Contents

The Punch DialogManager provides a convenient way to handle user prompts and dialog boxes.

Introduction

Message boxes and dialog boxes are commonly found in any given desktop application. To display a message box, it's very tempting to use the standard MessageBox class. Problem with that is you can't style it and it just feels out of place, but as a developer you also don't want to spend your time on creating views and view models for message boxes and dialog boxes.

In addition, a user prompt is actually an asynchronous operation in particular in Silverlight where there is no such thing as a modal dialog box. The application has to wait for the user to respond and then continue with the current workflow. In more technical terms, a user prompt is often part of a larger asynchronous method with asynchronous operations before and after.

The functionality described herein is not available in Windows Store apps. Windows Store apps don't have the notion of popups, other than the MessageDialog, which already provides everything needed to display a message popup in a Windows Store app.

Meet the DialogManager

Punch provides a nice little service that deals with displaying user prompts in general. The service is called the DialogManager. The DialogManager is made up of two parts. The IDialogManager interface and the default DialogManager implementation.

During start-up of the application, the bootstrapper ensures that the DialogManager gets added to the composition container.

Displaying a message prompt

The simplest use case of the DialogManager is to display a message with some standard buttons for the user to respond. The following example displays a message box asking the user "Are you sure?" with a Yes and a No button.

```
[Export]
public class MainPageViewModel
{
  private readonly IDialogManager _dialogManager;
   [ImportingConstructor]
  public MainPageViewModel(IDialogManager dialogManager)
   {
      _dialogManager = dialogManager;
   }
  public void ShowPrompt()
   {
      _dialogManager.ShowMessageAsync("Are you sure?", DialogButtons.YesNo);
   }
}
```

So far so good. In response to a bound action in the UI, this ViewModel displays the said message box and by clicking Yes or No, the user can dismiss the message. That's nice, but not so useful at the moment. As a developer you probably need to wait for the user's response and then continue one way or the other depending on whether the response was Yes or No.

So, let's turn this into an asynchronous method that waits for the user's response and displays another message based on the response.

```
[Export]
public class MainPageViewModel
{
  private readonly IDialogManager _dialogManager;
   [ImportingConstructor]
  public MainPageViewModel(IDialogManager dialogManager)
   {
      _dialogManager = dialogManager;
```

```
    }
  public async void ShowPrompt()
    {
       var dialogResult = await _dialogManager.ShowMessageAsync("Are you sure?", DialogButtons.YesNo);
      if (dialogResult == DialogResult.Yes)
          await _dialogManager.ShowMessageAsync("Good to know!", DialogButtons.Ok);
      if (dialogResult == DialogResult.No)
          await _dialogManager.ShowMessageAsync("Too bad!", DialogButtons.Ok);
    }
}
```

That's better. Now let's say we really only have additional logic if the user says Yes. If the user says No, we effectively just want to cancel the asynchronous method. For this purpose, the DialogManager allows for the developer to designate one button as the cancel button. At the same time you specify another button as the default button.

Note: DialogResult.Ok and DialogResult.Cancel are implicit dedicated default and cancel buttons respectively.

Doing this has another effect. The default button is automatically associated with hitting Enter on the keyboard and the cancel button with hitting ESC.

```
[Export]
public class MainPageViewModel
{
  private readonly IDialogManager _dialogManager;
   [ImportingConstructor]
  public MainPageViewModel(IDialogManager dialogManager)
   {
      _dialogManager = dialogManager;
   }
  public async void ShowPrompt()
   {
      await _dialogManager.ShowMessageAsync("Are you sure?", DialogResult.Yes, DialogResult.No,
                                  DialogButtons.YesNo);
      await _dialogManager.ShowMessageAsync("Good to know!", DialogButtons.Ok);
   }
}
```

Now we no longer have to check what button the user clicked on. The asynchronous method will only continue past the "await" if the user clicked Yes. If the user clicks No. The asynchronous method will be terminated.

Customizing buttons

Punch provides a list of commonly used buttons as part of the DialogButtons class. This list of buttons is modeled after the standard MessageBox.

If those buttons are not sufficient, a developer can define their own buttons. Any arbitrary object can be a button. The object becomes the content of the XAML button and as long as XAML knows how to render it everything is good. The easiest object type for a custom button is obviously a simple string. The following example displays a message box with a Save and Cancel button and displays the string associated with the button clicked by the user.

```
[Export]
public class MainPageViewModel
{
  private readonly IDialogManager _dialogManager;
   [ImportingConstructor]
  public MainPageViewModel(IDialogManager dialogManager)
   {
      _dialogManager = dialogManager;
   }
  public async void ShowPrompt()
   {
      var dialogResult = await _dialogManager.ShowMessageAsync("Are you sure?", new[] {"Save", "Cancel"});
      await _dialogManager.ShowMessageAsync(string.Format("The user clicked: {0}", dialogResult,
                                  DialogButtons.Ok);
   }
}
```

As with the out-of-box buttons, the developer can specify the default and cancel buttons.

