UIMAN DOCUMENTATION

TABLE OF CONTENTS

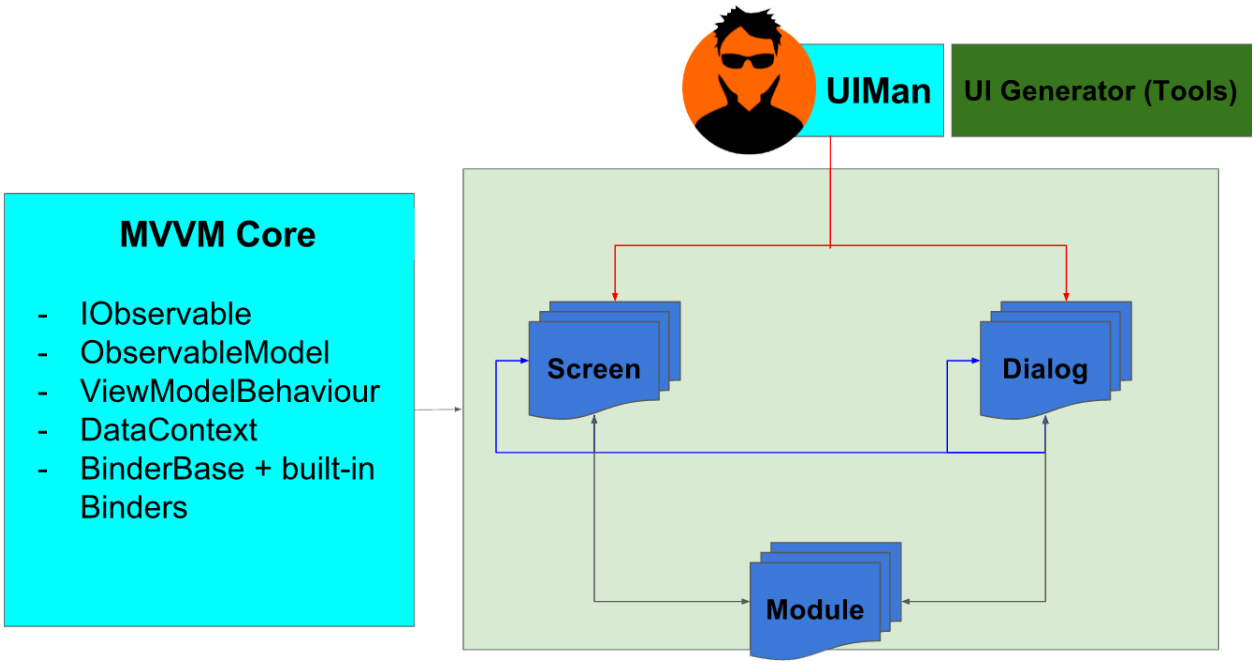
OVERVIEW

Fast and flexible solution for UI development and management with MVVM pattern. UIMan is designed most suitable for mobile game.

FEATURES

- Support Databinding, Observable and MVVM for implementing UI with uGUI.
- Prefab base UI, async loading.
- Auto Code Generation.
- UI flow and layer management system.
- UI Events.
- Customizable activity indicator.
- Easy to extend and customize.

INTRODUCTION



IMG1. UIMan structure

CONCEPTION

UIMan gives the rule that separates your UI into 3 types:

Screen

A group of UI elements is packaged into a prefab, that is a screen in your game. Screen can be seen as a Unity scene, but contain only UI.

There is only one screen can show at the same time, if you call UIMan for showing a screen, it will hide current screen automatically (if it have) before show the other.

For example, MainMenu or Battle UI is screens in your game.

Dialog

A group of UI elements is packaged into a prefab, that is one of dialogs in your game such as a message box or something else...

While screen can show only one at the same time, dialog has it's queue to show if you order to show at the same time.

If you call to show multiple dialog at the same time, UIMan will show only one dialog and push the others into queue to show when first dialog is hide.

If you order to show a dialog when there are not any dialog in transition (playing animation), UIMan will show it at the top layer of UI.

