Quick Start Guide to Strange IoC for Unity 3D

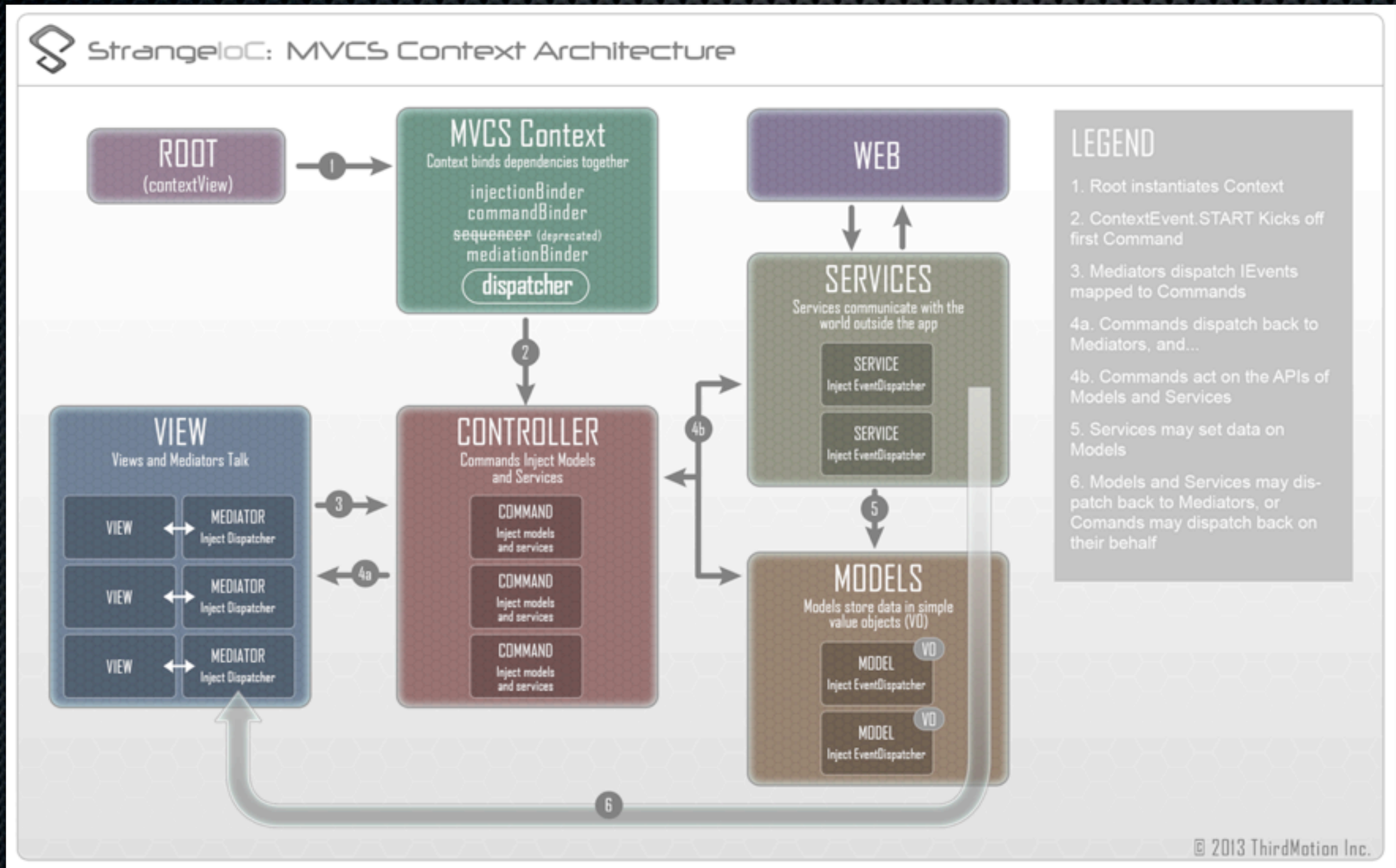
by Paul Borawski

Game Designer / Programmer / Animator

What is Strange IoC?

- Strange IoC is a **M**odel **V**iew **C**ontroller **S**ervice framework built around the design pattern of Dependency Injection
- IoC stands for Inversion of Control which means classes inject what they need instead of passing in what they need from other classes through methods and/or constructors. This creates code independent of one another.

Overview of MVCS and Strange IoC



Why use Strange IoC?

- Very little dependency, meaning you can update, remove, and add components with little to no rework
- Everything is self contained. Views don't know about Data and vice versa.
- Easy to start prototyping and mocking services.
- Interfaces are heavily used, so changing of implementation is extremely easy.

Signals vs Events

- Signal dispatch results in no new object creation, and therefore no need to GC a lot of created instances.
- Signal dispatches are type-safe and will break at compile-time if the Signals and their mapped callbacks don't match.
- Signals do create a LOT of files however.
- Just use them, you won’t regret it.

