6. Update the Shell constructor to import an instance of the **IRegionManager**. This is shown in the following code.
     1. C#
     2. **[ImportingConstructor]**
     3. public Shell(**IRegionManager regionManager**)
     4. {
     5. InitializeComponent();
     6. **// store the region manager**
     7. **this.regionManager = regionManager;**
     8. }
  7. To simplify the exercise, the shell will be the View Model for its view; therefore you should set the data context as shown in the following code. In a real application, a ViewModel class should be created and set as a data context.
     1. C#
     2. [ImportingConstructor]
     3. public Shell(IRegionManager regionManager)
     4. {
     5. InitializeComponent();
     6. // store the region manager
     7. this.regionManager = regionManager;
     8. **// set datacontext**
     9. **this.DataContext = this;**
     10. }
  8. Add the following code to the **Shell** constructor to initialize the navigation commands. This code will use the **JournalBack** and **JournalForward** methods will be implemented in the following steps.
     1. C#
     2. [ImportingConstructor]
     3. public Shell(IRegionManager regionManager)
     4. {
     5. InitializeComponent();
     6. // store the region manager
     7. this.regionManager = regionManager;
     8. **// Initialize journal navigation commands**
     9. **this.JournalBackCommand = new DelegateCommand(this.JournalBack);**
     10. **this.JournalForwardCommand = new DelegateCommand(this.JournalForward);**
     11. // set datacontext
     12. this.DataContext = this;
     13. }
  9. Add the methods **JournalBack** and **JournalForward** methods that will handle the navigation. These methods use the Prism Journal API to go back and forward respectively. In the following steps, you will add the code necessary to retrieve the navigation service used to get the journal instance.
     1. C#
     2. **private void JournalBack()**
     3. **{**
     4. **// Navigate back on the main region**
     5. **this.MainRegionNavigationService.Journal.GoBack();**
     6. **}**
     8. **private void JournalForward()**
     9. **{**
     10. **// Navigate forward on the main region**
     11. **this.MainRegionNavigationService.Journal.GoForward();**
     12. **}**
  10. Add the **MainRegionNavigationService** property to the Shell class, that will retrieve the navigation service of the MainRegion, using the following code.
      1. C#
      2. public DelegateCommand JournalBackCommand { get; set; }
      3. public DelegateCommand JournalForwardCommand { get; set; }
      4. **private IRegionNavigationService MainRegionNavigationService**
      5. **{**
      6. **get**
      7. **{**
      8. **// Retrieve the navigation service associated with the MainRegion**
      9. **return this.regionManager.Regions[RegionNames.MainRegion].NavigationService;**
      10. **}**
      11. **}**
  11. Open the **Shell.xaml** file, to update the XAML to include the back and forward buttons.
  12. Update the Grid's column and row definition as follows, to add and additional row where the button will be located.
      1. XAML
      2. <Grid>
      3. **<Grid.RowDefinitions>**
      4. **<RowDefinition Height="50"/>**
      5. **<RowDefinition />**
      6. **</Grid.RowDefinitions>**
      7. **<Grid.ColumnDefinitions>**
      8. **<ColumnDefinition Width="150"/>**
      9. **<ColumnDefinition />**
      10. **</Grid.ColumnDefinitions>**

**...**

* 1. Update the content of the Grid as follows to add the new buttons, and update the row and column properties of the other controls. Notice that the Button's Command properties is bound to corresponding navigation commands.