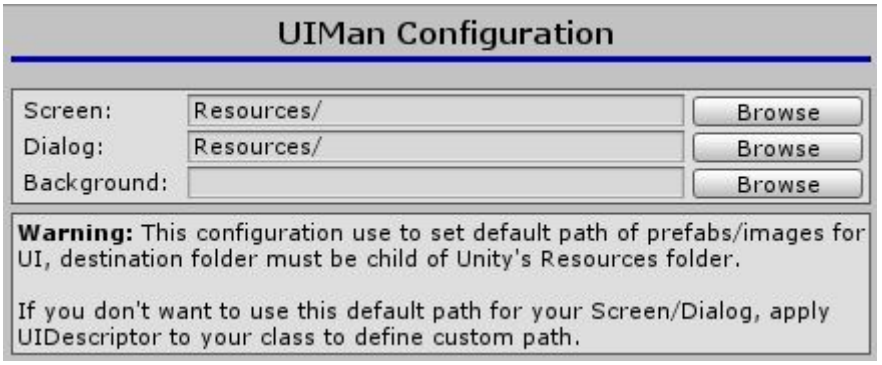
Module

A group of UI elements is packaged into a prefab, it use to perform as a part of screen or dialog. *Such as a item in a list.*

All these types have a ViewModel base class that you will extend to implement view logic and binding task.

INSTALLATION & SETUP

1. Import your UIMan's unitypackage.
2. Go to menu UIMan/Configuration and set your path to store UI's prefab.



IMG2. UIMan configuration interfaces

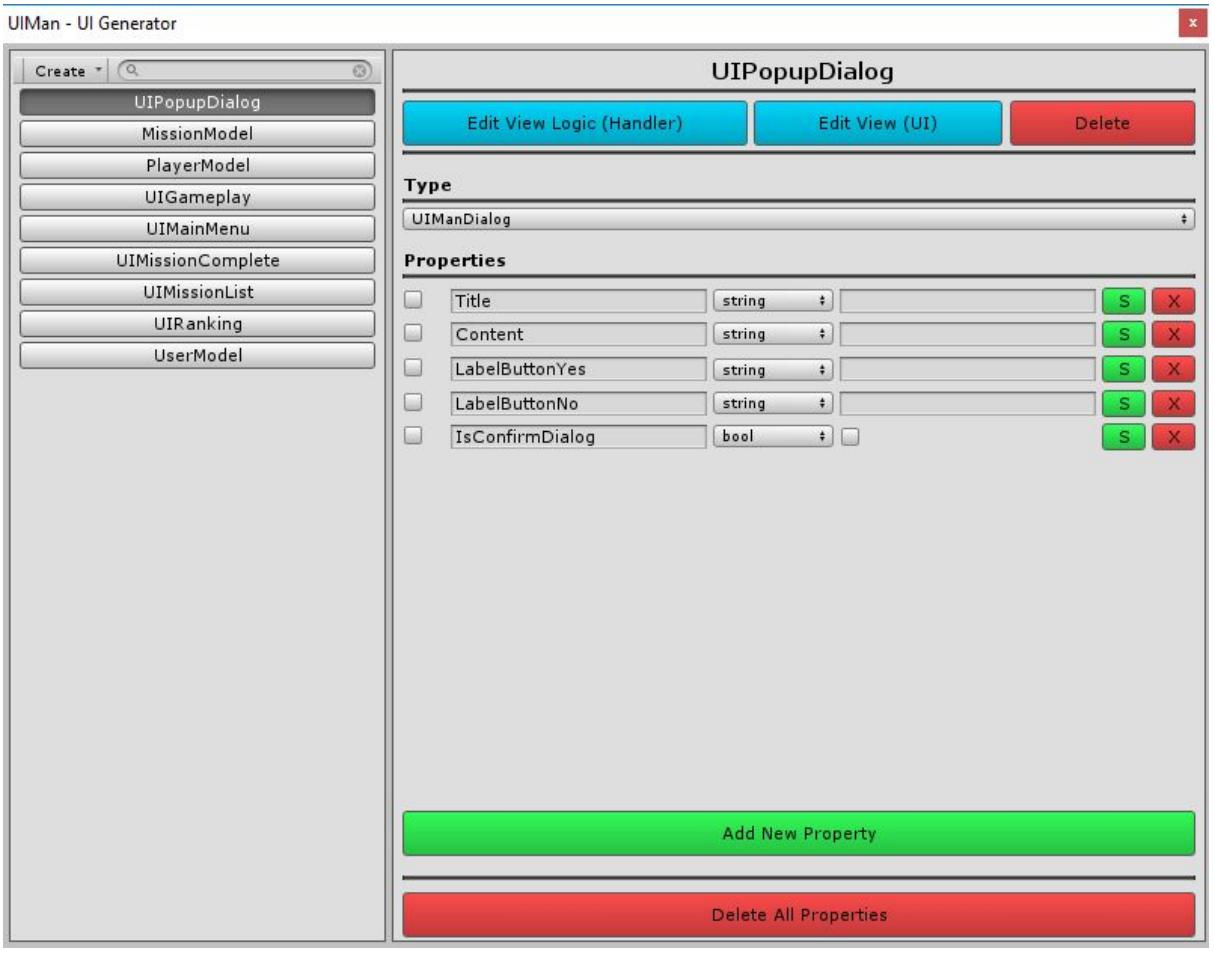
   - *Screen:* store your screen prefab files.
   - *Dialog:* store your dialog prefab files
   - *Background:* store your screen's background images.