Displaying a dialog box

Displaying a dialog box is largely the same as displaying a simple message, except that the caller provides their own ViewModel and View for the content. Everything you learned about customizing the buttons etc. applies to showing a dialog box in exactly the same manner.

The following example demonstrates a simple dialog box that asks the user for their name and then displays a message. Notice that we don't have to do anything to handle the cancel button. As mentioned earlier, DialogResult.Cancel will implicitly become the cancel button and as a result automatically cancels the asynchronous method, so that the second "yield return" will only be executed if the user clicks Ok.

```
<UserControl x:Class="DialogManagerSamples.NamePromptView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Width="Auto"
             Height="Auto">
  <Grid x:Name="LayoutRoot">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <TextBlock Grid.ColumnSpan="2" Margin="10" Text="Hello Stranger, what is your name?" />
    <TextBlock Grid.Row="1" Margin="10" VerticalAlignment="Center" Text="Firstname:" />
    <TextBlock Grid.Row="2" Margin="10" VerticalAlignment="Center" Text="Lastname:" />
    <TextBox x:Name="FirstName" Grid.Row="1" Grid.Column="1" Width="200" Margin="10" />
    <TextBox x:Name="LastName" Grid.Row="2" Grid.Column="1" Width="200" Margin="10" />
  </Grid>
</UserControl>
```

```
[Export]
public class NamePromptViewModel : Screen
{
  private string _firstName;
  private string _lastName;
  public string FirstName
  {
    get { return _firstName; }
    set
    {
      _firstName = value;
      NotifyOfPropertyChange(() => FirstName);
    }
  }
  public string LastName
  {
    get { return _lastName; }
    set
    {
      _lastName = value;
      NotifyOfPropertyChange(() => LastName);
    }
  }
}
[Export]
public class MainPageViewModel
{
  private readonly IDialogManager _dialogManager;
  private readonly NamePromptViewModel _namePrompt;
  [ImportingConstructor]
  public MainPageViewModel(IDialogManager dialogManager, NamePromptViewModel namePrompt)
  {
    _dialogManager = dialogManager;
    _namePrompt = namePrompt;
  }
  public async void AskForName()
  {
```

```
        await _dialogManager.ShowDialogAsync(_namePrompt, DialogButtons.OkCancel);
        await _dialogManager.ShowMessageAsync(
              string.Format("Nice to meet you {0} {1}", _namePrompt.FirstName, _namePrompt.LastName),
               DialogButtons.Ok);
    }
}
```

Accessing the dialog host

In the above example, it would be nice if we could disable the Ok button as long as at least one of the text boxes is empty. Instead of throwing an error message if the user didn't type anything, it makes for a much nicer user experience if we don't even let the user click Ok, until they have entered all the required data.

Punch provides access to the buttons via the dialog host. The dialog host can be obtained through the DialogHost static class.

The following example demonstrates how to use the DialogHost method to enable and disable the Ok button.

```
[Export]
public class NamePromptViewModel : Screen
{
  private string _firstName;
  private string _lastName;
  private DialogButton _okButton;
  private void OnCompleteChanged()
   {
     if (_okButton != null)
         _okButton.Enabled = IsComplete;
   }
  public string FirstName
   {
     get { return _firstName; }
     set
      {
         _firstName = value;
         NotifyOfPropertyChange(() => FirstName);
         OnCompleteChanged();
      }
   }
  public string LastName
   {
     get { return _lastName; }
     set
      {
         _lastName = value;
         NotifyOfPropertyChange(() => LastName);
         OnCompleteChanged();
      }
   }
  public bool IsComplete
   {
     get { return !string.IsNullOrWhiteSpace(FirstName) && !string.IsNullOrWhiteSpace(LastName); }
   }
  protected override void OnActivate()
   {
     base.OnActivate();
      var dialogHost = DialogHost.GetCurrent(this);
     _okButton = dialogHost.GetButton(DialogResult.Ok);
     _okButton.Enabled = IsComplete;
   }
}
```

Custom commands (new feature in v2.1)

As of version 2.1, the DialogManager supports the use of custom UI commands making it infinitely more useful for many scenarios not covered by a simple list of buttons that close the dialog.

Custom commands come in the form of the DialogUICommand<T> class and its associated interfaces. Additional overloads for ShowDialogAsync and ShowMessageAsync accept a list of commands. By default, each command still closes the dialog or message box with the associated DialogResult, but the developer can fully control the desired behavior by handling the command's Invoked event.

The following example shows the Login popup from TempHire taking advantage of custom commands. In the ShowAsync method it sets up the "Login" and "Close" command in case of WPF and then displays itself through the DialogManager. The login command's Invoked handler first cancels the command. Canceling the command will instruct the DialogManager not to proceed with closing the dialog, while we asynchronously handle the login. Once the asynchronous login is successful, it instructs the dialog host to close.