     1. XAML
     2. <Grid.ColumnDefinitions>
     3. <ColumnDefinition Width="150"/>
     4. <ColumnDefinition />
     5. </Grid.ColumnDefinitions>
     6. **<StackPanel Grid.Row="0" Grid.ColumnSpan="2" VerticalAlignment="Center" HorizontalAlignment="Center" Orientation="Horizontal">**
     7. **<Button Margin="0,5,5,5" Command="{Binding JournalBackCommand}" Width="60">Back</Button>**
     8. **<Button Margin="0,5,5,5" Command="{Binding JournalForwardCommand}" Width="60">Forward</Button>**
     9. **</StackPanel>**
     10. **<Border BorderBrush="DarkBlue" Grid.Row="1" BorderThickness="4" CornerRadius="4"></Border>**
     11. **<ItemsControl prism:RegionManager.RegionName="MenuRegion"**
     12. **Grid.Column="0" Grid.Row="1" Margin="5,0,5,5" HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch"/>**
     13. **<Border Grid.Column="2" BorderBrush="DarkBlue" BorderThickness="4" CornerRadius="4" Grid.Row="1"></Border>**
     14. **<ContentControl prism:RegionManager.RegionName= "MainRegion"**
     15. **Grid.Column="1" Grid.Row="1" Margin="5,5,5,5" HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch"/>**
     16. **</Grid>**
  2. Press **F5** to run the application.
  3. Verify that the application has the new navigation buttons enabled. If you click the buttons now, nothing will happen as no navigation has been performed. In the following task, you will update the buttons to be disabled when navigation is not available.
     1. 
     2. Navigation buttons
  4. Click **Go to ModuleAView**,to navigate to ViewA, and thenclick **Go to ModuleBView** to navigate to ViewB.
  5. Click **Back** and verify that the main region navigates back to ViewA.
     1. 
     2. Back button navigating from ViewB to ViewA
  6. Click **Forward** and verify that the main region navigates to ViewB.
     1. 
     2. Forward button navigating from ViewA to ViewB
  7. Close the Browser.

Task 2 - Using the CanGoBack and CanGoForward

* 1. In this task you will update the code necessary to enable/disable the navigation button according to the **CanGoForward** and **CanGoBack** methods of the Journal. To do this, you will make use of the CanExecute methods of the navigation commands, but additional steps will be necessary to assure the region is already in place at the moment the CanExecute methods are queried by the UI.
  2. Open the Shell.xaml.cs file
  3. Update the Shell class constructor instantiate the commands specifying the CanExecute methods as follows.
     1. C#
     2. // Initialize journal navigation commands
     3. this.JournalBackCommand = new DelegateCommand(this.JournalBack**, this.CanNavigateBack**);
     4. this.JournalForwardCommand = new DelegateCommand(this.JournalForward**, this.CanNavigateForward**);
  4. Implement the **CanNavigateBack** and **CanNavigateForward** methods as in the following code.
     1. C#
     2. **private bool CanNavigateBack()**
     3. **{**
     4. **// Check if main region journal can go back**
     5. **return this.MainRegionNavigationService.Journal.CanGoBack;**
     6. **}**
     8. **private bool CanNavigateForward()**
     9. **{**
     10. **// Check if main region journal can go back**
     11. **return this.MainRegionNavigationService.Journal.CanGoForward;**
     12. **}**
     13. **Note:** Now the CanExecute methods are implementing using the Journal APIs. Nevertheless, if you run the application now you will get an exception, as at the time the **CanNavigateBack** and **CanNavigateForward** method are queried by the UI, the **MainRegionNavigationService** property is null because the MainRegion is not yet available in the RegionManager. You will solve this problem in the following tasks.
  5. Add the **isMainRegionAvailable** member to the **Shell** class, which will be used as a flag that indicates if the main region has been loaded, and therefore the **CanNavigateBack** and **CanNavigateForward** methods can be queried.
     1. C#
     2. public partial class Shell : UserControl
     3. {
     4. private readonly IRegionManager regionManager;
     5. **private bool isMainRegionAvailable = false;**
     6. **…**
  6. Update the **CanNavigateBack** and **CanNavigateForward** methods as follows, to make use of the **isMainRegionAvailable** flag.