Let’s start making something

- Walk you through set up of a project and then two workflows.
- First workflow is click a button and an overlay shows up.
- Second workflow is you click get friends button and a list of friends populate on screen.

Beginning steps

1) Import Strange IoC from Unity Asset store or github
2) Create an app Context script and root script
3) Create empty game object and attach root script
4) Create a canvas for UI
5) Create start command

Beginning steps cont...

6) Create prefab(s) for your views
7) Create View script and Mediator script
8) Bind those in the app context
9) Create Signal(s) to be dispatched from view and bind to singleton or command.
10) For interaction outside of view, create a signal and command scripts and bind in context as well.

App Context Example

```
public class AppContext : MVCSContext
{
     public AppContext (MonoBehaviour view) : base(view)
     {
     }

     public AppContext (MonoBehaviour view, ContextStartupFlags flags) : base(view, flags)
     {
     }

     // Unbind the default EventCommandBinder and rebind the SignalCommandBinder
     protected override void addCoreComponents()
     {
          base.addCoreComponents();
          injectionBinder.Unbind<ICommandBinder>();
          injectionBinder.Bind<ICommandBinder>().To<SignalCommandBinder>().ToSingleton();
     }

     // Override Start so that we can fire the StartSignal
     override public IContext Start()
     {
          base.Start();
          StartSignal startSignal= (StartSignal)injectionBinder.GetInstance<StartSignal>();
          startSignal.Dispatch();
          return this;
     }

     protected override void mapBindings()
     {
          //Note how we've bound it "Once". This means that the mapping goes away as soon as the command fires.
          commandBinder.Bind<StartSignal>().To<StartCommand>().Once ();
     }
}
```

App Root Example

```
public class AppRoot : ContextView
{

     void Awake()
     {
          //Instantiate the context, passing it this instance.
          context = new AppContext(this);

          //This is the most basic of startup choices, and probably the most common.
          //You can also opt to pass in ContextStartFlag options, such as:
          //
          //context = new MyFirstContext(this, ContextStartupFlags.MANUAL_MAPPING);
          //context = new MyFirstContext(this, ContextStartupFlags.MANUAL_MAPPING | ContextStartupFlags.MANUAL_LAUNCH);
          //
          //These flags allow you, when necessary, to interrupt the startup sequence.
     }

     void Update()
     {
     }
}
```

Start Command Example

```
public class StartCommand : Command
{

	[Inject(ContextKeys.CONTEXT_VIEW)]
	public GameObject contextView{get;set;}

	private string _prefabPathStr = "prefabs/";

	private Canvas _rootCanvas;

	public override void Execute()
	{
		_rootCanvas = contextView.GetComponentInChildren<Canvas>();

		GameObject __go = UnityEngine.Object.Instantiate(Resources.Load(_prefabPathStr+"DemoViewPanel")) as GameObject;
		__go.name = "DemoView";
		__go.AddComponent<DemoView>();
		__go.transform.SetParent(_rootCanvas.transform, false);
	}
}
```

Demo View Example (shortened)

```
public class DemoView : AppView
{
	//Vars declared here...
	public DemoView()
	{
	}

	//where I initiate all my variables and UI elements
	override public void init()
	{
		Debug.Log("DemoView::init called...");
		_overlayPanelGO = _transform.Find("OverlayPanel").gameObject;
		_toggleOverlayBtn = _transform.Find("ToggleOverlayBtn").GetComponent<Button>();

		base.init();
		_registerListeners();
	}

	public void ToggleOverlay()
	{
		if (_overlayPanelGO.activeSelf)
			_overlayPanelGO.SetActive(false);
		else
			_overlayPanelGO.SetActive(true);
	}

	private void _registerListeners ()
	{
		AddButtonClickListener(_toggleOverlayBtn, "toggleOverlay", null);
	}
}
```

Demo Mediator Example (shortened)

```
public class DemoMediator : Mediator
{
     //This is how your Mediator knows about your View.
     [Inject]
     public DemoView view{ get; set;}

     public override void OnRegister(){
          _addListeners();
          view.init();
     }

     private void onButtonClicked(String __btnNameStr, object __data){
          switch (__btnNameStr)
          {
               case "toggleOverlay":
                    view.ToggleOverlay();
               break;
          }
     }

     private void _addListeners(){
          view.ViewButtonClickSignal.AddListener(onButtonClicked);
     }

     private void _removeListeners(){
          view.ViewButtonClickSignal.RemoveListener(onButtonClicked);
     }

     public override void OnRemove()
     {
          Debug.Log("Mediator OnRemove");
          _removeListeners();
     }
}
```

