Contents

Navigation is an essential aspect of any application. Punch provides the Navigator class that implements universal and configurable UI navigation for WPF, Silverlight and Windows Store apps.

Using the Navigator

Navigation support in Punch consists of two parts. The Punch INavigator interface and the default Navigator implementation. The Navigator works in conjunction with a conductor that manages the current active ViewModel.

We first need a visual container for where the navigation takes place. A visual container is simply a View with a ContentControl bound to the conductor's active item as illustrated below.

```
<UserControl x:Class="ShellView"
             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"
             d:DesignHeight="462"
             d:DesignWidth="857"
             mc:Ignorable="d">
    <Grid x:Name="LayoutRoot" Background="White">
        <ContentControl x:Name="ActiveItem" />
    </Grid>
</UserControl>
```

Once we have our container, we can put the conductor ViewModel behind it and initialize the corresponding Navigator as in the following example.

```
public class ShellViewModel : Conductor<object>
{
    private readonly INavigator _navigator;
    public ShellViewModel()
    {
        _navigator = new Navigator(this);
    }
}
```

Navigation itself is performed with one of the several overloads of NavigateToAsync, which takes the ViewModel type we wish to navigate to and an optional action to initialize the target ViewModel once it has been created. The actual instance of the target ViewModel gets created through the MEF container.

```
public async void Add()
{
    try
    {
        var nameEditor = _nameEditorFactory.CreateExport().Value;
        await _dialogManager.ShowDialogAsync(nameEditor, DialogButtons.OkCancel);
        await _navigator.NavigateToAsync<StaffingResourceDetailViewModel>(
            target => target.Start(nameEditor.FirstName, nameEditor.MiddleName, nameEditor.LastName));
    }
    catch (TaskCanceledException)
    {
        UpdateCommands();
    }
}
```

Guarding the current active ViewModel

Part of a good navigation scheme should be the ability to block navigation if for example the current active ViewModel has pending changes. In a case like this we should present the user with an option to save or discard their changes before navigating away from the current ViewModel.

By default, the Navigator class checks if the current ViewModel implements IGuardClose. Screen and the Conductor base classes all implement IGuardClose, so we can simply override the CanClose method as illustrated below. Notice in the above example that navigation is an asynchronous operation. We have all the time in the world to wait for user input before we let the navigation continue or cancel if the user decides they made a mistake. The following example demonstrates how to prompt the user to save or discard pending changes or cancel all together.

```
public override async void CanClose(Action<bool> callback)
{
  try
  {
    if (UnitOfWork.HasChanges())
    {
      var dialogResult = await _dialogManager.ShowMessageAsync(
        "There are unsaved changes. Would you like to save your changes?",
        DialogResult.Yes, DialogResult.Cancel, DialogButtons.YesNoCancel);
      using (Busy.GetTicket())
      {
        if (dialogResult == DialogResult.Yes)
          await UnitOfWork.CommitAsync();
        if (dialogResult == DialogResult.No)
          UnitOfWork.Rollback();
        callback(true);
      }
    }
    else
      base.CanClose(callback);
  }
  catch (TaskCanceledException)
  {
    callback(false);
  }
  catch (Exception)
  {
    callback(false);
    throw;
  }
}
```

Overriding the default guard

In case the default behavior is not sufficient, the Navigator can be configured with a custom guard. The following example configures the Navigator with a guard that always let's the user navigate away from the current ViewModel not matter what. The guard is an asynchronous function that receives the current ViewModel as the argument and needs to return Task<bool>. If the task's result value is true, then the navigation will continue, otherwise the navigation task will be cancelled.

```
_navigator = new Navigator(this)
  .Configure(config => config.WithActiveItemGuard(x => TaskFns.FromResult(true)));
```