   Make sure these folder must be child of Resources folder for loading at runtime..

3. UIMan is now ready for use!

CREATING A NEW UI

UI includes a prefab as View, C# script as ViewModel that contains Observable Properties, fields and all View's logic, UIMan supports auto generate these thing, just use UI Generator step by step:



IMG3. UI Generator interfaces

1. Goto menu UIMan/UI Generator.
2. Click Create.
3. At "New type" dialog, select your UI type (UIManDialog or UIManScreen).
4. Click Create then select folder that you want to save generated code in.
5. Input your UI name, UIMan will add prefix "UI" to that name automatically (if that name does not constain that prefix).
6. There are three file will be generated by UIMan as below:
    - <Your UI Name>.cs
    - <Your UI Name>.Handler.cs
    - <Your UI Name>.prefab

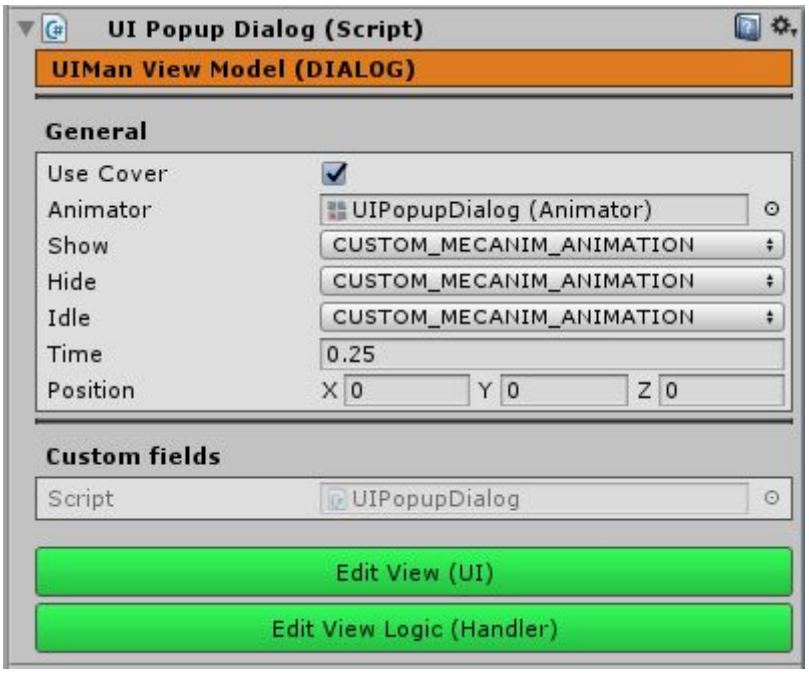
C# script file will be saved in your selected path, while prefab file will save in the path that you has been config.

Two C# script file is defines ViewModel class for your UI, Handler file will be generated only one time while the other will generate automatically by UI Generator when you want to modify.