Create first signal

```
using System;
using strange.extensions.signal.impl;

public class ButtonClickSignal : Signal<string, object>
{
}
```

- Signals can have up to four typed params
- When binding to a command, you cannot bind the same type more than once (Ex: Signal<int, int> will not work when binding to command)
- If you want more than 4 params or multiples of same, create a value object class and use that (Ex: Signal<ISampleDataVO>)

App Context Updated

```
public class AppContext : MVCSContext
{
     //Note: removed other code for focus and screen space

     protected override void mapBindings()
     {
          mediationBinder.Bind<DemoView>().To<DemoMediator>();

          //Note how we've bound it "Once". This means that the mapping goes away as soon as the command fires.
          commandBinder.Bind<StartSignal>().To<StartCommand>().Once ();

          //Theses Signals isn't bound to any Command,
          //so we map it as an injection so a Command can fire it, and a Mediator can receive it
          injectionBinder.Bind<ButtonClickSignal>().ToSingleton();
     }

}
```

Create signal command combo

```
public class GetFriendsListSignal: Signal<int>
{
}

public class GetFriendsListCommand : Command
{
	[Inject]
	public int ListLimitInt{get;set;}

	[Inject]
	public FulfillFriendsListSignal FulfillFriendsListSig{get;set;}

	[Inject]
	public IGetDataFromWebService GetDataFromWebServ{get;set;}

	public override void Execute()
	{
		Debug.Log ("GetFriendsListCommand::Execute called...");
		Retain();
		GetDataFromWebServ.ReturnWebDataSig.AddListener(_returnWebDataHandler);
		GetDataFromWebServ.RequestData("http://www.someservice.com/friendslist");
	}

	private void _returnWebDataHandler(object __dataObj)
	{
		string[] __friendsStrArr = __dataObj as string[];
		string[] __newFriendStrArr = new string[ListLimitInt];

		for (int __i = 0; __i < ListLimitInt; __i++)
		{
			__newFriendStrArr[__i] = __friendsStrArr[__i];
		}

		FulfillFriendsListSig.Dispatch(__newFriendStrArr);
		Release();
	}
}
```

Create interface and service combo

```
public interface IGetDataFromWebService {
	GameObject contextView{get;set;}
	void RequestData(string url);
	ReturnWebDataSignal ReturnWebDataSig{get;set;}
}

public class MockGetDataFromWebService : IGetDataFromWebService
{
	[Inject(ContextKeys.CONTEXT_VIEW)]
	public GameObject contextView{get;set;}

	[Inject]
	public ReturnWebDataSignal ReturnWebDataSig{get;set;}
	private string _url;

	public MockGetDataFromWebService (){}

	public void RequestData(string url)  {
		Debug.Log("ExampleService::Request called with "+url);
		_url = url;
		MonoBehaviour root = contextView.GetComponent<AppRoot>();
		root.StartCoroutine(waitASecond());
	}

	private string[] _mockGetFriendsList()    {
		return new string[]{"bobby", "brenna", "jaryd", "chris", "andy", "rachel", "bryan", "bridget", "nick", "willy",
"kevin", "danny", "stephanie"} ;
	}

	private IEnumerator waitASecond()    {
		Debug.Log("ExampleService::waitASecond");
		yield return new WaitForSeconds(1f);

		//Pass back some fake data via a Signal
		ReturnWebDataSig.Dispatch(_mockGetFriendsList());
	}
}
```

App Context Updated

```
public class AppContext : MVCSContext
{
     //Note: removed other code for focus and screen space

     protected override void mapBindings()
     {
          mediationBinder.Bind<DemoView>().To<DemoMediator>();
          commandBinder.Bind<GetFriendsListSignal>().To<GetFriendsListCommand>();
          injectionBinder.Bind<IGetDataFromWebService>().To<MockGetDataFromWebService>().ToSingleton();

          //Note how we've bound it "Once". This means that the mapping goes away as soon as the command fires.
          commandBinder.Bind<StartSignal>().To<StartCommand>().Once ();

          //Theses Signals isn't bound to any Command,
          //so we map it as an injection so a Command can fire it, and a Mediator can receive it
          injectionBinder.Bind<ButtonClickSignal>().ToSingleton();
     }

}
```

Strange IoC resources

- Start here: http://strangeioc.github.io/strangeioc/TheBigStrangeHowTo.html
- http://www.rivellomultimediaconsulting.com/unity3d-mvcs-architectures-strangeioc/
- https://strangeioc.wordpress.com/
- https://groups.google.com/forum/#!forum/strangeioc
- https://github.com/strangeioc/strangeioc

My info to contact me

If you have any further questions on Strange IoC, contact me through any of the following ways.

twitter: @animepauly

facebook: https://www.facebook.com/animepauly

email: animepauly@mac.com

skype: animepauly

phone: 773.398.7270