Guarding the target ViewModel

Similar to guarding the current active ViewModel, the Navigator supports guarding the target ViewModel. By default no guard is active for the target ViewModel. A possible such guard could be used to ensure the current user has permission to navigate to the specified ViewModel as demonstrated in the following example.

```
[RequiresRoles("Administrator")]
public class AdminViewModel : Screen
{
}
public class ShellViewModel : Conductor<object>
{
  private readonly IAuthenticationService _authenticationService;
  private readonly INavigator _navigator;
  public ShellViewModel(IAuthenticationService authenticationService)
```

```
    {
        _authenticationService = authenticationService;
        _navigator = new Navigator(this)
            .Configure(config => config.WithTargetGuard(Authorize));
    }
    public async void GoToAdmin()
    {
        await _navigator.NavigateToAsync<AdminViewModel>();
    }
    private async Task<bool> Authorize(Type viewModelType)
    {
        if (_authenticationService.Principal == null)
            return await TaskFns.FromResult(false);   // Abort navigation
        var attributes = viewModelType.GetCustomAttributes(typeof (AuthorizationAttribute), true)
            .Cast<AuthorizationAttribute>()
            .ToList();
        var authorized = !attributes.Any() ||
                         attributes.All(attribute => attribute.Authorize(_authenticationService.Principal));
        // Fail the navigation task if not authorized, so the caller knows what happened.
        if (!authorized)
            throw new UnauthorizedAccessException("Access denied");
        return await TaskFns.FromResult(true);
    }
}
```

Handling Navigation events

ViewModels can participate in and control a pending navigation by implementing INavigationTarget. INavigationTarget provides several event methods that correspond to specific events in the navigation pipeline.

```
/// <summary>
///     An optional interface for a view model to add code that responds to navigation events.
/// </summary>
public interface INavigationTarget
{
    /// <summary>
    ///     Invoked when the view model becomes the current active view model at
    ///     the end of a navigation request.
    /// </summary>
    /// <param name="args">Data relating to the pending navigation request.</param>
    void OnNavigatedTo(NavigationArgs args);
    /// <summary>
    ///     Invoked immediately before the view model is deactivated and is no
    ///     longer the active view model due to a navigation request.
    /// </summary>
    /// <param name="args">Data relating to the pending navigation request.</param>
    void OnNavigatingFrom(NavigationCancelArgs args);
    /// <summary>
    ///     Invoked immediatly after the view model is deactivated and is no
    ///     longer the active view model due to a navigation request.
    /// </summary>
    /// <param name="args"></param>
    void OnNavigatedFrom(NavigationArgs args);
}
```

Asynchronous event handling (Requires Punch v2.4 or higher)

The methods above support asynchronous handling of each of the events using the following pattern.

```
public async void OnNavigatedTo(NavigationArgs args)
{
    // Pause pending navigation
    args.Defer();
    try
    {
        await Start((Guid) args.Parameter);
        // Successfully continue pending navigation
        args.Complete();
    }
    catch (Exception e)
```

```
    {
      // Fail pending navigation with provide Exception
      args.Fail(e);
    }
}
```

Differences in Windows Store apps

In Windows Store apps the Navigator class supports the use of the Frame control as a navigation container. In addition, CocktailMefWindowsStoreApplication automatically configures an instance of the Navigator for the application's root frame and registers it with the MEF container.

We can then inject the root Navigator as a dependency into any ViewModel to navigate from page to page as demonstrated in the following example. The Navigator also supports going forward and backward in the Frame's navigation history.

```
public class ListPageViewModel : Screen
{
  private readonly INavigator _navigator;
  // Inject Punch root navigation service
  public ListPageViewModel(INavigator navigator)
   {
      _navigator = navigator;
   }
  public Customer SelectedCustomer
   {
     get { return _selectedCustomer; }
     set
      {
        if (Equals(value, _selectedCustomer)) return;
         _selectedCustomer = value;
         NotifyOfPropertyChange(() => SelectedCustomer);
         NavigateToDetailPage();
      }
   }
  public bool CanGoBack
   {
     get { return _navigator.CanGoBack; }
   }
  public async void GoBack()
   {
     try
      {
         await _navigator.GoBackAsync();
      }
     catch (Exception e)
      {
         _errorHandler.Handle(e);
      }
   }
  private async void NavigateToDetailPage()
   {
     try
      {
        // Navigate to detail page and initialize page with the selected customer.
        await _navigator.NavigateToAsync<DetailPageViewModel>(
            target => target.Start(_selectedCustomer.CustomerID));
      }
     catch (Exception e)
      {
         _errorHandler.Handle(e);
      }
   }
}
```