Exrin Documentation

Release latest

Mar 03, 2017

Contents

A Xamarin Forms Application Framework designed to enable teams to develop consistent, reliable and highly performant mobile apps. Exrin lets you put more focus on how the app will look and functionality, while it takes care of handling many common tasks.

An extended MVVM framework is the best way to think of Exrin, with additional features including:

- Stack based Navigation Service
- ViewModel ICommand Operations
- Insights Wrapper
- Thread Helpers

There are no dependencies on Xamarin Forms, hence this will work with any version.

CHAPTER 1

Getting Started

Install Nuget Package

Search for Exrin in Nuget and install into the Native Project and the common PCL.

Initialize Framework

In your native project, after the Xamarin Forms Init, add in the following code

```
Exrin.Framework.App.Init();
```

This sets up your project and you are now free to use whatever parts of Exrin you want. If you want to run your entire project on the Exrin Framework, follow all the details below.

Proxies

In order for Exrin to avoid dependencies on your platform or other packages it creates what we call a proxy to connect Exrin to Xamarin Forms. This allows Exrin to stay independent from Xamarin Forms.

Injection

Dependency injection is a major part of Exrin and we wanted you to choose your favorite framework. This is where we proxy your Dependency Injection framework to Exrin.

A well known DI Framework is AutoFac, which will be used in this example. As per the example below, create a Folder in your project of Proxy, then a class called Injection that implements IInjectionProxy.

Important Note: In the Init() method, you must inject IInjectionProxy into itself. I call this Injection Inception.

```
public class Injection : IInjectionProxy
{
    private static ContainerBuilder _builder = null;
    private static IContainer Container { get; set; } = null;
    private static IList<Type> _registered = new List<Type>();

    public void Init()
    {
        if (_builder == null)
        {
            _builder = new ContainerBuilder();

            _builder.RegisterInstance<IInjectionProxy>(this).SingleInstance();
        }
    }
    public void Complete()
    {
        if (Container == null)
            Container = _builder.Build();
    }
    private void Register<T>(IRegistrationBuilder<T, IConcreteActivatorData,␣
↪SingleRegistrationStyle> register, InstanceType type)
    {
        switch (type)
        {
            case InstanceType.EachResolve:
                register.InstancePerDependency();
                break;
            case InstanceType.SingleInstance:
                register.SingleInstance();
                break;
            default:
                register.InstancePerDependency();
                break;
        }
    }
    public void Register<T>(InstanceType type) where T : class
    {
        Register(_builder.RegisterType<T>(), type);
        _registered.Add(typeof(T));
    }

    public void RegisterInterface<I, T>(InstanceType type) where T : class, I
                                        where I : class
    {
        Register(_builder.RegisterType<T>().As<I>(), type);
        _registered.Add(typeof(I));
    }

    public void RegisterInstance<I, T>(T instance) where T : class, I
                                        where I : class
    {
        _builder.RegisterInstance<T>(instance).As<I>().SingleInstance();
        _registered.Add(typeof(I));
    }

    public void RegisterInstance<I>(I instance) where I : class
    {
```

```
        _builder.RegisterInstance(instance).As<I>().SingleInstance();
        _registered.Add(typeof(I));
    }


    public T Get<T>(bool optional = false) where T : class
    {
        if (Container == null)
            throw new NullReferenceException($"{nameof(Container)} is null. Have you
↪called {nameof(IInjectionProxy)}.{nameof(Init)}() and {nameof(IInjectionProxy)}.
↪{nameof(Complete)}()?");

        if (optional)
            if (!Container.IsRegistered<T>())
                return null;

        return Container.Resolve<T>();
    }

    public object Get(Type type)
    {
        if (Container == null)
            throw new NullReferenceException($"{nameof(Container)} is null. Have you
↪called {nameof(IInjectionProxy)}.{nameof(Init)}() and {nameof(IInjectionProxy)}.
↪{nameof(Complete)}()?");
        return Container.Resolve(type);
    }

    public bool IsRegistered<T>()
    {
        return _registered.Contains(typeof(T));
    }

}
```

