Decoupling views and navigation Xamarin.Forms

Posted: Thursday, 4 October 2018 Author: Patrick Allwood Estimated reading time: 20 minute read

Xamarin Forms is a great option for cross platform mobile development, but it's not uncommon to see posts on StackOverflow or forums from people struggling with issues surrounding Navigation between screens and having suitable interception points in the page lifecycle for pre-loading data before the page presents.

In this article I'm going to present a method of navigation in Xamarin Forms which:

- Puts the ViewModels squarely in charge of Navigation
- Solves some pain points around page lifecycle events
- Promotes ease of unit testing, and doesn't require breaking the MVVM abstraction by having ViewModels be aware of which views will be presented

I've put together a sample project demonstrating this method, available on GitHub.

So what's the problem that needs to be solved?

Xamarin Forms encourages the use of the MVVM architectural pattern, which separates presentation from state and behaviour and enables unit testing the behaviour of your app without being tied to the UI. In this pattern, the ViewModels, which encompasses the state and behaviour of your application, should have no knowledge of the Views that present them to the user. ViewModels are the application, Views are merely a skin on top.

Navigating between screens is behavioural logic, and ought to be triggered from a ViewModel. However the abstraction that Xamarin Forms gives us for navigation requires that we pass a Page, which is a View in Xamarin terms, to the Xamarin navigation service. This creates a bit of a square peg/round hole situation - if we perform the navigation in our ViewModel, we need to make the ViewModels aware of the Views used to present themselves and we break the MVVM pattern by coupling the behaviour to the UI.

It's not uncommon to see people trying to respect the MVVM abstraction by moving their navigation into the code-behind of the views, but this moves behavioural logic which should be testable into an area that is difficult to test. There's even libraries out there to help unit testing pages and UI components in an attempt to get around this issue.

My belief is that the navigation abstraction here is wrong, and *when navigating between screens a ViewModel should request another ViewModel be shown*. Doing this would adhere to MVVM, not require any gymnastics to unit test, and allow us to set up the next ViewModel fully ahead of time, instead of needing a page event to fire before the page appears in order to set up the state (this event doesn't exist in the current version of Xamarin Forms, but I believe it's on the roadmap for the future due to frequent demand). *All the problems go away if you change the navigation abstraction*.

First, some MVVM boilerplate

We're going to need some MVVM boilerplate. How the MVVM architectural pattern is implemented is out of the scope of this article, but here's a guide in case you need a refresher.

Lets go ahead and define what we want our ViewModel lifecycle calls to look like. For the sake of this article I'm going to keep it simple and only have two lifecycle events, one called before the first time a ViewModel comes into view in order to perform some initialisation, and the other called when we're done with the ViewModel to do some cleanup.

```
public interface IViewModelLifecycle
{
    /// <summary>
    /// Called exactly once, before the viewmodel enters the navigation stack
    /// </summary>
    Task BeforeFirstShown();


    /// <summary>
    /// Called exactly once, when the viewmodel leaves the navigation stack
    /// </summary>
    Task AfterDismissed();

    // You may also wish to implement additional lifecycle hooks eg.
    // Before a viewmodel is shown when navigating backwards, or after a viewmodel has been shown
}
```

And implement a ViewModel base class...

```
public abstract class PropertyChangedBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;


    protected void RaiseAllPropertiesChanged()
    {
        OnPropertyChanged(string.Empty);
    }

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    protected bool SetPropertyAndRaise<T>(ref T field, T newValue, [CallerMemberName] string prope
    {
        if (EqualityComparer<T>.Default.Equals(field, newValue))
        {
            return false;
        }

        field = newValue;

        OnPropertyChanged(propertyName);

        return true;
    }
}

public abstract class ViewModelBase : PropertyChangedBase, IViewModelLifecycle
{
    public virtual Task BeforeFirstShown()
    {
        return Task.CompletedTask;
    }

    public virtual Task AfterDismissed()

    {
        return Task.CompletedTask;
    }
}
```

Setting up for ViewModel-First Navigation

Xamarin Forms gives us the `INavigation` interface in order to move between pages. As we want to work exclusively with ViewModels, we'll define our own Navigation service interface, and the implementation will act as an adapter to translate our requests to navigate by `ViewModel` into requests to navigate by `Page` for the Xamarin Navigation service.