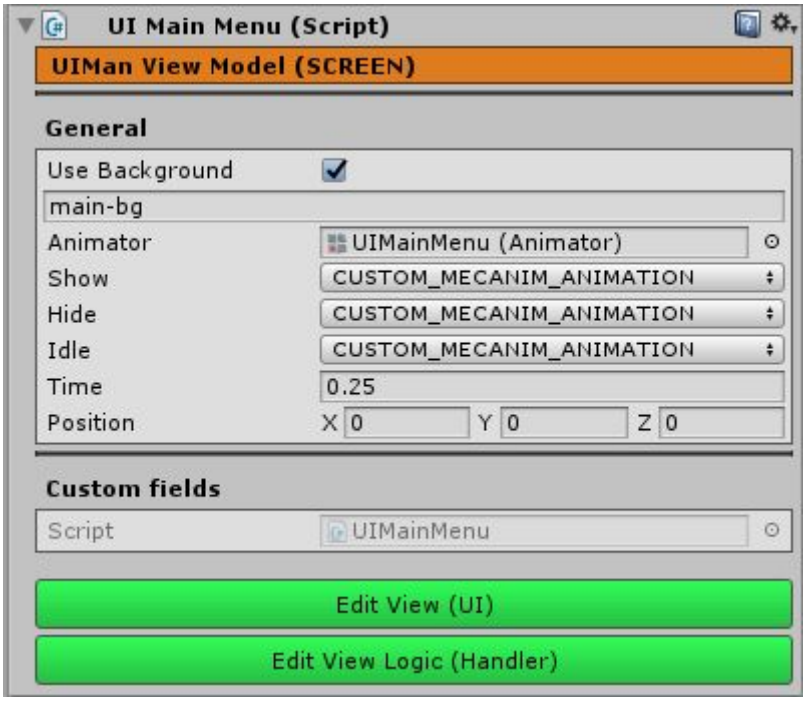
Prefab file is your View, it's UI and contains basic element and some require component has been attached.

7. Select generated ViewModel by selecting it in the left panel. The right panel contains all property and it let's you modify (add/rename/remove/change type…) properties.

8. You can edit your View (UI) by select prefab and press "Edit View (UI)" button on inspector. You also can make the UI instantiate into scene by press "Edit View (UI)" in UI Generator window.



IMG4. Dialog ViewModel inspector



IMG5. ScreenViewModel inspector

VIEWMODEL IMPLEMENTATION

UIMan let's you override some functions are:

OnShow: when you call ShowScreen/ShowDialog, UIMan calls this function first and receives parameter from caller as object array.

OnShowComplete: after Show motion is completed, UIMan will calls this function.

OnHide: before the screen/dialog become hide.

OnHideComplete: UIMan call this function after hiding transition is finished.

AnimationShow, AnimationHide, AnimationIdle: These function let you override animation for your UI if you want to tweening by your self, make sure you select "CUSTOM_SCRIPT_ANIMATION" in View's Inspector for that.

All these functions can be override in Handler file that auto generated by UI Generator. You also define your field, property or other custom implementation here.

SHOWING/HIDING THE UI

UIMan let you call to show or hide any Screen or Dialog with single line of code, it's **generic**, **dynamic** parameter and easy to use!

```
UIMan.Instance.ShowPopup ("About", "Message here!", "OK"); // Show built-in popup
(messsage box, confirm box...)

UIMan.Instance.ShowScreen<UIMainMenu> ();  // Show your screen

UIMan.Instance.ShowDialog<UIRanking> (999); // Show your custom dialog

UIMan.Instance.HideScreen<UIMainMenu> (); // Hide your screen

UIMan.Instance.HideDialog<UIRanking> (); // Hide your custom dialog

this.HideMe (); //Hide this screen/dialog

// And more...
```

IMG6. Showing/hiding a UI with single line of code

CREATING A MODULE

Module is a View and ViewModel as Screen or Dialog, but it is a child/part of Screen or Dialog, it is designed for binding a model instance to View, you can reused it in any Screen or Dialog.
Module does not support auto code generation right now, so you must implement it by your self.

1. Creating a new module's View as Screen or Dialog. You can make a prefab or attach it as child of any Screen/Dialog directly. There is no need to put it in Resources folder, put any where that you want.

2. Writing a new C# class that extend to UIManModule<T> with T is your model class. It's maybe a POCO or ObservableModel.