Navigation Proxy

The INavigationProxy proxies the Navigation Page for Exrin. This allows Exrin to not be dependant upon any Xamarin Forms version. You must implement INavigationProxy as shown in the example below.

```
public class NavigationProxy : Exrin.Abstraction.INavigationProxy
{

    private NavigationPage _page = null;
    public event EventHandler<IViewNavigationArgs> OnPopped;
    private Queue<object> _argQueue = new Queue<object>();
    public VisualStatus ViewStatus { get; set; } = VisualStatus.Unseen;

    public NavigationProxy(NavigationPage page)
    {
        _page = page;
        _page.Popped += _page_Popped;
    }

    private void _page_Popped(object sender, NavigationEventArgs e)
    {
```

```
        if (OnPopped != null)
        {
            var poppedPage = e.Page as IView;
            var currentPage = _page.CurrentPage as IView;
            var parameter = _argQueue.Count > 0 ? _argQueue.Dequeue() : null;
            OnPopped(this, new ViewNavigationArgs() { Parameter = parameter,␣
↪CurrentView = currentPage, PoppedView = poppedPage });
        }
    }

    public void SetNavigationBar(bool isVisible, object page)
    {
        var bindableObject = page as BindableObject;
        if (bindableObject != null)
            NavigationPage.SetHasNavigationBar(bindableObject, isVisible);
    }

    public object NativeView { get { return _page; } }

    public bool CanGoBack()
    {
        return _page.Navigation.NavigationStack.Count > 1;
    }

    public async Task PopAsync(object parameter)
    {
        _argQueue.Enqueue(parameter);
        await _page.PopAsync();
    }

    public async Task PopAsync()
    {
        await _page.PopAsync();
    }

    public async Task PushAsync(object page)
    {
        var xamarinPage = page as Page;

        if (xamarinPage == null)
            throw new Exception("PushAsync can not push a non Xamarin Page");

        await _page.PushAsync(xamarinPage);
    }

    public async Task ShowDialog(IDialogOptions dialogOptions)
    {
        if (ViewStatus == VisualStatus.Visible)
        {
            await _page.DisplayAlert(dialogOptions.Title, dialogOptions.Message, "OK
↪");
            dialogOptions.Result = true;
        }
        else
        {
            throw new Exception("You can not call ShowDialog on a non-visible page");
        }
    }
```

```
    public Task ClearAsync()
    {
        _page = new NavigationPage();
        return Task.FromResult(true);
    }

    public Task SilentPopAsync(int indexFromTop)
    {
        _page.Navigation.RemovePage(_page.Navigation.NavigationStack[_page.Navigation.
↪NavigationStack.Count - indexFromTop - 1]);
        return Task.FromResult(true);
    }
}
```

Views

All views (aka pages) in this framework must implement IView. It is recommended that all your pages inherit from a single BasePage as per the example.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Mobile.Base.BaseView">
</ContentPage>
```

```
public partial class BaseView : ContentPage, Exrin.Abstraction.IView
{
    public BaseView()
    {
        InitializeComponent();
    }

            protected override bool OnBackButtonPressed()
    {
        return ((Exrin.Abstraction.IView)this).OnBackButtonPressed();
    }

    Func<bool> Exrin.Abstraction.IView.OnBackButtonPressed { get; set; }
}
```

Models

In the MVVM pattern, Models are there to host the business logic (behavior) and state. We will look into actually performing an action in the Model later, right now we just need to set it up. We recommend you setup a Base Model as per the example below.

```
public class BaseModel : Exrin.Framework.Model
{
    public BaseModel(IExrinContainer exrinContainer, IModelState modelState)
        : base(exrinContainer, modelState)
    {
```

```
    }
}
```

View Models

View Models are meant to be nothing more than glue code moving events and data between the View (Page) and Model.

Setting up a base View Model is recommended and it will need to have some objects injected into it as per the example.

```
public class BaseViewModel : Exrin.Framework.ViewModel
{
    public BaseViewModel(IAuthModel authModel, IExrinContainer exrinContainer,
                         IVisualState visualState, string caller =␣
↪nameof(BaseViewModel))
                         : base(exrinContainer, visualState, caller)
        {
            }
    }
```

View Containers

View Containers are there to describe your high level visual setup, for example if you have a Navigation, TabbedPage or MasterDetailPage.