As starting point, a simple but workable interface for our Navigation service is as follows

```
public interface INavigationService
{
    /// <summary>
    /// Sets the viewmodel to be the main page of the application
    /// </summary>
    void PresentAsMainPage(ViewModelBase viewModel);

    /// <summary>
    /// Sets the viewmodel as the main page of the application, and wraps its page within a Naviga
    /// </summary>
    void PresentAsNavigatableMainPage(ViewModelBase viewModel);

    /// <summary>
    /// Navigate to the given page on top of the current navigation stack
    /// Navigate to the given page on top of the current navigation stack
    /// </summary>
    Task NavigateTo(ViewModelBase viewModel);

    /// <summary>
    /// Navigate to the previous item in the navigation stack
    /// </summary>
    Task NavigateBack();

    /// <summary>
    /// Navigate back to the element at the root of the navigation stack
    /// </summary>
    Task NavigateBackToRoot();
}
```

The first thing our `NavigationService` needs is a way to set the `App.MainPage`, and through that, get hold of the `Xamarin.Forms.INavigation` service.

```
public interface IHaveMainPage
{
    Page MainPage { get; set; }
}
```

And apply this to our `App.cs`

```
public partial class App : Application, IHaveMainPage
{
    public App()
    {
        InitializeComponent();

        MainPage = new MainPage();
    }
}
```

The other thing we need is the ability to create the correct `Page` for any given view model. I'm going to have a view locator which uses a naming convention to associate a ViewModel with a Page, ie. `MainViewModel` would be bound to an instance of `MainView`. Those of us that aren't keen on naming conventions like this can supply their own `IViewLocator`.

```
public interface IViewLocator
{
    Page CreateAndBindPageFor<TViewModel>(TViewModel viewModel) where TViewModel : ViewModelBase;
}

public class ViewLocator : IViewLocator
{
    public Page CreateAndBindPageFor<TViewModel>(TViewModel viewModel) where TViewModel : ViewMode
    {
        var pageType = FindPageForViewModel(viewModel.GetType());

        var page = (Page)Activator.CreateInstance(pageType);

        page.BindingContext = viewModel;

        return page;
    }

    protected virtual Type FindPageForViewModel(Type viewModelType)
    {
        var pageTypeName = viewModelType
            .AssemblyQualifiedName
            .Replace("ViewModel", "View");

        var pageType = Type.GetType(pageTypeName);
        if (pageType == null)
            throw new ArgumentException(pageTypeName + " type does not exist");

        return pageType;
    }
}
```

Implementing the Navigation Service

```
public class NavigationService : INavigationService
{
    private readonly IHaveMainPage _presentationRoot;
    private readonly IViewLocator _viewLocator;

    public NavigationService(IHaveMainPage presentationRoot, IViewLocator viewLocator)
    {
        _presentationRoot = presentationRoot;
        _viewLocator = viewLocator;
    }

    private Xamarin.Forms.INavigation Navigator => _presentationRoot.MainPage.Navigation;
}
```

And from here on out, it should be quite straightforward to implement our Navigation methods, and give our ViewModels a poke at the appropriate point in their lifecycle.