3. Override DataInstance property if needed.

4. Writing your custom ViewLogic in this script if needed.

5. Attaching this Module to Module's View has been created at step 1.

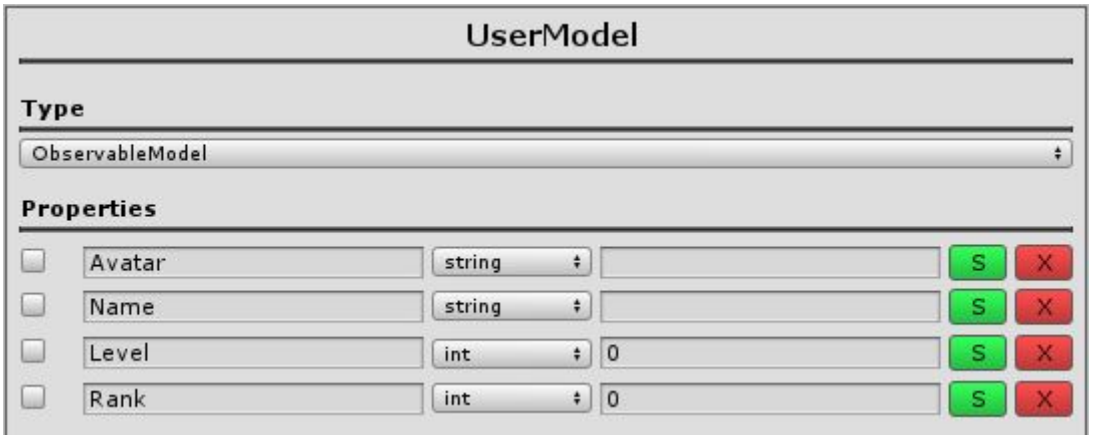
```
public partial class UserModel : ObservableModel {
    string _name= "";
    [UIManProperty]
    public string Name {
        get { return _name; }
        set { _name= value; OnPropertyChanged(); }
    }
}
public class UserModule : UIManModule<UserModel> {
    public void OnClick () {
        UIMan.Instance.ShowPopup ("Username: ", DataInstance.Name);
    }
}
```

IMG7. Implementing the Module' ViewModel

When you change Instance of Module, it will notify that change to Module's View automatically. If you want to notify to View when you change then Instance's fields or properties, you must call NotifyObjectChange function of Module's ViewModel manually or use ObservaleModel to define your model instead. See ObservableModel for more detail.

OBSERVABLE MODEL

This is base class implement of IObservable, which includes features same as ViewModel. You can use UI Generator for generating the ObservableModel or implementing by youself.



IMG8. Part of UI Generator that show ObservableProperties

DIALOG'S CALLBACK

Callback is very important for implementing your dialog. It is always show with some tasks, that will report to "caller" the result when user press any button on dialog.

```
public void OK ()
{
    this.Callback(0, _args); // Call a callback with any parameters
}
public void No ()
{
    this.Callback(1, _args);
}
```

IMG9. UIPopupDialog's callbacks

- From the caller, you can pass many function that accept the array of objects as parameters received from dialog. Let's create a new UICallback instance and pass any functions that you like to.
- UICallback store callback delegates as a list with the order same as you pass from caller. Dialog's ViewModel has a Callback function that accepts index of callback and arrays of argument as parameter, call it when you want to report to caller. Please keep in your mind when you call Callback, UIMan will hide your Dialog automatically, that is a rule for Dialog, it only calls back when it already hide.

BINDER

Binder is component that supports binding a data source from ViewModel to View's elements
You can creating your custom Binder easily by copy the template from BinderTemplate.cs

```
using UnityEngine;
using UnityEngine.UI;

namespace UnuGames.MVVM
{
    [DisallowMultipleComponent]
    public class BinderTemplate : BinderBase
    {
        [HideInInspector]
        public BindingField yourValue = new BindingField ("Text");
        // Define any field for binding as you want, just copy above
fied
        public override void Init (bool forceInit)
        {
            if (CheckInit (forceInit)) {
                // Get view's components here
                SubscribeOnChangedEvent (yourValue, OnUpdateValue);
            }
        }
        public void OnUpdateValue (object newValue)
        {
            if (newValue == null) {
                // Do what you want for null value
                return;
            }
            // Cast newValue into your binding type and assign to view
components
        }
    }
}
```

IMG10. Binder Template

Adding any fields, any update function as you want. Writing update functions that receive "new value" as parameter from OnPropertyChanged event, casting that object to binding type and set it to your View's component.