```
public class AuthenticationViewContainer : Exrin.Framework.ViewContainer,␣
↪ISingleContainer
{

    public AuthenticationViewContainer(AuthenticationStack stack)
                                                        :base(ViewContainers.
↪Authentication.ToString(), stack.Proxy.NativeView)
    {
        Stack = stack;
    }

    public IStack Stack { get; set; }

}
```

This simple example shows a SingleContainer, which is essentially an empty container. You define the stack you want assigned to this view container.

Stacks

Stacks are referring to Navigation Stacks. A stack is a container that holds a number of views that you can navigate between. The most common example for this is an authentication stack and a main stack. One for login, the other as your main app. Some apps only need these 2, others may require several. Exrin has no restrictions on the amount of stacks you can have.

In the stack you must inherit from BaseStack, then Map the ViewModels, Views and Keys to each other. You must also set the default starting page of the stack.

When creating your views you will find it easier to refer to if you create an enum of them. We do this to separate the actual type or implementation of the page to a key used for navigating to it.

```
    public enum Authentication
{
    Login,
    Pin
}

public enum Main
{
    Main,
            About
}
```

At this point we also need to create an enum of the Stacks we are creating to enable us to switch between them later.

```
public enum Stacks
{
    Authentication,
    Main
}
```

```
public class AuthenticationStack : BaseStack
{
    public AuthenticationStack(IViewService viewService)
        : base(new NavigationProxy(new NavigationPage()), viewService, Stacks.
↪Authentication)
    {
        ShowNavigationBar = false;
    }
    protected override void Map()
    {
        base.NavigationMap<PinView, PinViewModel>(nameof(Authentication.Pin));
        base.NavigationMap<LoginView, LoginViewModel>(nameof(Authentication.Login));
    }

    public override string NavigationStartKey
    {
        get
        {
            return nameof(Authentication.Login);
        }
    }
}
```

Bootstrapper

Inherit from Exrin.Framework.Bootstrapper and override the InitStacks and InitModels to register or inject what you have setup.

In the base constructor you will see this is where we send the instatiated Injection object and an Action that assigns a page to the MainPage in Xamarin Forms.

```
public class Bootstrapper : Exrin.Framework.Bootstrapper
{
    public Bootstrapper() : base(new InjectionProxy(),
                                                                    (newView) => {␣
↪Application.Current.MainPage = newView as Page; }) { }

}
```

The bootstrapper will then build the container in the injection framework and connect all the Views, ViewModels, Models, Stacks and ViewHierarchies.

Launching the App

From here we are finally at a point where we will put our line of code in the App.cs file and start the app using Exrin.

```
public App()
{
    new Bootstrapper().Init().Get<INavigationService>().Navigate(new StackOptions() {␣
↪StackChoice = Stacks.Authentication });
}
```

IOperation

In order to add functionality to your ViewModel, Exrin requires that you use the IViewModelExecute for any Commands. As in the example below you will see the command for when a key is pressed on the Pin Screen in our sample app. It contains nothing more than glue code to connect to the appropriate IViewModelExecute.

```
public IRelayCommand LoginCommand
{
    get
    {
        return GetCommand(() =>
            {
                return Execution.ViewModelExecute(new LoginOperation(model)); // Pass␣
↪the instance of your model through to use
            });
    }
}
```

You need to create the class LoginOperation, which keeps the logic for the operation, separating it for each unit testability.

```
public class LoginOperation : IOperation
{
    private readonly IAuthModel _authModel;

    public LoginOperation(IAuthModel authModel)
    {
        _authModel = authModel;
    }

    public Func<IList<IResult>, object, CancellationToken, Task> Function
    {
```

```
        get
        {
            return async (results, parameter, token) =>
            {
                Result result = null;

                if (await _authModel.Login())
                    result = new Result() { ResultAction = ResultType.Navigation,␣
↪Arguments = new NavigationArgs() { Key = Main.Main, StackType = Stack.Main } };
                else
                    result = new Result() { ResultAction = ResultType.Display,␣
↪Arguments = new DisplayArgs() { Message = "Login was unsuccessful" } };

                results.Add(result);
            };
        }
    }

    public bool ChainedRollback { get; private set; } = false;

    public Func<Task> Rollback { get { return null; } }
}
```

Nesting Files

Due to the need for more classes than usual with this approach it is recommended you nest your files using Visual Studio's DependantUpon tag. Because Visual Studio doesn't have an inbuilt way to manage this, using the extension File Nesting is recommended.

Summary

Be sure to look at Unit Testing next to see the benefits of the IViewModelExecute and IModelExecute setup.