When changing the App's `MainPage` we need to be aware of any existing pages and call `AfterDismissed` on any ViewModels going out of scope, as well as listening for `NavigateBack` requests through the Navigation Bar's back button.

```
public void PresentAsMainPage(ViewModelBase viewModel)
{
    var page = _viewLocator.CreateAndBindPageFor(viewModel);

    IEnumerable<ViewModelBase> viewModelsToDismiss = FindViewModelsToDismiss(_presentationRoot.Ma

    if (_presentationRoot.MainPage is NavigationPage navPage)
    {
        // If we're replacing a navigation page, unsub from events
```

```
            navPage.PopRequested -= NavPagePopRequested;
        }

        viewModel.BeforeFirstShown();

        _presentationRoot.MainPage = page;

        foreach (ViewModelBase toDismiss in viewModelsToDismiss)
        {
            toDismiss.AfterDismissed();
        }
    }

    public void PresentAsNavigatableMainPage(ViewModelBase viewModel)
    {
        var page = _viewLocator.CreateAndBindPageFor(viewModel);

        NavigationPage newNavigationPage = new NavigationPage(page);


        IEnumerable<ViewModelBase> viewModelsToDismiss = FindViewModelsToDismiss(_presentationRoot.Ma

        if (_presentationRoot.MainPage is NavigationPage navPage)
        {
            navPage.PopRequested -= NavPagePopRequested;
        }

        viewModel.BeforeFirstShown();

        // Listen for back button presses on the new navigation bar
        newNavigationPage.PopRequested += NavPagePopRequested;
        _presentationRoot.MainPage = newNavigationPage;

        foreach (ViewModelBase toDismiss in viewModelsToDismiss)
        {
```

```
        toDismiss.AfterDismissed();
    }
}

private IEnumerable<ViewModelBase> FindViewModelsToDismiss(Page dismissingPage)
{
    var viewmodels = new List<ViewModelBase>();

    if (dismissingPage is NavigationPage)
    {
        viewmodels.AddRange(
            Navigator
                .NavigationStack
                .Select(p => p.BindingContext)
                .OfType<ViewModelBase>()
        );
    }
    else
    {

        var viewmodel = dismissingPage?.BindingContext as ViewModelBase;
        if (viewmodel != null) viewmodels.Add(viewmodel);
    }

    return viewmodels;
}

private void NavPagePopRequested(object sender, NavigationRequestedEventArgs e)
{
    if (Navigator.NavigationStack.LastOrDefault()?.BindingContext is ViewModelBase poppingPage)
    {
        poppingPage.AfterDismissed();
    }
}
```

Pushing and popping pages on the NavigationStack is super easy, we call our lifecycle hook before a new page is shown, and when an old page is discarded.

```
public async Task NavigateTo(ViewModelBase viewModel)
{
    var page = _viewLocator.CreateAndBindPageFor(viewModel);

    await viewModel.BeforeFirstShown();

    await Navigator.PushAsync(page);
}

public async Task NavigateBack()
{
    var dismissing = Navigator.NavigationStack.Last().BindingContext as ViewModelBase;

    await Navigator.PopAsync();

    dismissing?.AfterDismissed();
}

public async Task NavigateBackToRoot()
{
    var toDismiss = Navigator
        .NavigationStack
        .Skip(1)
        .Select(vw => vw.BindingContext)
        .OfType<ViewModelBase>()
        .ToArray();

    await Navigator.PopToRootAsync();

    foreach (ViewModelBase viewModel in toDismiss)
    {
        viewModel.AfterDismissed().FireAndForget();
    }
}
```

Finally we can change our App's startup to Navigate ViewModel first.

```
public partial class App : Application, IHaveMainPage
{
    public App()
    {
        InitializeComponent();

        var navigator = new NavigationService(this, new ViewLocator());

        var rootViewModel = new MainViewModel(navigator);

        navigator.PresentAsNavigatableMainPage(rootViewModel);
    }
}
```

And there we go! We've got a navigation service that:

- Allows us to have complete control over the creation of our ViewModels
- Fits nicely into the MVVM abstraction by not requiring ViewModels to be aware of the Pages that are used to present themselves
- Is unit testable and promotes unit testing of your navigation logic, if you would have otherwise performed navigation in the code-behind of a view in order to not break the MVVM abstraction, it can now be moved to the ViewModel
- Is easily extensible. Some of the things we've done with this are
    - Abstracted away the creation of NavigationPages to another component so that more specialised NavigationPages can be used instead.
    - Provide more intricate navigations such as inserting a new page in the navigation stack and navigating backwards to it.
    - Provide an abstraction for showing viewmodels as modal dialogs

- Support nested navigation pages and navigation stacks

A sample project which pulls all of the above together and demonstrates navigating between pages can be found on GitHub

About the author

Patrick Allwood

Senior Software Developer