In order to avoid potential memory leaks, all Invoked handlers will automatically be removed once the dialog host closes.

```
[Export]
public class LoginViewModel : Screen
{
  private readonly IAuthenticationService _authenticationService;
  private readonly IGlobalCache _globalCache;
  private readonly IDialogManager _dialogManager;
  private string _failureMessage;
  private string _password;
  private string _username;
  private IDialogUICommand<DialogResult> _loginCommand;

   [ImportingConstructor]
  public LoginViewModel(IAuthenticationService authenticationService, IDialogManager dialogManager,
                        [Import(AllowDefault = true)] IGlobalCache globalCache)
   {
     /// Snip for clarity
   }
  public IBusyWatcher Busy { get; private set; }
  public string Username
   {
     get { return _username; }
     set
      {
         _username = value;
         NotifyOfPropertyChange(() => Username);
         UpdateCommands();
      }
   }
  public string Password
   {
     get { return _password; }
     set
      {
         _password = value;
         NotifyOfPropertyChange(() => Password);
         UpdateCommands();
      }
   }
  public string FailureMessage
   {
     get { return _failureMessage; }
     set
      {
         _failureMessage = value;
         NotifyOfPropertyChange(() => FailureMessage);
         NotifyOfPropertyChange(() => FailureMessageVisible);
      }
   }
  public bool FailureMessageVisible
   {
     get { return !string.IsNullOrWhiteSpace(_failureMessage); }
   }
  private bool CanLogin
   {
     get { return !string.IsNullOrWhiteSpace(Username) && !string.IsNullOrWhiteSpace(Password); }
   }
  private async Task LoginAsync()
   {
     using (Busy.GetTicket())
      {
         FailureMessage = "";
         var credential = new LoginCredential(Username, Password, null);
        // Clear username and password fields
        Username = null;
```

```
            Password = null;
            try
            {
                await _authenticationService.LoginAsync(credential);
                if (_globalCache != null)
                {
                    try
                    {
                        await _globalCache.LoadAsync();
                    }
                    catch (Exception e)
                    {
                        throw new Exception("Failed to load global entity cache. Try again!", e);
                    }
                }
            }
            catch (Exception e)
            {
                FailureMessage = e.Message;
            }
        }
    }
    public Task ShowAsync()
    {
        var commands = new List<IDialogUICommand<DialogResult>>();
        _loginCommand = new DialogUICommand<DialogResult>("Login", DialogResult.Ok, true);
        _loginCommand.Invoked += async (sender, args) =>
            {
                args.Cancel();  // Cancel command, we'll take it from here.
                await LoginAsync();
                if (_authenticationService.IsLoggedIn)
                    args.DialogHost.TryClose(_loginCommand.DialogResult);
            };
        commands.Add(_loginCommand);
#if !SILVERLIGHT
        var closeCommand = new DialogUICommand<DialogResult>("Close", DialogResult.Cancel, false, true);
        commands.Add(closeCommand);
#endif
        UpdateCommands();
        return _dialogManager.ShowDialogAsync(commands, this);
    }
    private void UpdateCommands()
    {
        _loginCommand.Enabled = CanLogin;
    }
}
```

Customizing the dialog host

If the standard look and feel of a dialog or message box doesn't fit with the overall look of your application, Punch allows for the complete customization of the dialog host.

To start customizing the dialog host, we first need to create our own ViewModel and subclass DialogHostBase. Punch will automatically discover this ViewModel and use it instead of the default.

```
public class CustomDialogViewModel : DialogHostBase
{

}
```

Once we have our own ViewModel, we can create a corresponding View. The following simple Silverlight example uses red bold text for the buttons instead of the standard look.

```
<controls:ChildWindow x:Class="CustomDialog.CustomDialogView"
                      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                      xmlns:controls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls"
                      xmlns:cal="http://www.caliburnproject.org">
    <Grid x:Name="LayoutRoot" MinWidth="166" Margin="2">
        <Grid.RowDefinitions>
```

```
        <RowDefinition />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <ContentControl x:Name="ActiveItem" Width="Auto" Height="Auto" HorizontalContentAlignment="Stretch"
                      VerticalContentAlignment="Stretch" IsTabStop="False" TabIndex="0" />
      <ItemsControl x:Name="DialogButtons" Grid.Row="1" IsEnabled="{Binding ActionsEnabled}" IsTabStop="False">
        <ItemsControl.ItemsPanel>
          <ItemsPanelTemplate>
            <StackPanel HorizontalAlignment="Right" Orientation="Horizontal" />
          </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
        <ItemsControl.ItemTemplate>
          <DataTemplate>
            <!-- Customized buttons -->
            <Button Width="75" Height="23" Margin="5" cal:Message.Attach="Close($dataContext)"
                    Content="{Binding Content}" IsEnabled="{Binding Enabled}"
                    Foreground="Red" FontWeight="Bold" />
          </DataTemplate>
        </ItemsControl.ItemTemplate>
      </ItemsControl>
    </Grid>
</controls:ChildWindow>
```