For using Binder, just attach it to any View object and assign to it with suitable property. By default, Binder fetching properties from the DataContext of parent GameObject, for changing the data source, you can drag any DataContext from others GameObject to the DataContext field of the Binder.

DATA CONTEXT

ViewModel or it's properties become a data source for Binder via DataContext. Attaching DataContext to any GameObject, select the ViewModel, select ViewModel's property if need, now you have a DataSource ready for use by the Binder.

There are two type that DataContext supports:

MonoBehaviour: Support for fetching all properties of the ViewModel. This type is suitable for binding ObservableProperties of the ViewModel.

Property: Sometimes your ObservableProperty is a object or a ObservableModel, this type support for fetching fields or properties that are childs of a field or property of the parent ViewModel.

UI EVENTS

UIMan allows you to listen to some UI transition event such as *OnScreenShow, OnBack, OnHide*… it is very convenient to implement your game logic that hook from UI event.

GETTING THE HANDLER

GetHandler<T> is function to get ViewModel instance (call as Handler) that is showing.

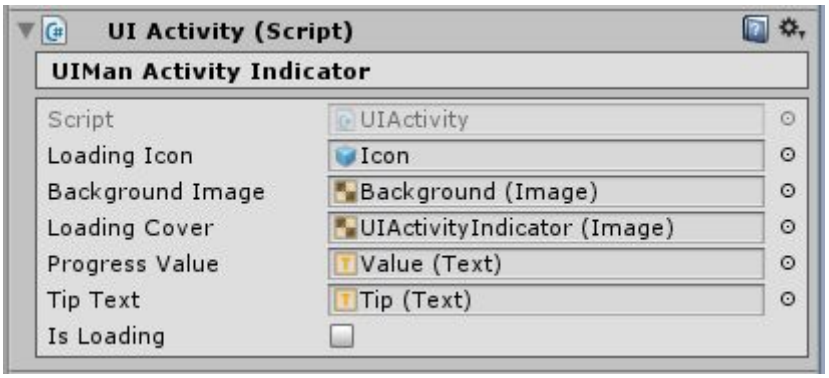
DESTROYING THE UI

By default, UIMan sets the alpha of parent CanvasGroup to hide your View, it means your UI will not render by Unity renderer but it is still in the scene, that can optimize the draw call, but the asset still exists on device memory (RAM, VRAM). The more UI you loaded, the more memory space is required for your game. For freely memory, you must destroy UI to release it's resources, UIMan offer DestroyUI<T> function for you to do that.

THE ACTIVITY INDICATOR

There are many ways for creating a Activity Indicator, UIMan defines a standard way for you as below.

UIActivityIndicator prefab: contains all necessary components for your Activity Indicator. Just drag into scene, modify and apply your change to prefab.



IMG11. Activity indicator's properties

- *Icon:* Image/icon that you want to animating, just update the ActivityIndicator's image and animation clip.
- *Background Image:* Changing it's image if you want (to make the default image for that), for showing the background image such as game's artworks.
- *Activity Cover:* A cover image that is lowest layer of ActivityIndicator, but overlay on the top of other game element (game UI, environment…) for hiding or making it blurry.
- *Progress Value:* For showing the Activity value such as load progression, just change the Text component's property for what you want to.
- *Tip Text:* For showing any tips of game, using SetTip function for change the text's value.

You can change anything in that prefab as you want, but please keep all reference is correct. For showing ActivityIndicator, just call with single line of code as below:

```
public void Ranking () {
    UIMan.Loading.Show (WaitForLoadRanking ());
}
IEnumerator WaitForLoadRanking () {
    // Fetching data from server
}
```

IMG12. Showing a LoadingIndicator

UIActivity have many overloading function that supports show a loading for IEnumerator (coroutine), AsyncOperation, WWW task, manually task...