     1. C#
     2. private bool CanNavigateForward()
     3. {
     4. **if (!this.isMainRegionAvailable)**
     5. **{**
     6. **return false;**
     7. **}**
     8. // Check if main region journal can go back
     9. return this.MainRegionNavigationService.Journal.CanGoForward;
     10. }
     12. private bool CanNavigateBack()
     13. {
     14. **if (!this.isMainRegionAvailable)**
     15. **{**
     16. **return false;**
     17. **}**
     18. // Check if main region journal can go back
     19. return this.MainRegionNavigationService.Journal.CanGoBack;
     20. }
  7. To set the value of the **isMainRegionAvailable** flagwhen theMainRegion is available, you should listen to the **CollectionChanged** event of the **RegionManager.Regions** collection.To do this, add the following code to the Shell's constructor.
     1. C#
     2. [ImportingConstructor]
     3. public Shell(IRegionManager regionManager)
     4. {
     5. InitializeComponent();
     6. // store the region manager
     7. this.regionManager = regionManager;
     8. **// listen for region changes to check when the main region is available**
     9. **this.regionManager.Regions.CollectionChanged += new NotifyCollectionChangedEventHandler(Regions\_CollectionChanged);**
     10. // Initialize journal navigation commands
     11. this.JournalBackCommand = new DelegateCommand(this.JournalBack, this.CanNavigateBack);
     12. this.JournalForwardCommand = new DelegateCommand(this.JournalForward, this.CanNavigateForward);
     13. // set datacontext
     14. this.DataContext = this;
     15. }
  8. Add the implementation of the **Regions\_CollectionChanged** method as shown in the following code. When the MainRegion is added, this method sets the **isMainRegionAvailable** to **true**, and it invokes the **RaiseCanExecuteChanged** methods of the navigation commands to let the UI know it should now re-query the CanExecute methods. Additionally, once the MainRegion is available, it listens to the **Navigated** event because every time the region navigates to other views the **CanGoBack** and **CanGoForward** properties of the Journal might change their value.
     1. C#
     2. **void Regions\_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e)**
     3. **{**
     4. **// onyl care if the main region was added**
     5. **if (e.Action == NotifyCollectionChangedAction.Add)**
     6. **{**
     7. **var region = e.NewItems[0] as IRegion;**
     8. **if (region != null && region.Name == RegionNames.MainRegion)**
     9. **{**
     10. **// set the flag to true**
     11. **this.isMainRegionAvailable = true;**
     12. **// Let the UI know it has to reevaluate the canExecute method of the commands**
     13. **this.JournalForwardCommand.RaiseCanExecuteChanged();**
     14. **this.JournalBackCommand.RaiseCanExecuteChanged();**
     15. **// start listening to all navigation events**
     16. **this.MainRegionNavigationService.Navigated += new EventHandler<RegionNavigationEventArgs>(MainRegionNavigationService\_Navigated);**
     17. **}**
     18. **}**
     19. **}**
  9. Add the implementation of the **MainRegionNavigationService\_Navigated** method as shown in the following code snippet. This method raises the **CanExecuteChanged** events of the navigation commands every time the MainRegion navigates to a view.
     1. C#
     2. **void MainRegionNavigationService\_Navigated(object sender, RegionNavigationEventArgs e)**
     3. **{**
     4. **// When a navigation occurs, let the UI know it has to reevaluate the canExecute method of the commands**
     5. **this.JournalForwardCommand.RaiseCanExecuteChanged();**
     6. **this.JournalBackCommand.RaiseCanExecuteChanged();**
     7. **}**
  10. Press **F5** to run the application.
  11. Verify that both navigation buttons are now disabled. This is because no navigation was performed, the Journal history is empty, and therefore the **CanGoBack** and **CanGoForward** of the Journal return false.
      1. 
      2. Navigation buttons disabled
  12. Click **Go to ModuleAView** to navigate to ViewA, and then click **Go to ModuleBView** to navigate to ViewB.
  13. Verify that the **Back** button is now enabled. This is because you can now navigate to the previous view.
      1. 
      2. The Back button gets enabled
  14. Click the **Back** button, and verify that you navigate to ViewA, and the **Forward** button is now enabled.
      1. 
      2. The Forward button gets enabled
  15. Click **Forward** to navigate to ViewB.
  16. Close the Browser.