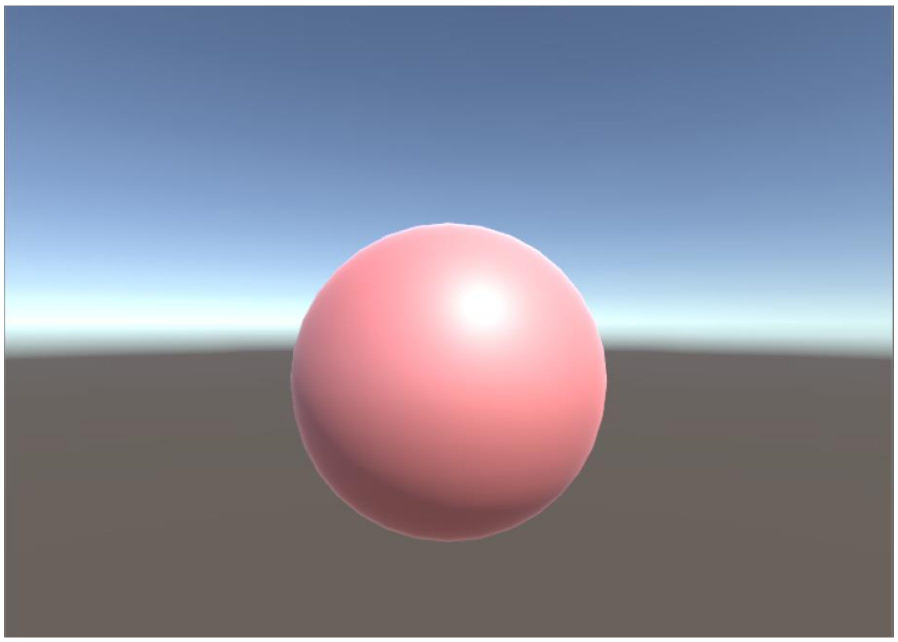
Getting Started Tutorial

This is a simple tutorial on getting started with RapidIoC. It is based off of **examples/GettingStartedExample** in the Unity project. It will help you learn the basics of RapidIoC dependency injection, signal mapping, and most importantly scene setup.

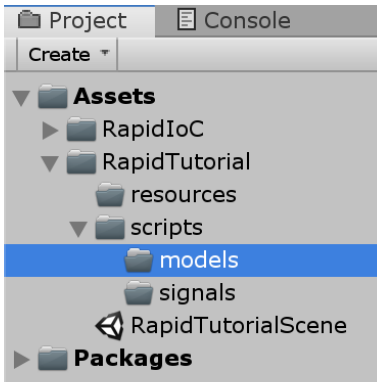
The end goal of this tutorial is very simple: creating a sphere that grows in size in small time increments. This tutorial will demonstrate how to do this using RapidIoC API.



1. Let's create a new project and import **RapidIoC.unitypackage**.

2. Create new folder **RapidTutorial**, inside create new scene **RapidTutorialScene**.

3. Because RapidIoC is best used in a multi-scene setups, I prefer to organize everything in folders labeled by their individual scene names, like so:



If you've noticed I've also created some extra folders:

- **scripts** – folder where all our game logic will be located.
    - **models** - scripts used to pass data between views or signals.
    - **signals** - all scene-related signals.
- **resources** - all scene-specific resources (e.g. prefabs, models, textures).

Noe: You are however welcome to organize folders however you like, but I found that this structure works best for me.

4. Open "scripts" folder we've just created, and create a new script **RapidTutorialSceneView**, this will be our main (and only) scene in this tutorial.

5. Open **RapidTutorialSceneView** in your IDE, and modify your scene view to derive from **cpGames.core.RapidIoC.MainSceneView**, you can remove auto-generated Start and Update functions. It should look something like this:

```
using cpGames.core.RapidIoC;

public class RapidTutorialSceneView : MainSceneView
{
}
```

6. Next let's create our sphere view and model. In the same **view** folder, create new script. Call it **SphereView**. In the "model" folder create script **SphereModel**. Open **SphereModel** first. It should not derive from anything, and will simply contain our data. In this tutorial it will hold sphere's color and size. Your **SphereModel** should look something like this:

```
using UnityEngine;

public class SphereModel
{
    public Color sphereColor = Color.white;
    public float sphereSize = 1.0f;
}
```

7. Now open **SphereView**. This will be our **ComponentModelView**. It will accept **SphereModel** as a generic argument. When the model is updated, it will update the sphere's color and size:

```
using cpGames.core.RapidIoC;
using UnityEngine;

public class SphereView : ComponentModelView<SphereModel>
{
    protected override void UpdateModel()
    {
        GetComponent<Renderer>().material.SetColor("_Color", 
Model.sphereColor);
        transform.localScale = Vector3.one * Model.sphereSize;
    }
}
```

8. But how will we assign the model to the view? There are several options. In this tutorial I will cover injecting a signal, and mapping it to a function that will update our model. Go back to Unity, in the **signal** folder create a new script **UpdateSphereSignal** and open it in IDE.

9. This will be our signal that notifies any listeners that new **SphereModel** is being issued. Derive it from a generic Signal with a **SphereModel** parameter:

```
using cpGames.core.RapidIoC;

public class UpdateSphereSignal : Signal<SphereModel> { }
```

10. Now we've created a signal, but to be able to inject it into **SphereView**, we must first bind it to context. Open again your **RapidTutorialSceneView**. This is a good place to bind your data as soon as the scene loads. For convenience, **SceneView** already provides an overridable function for that called **MapBindings** - this is where we will bind the signal with the local context:

```
using cpGames.core.RapidIoC;

public class RapidTutorialSceneView : MainSceneView
{
    protected override void MapBindings()
    {
        base.MapBindings();
        Rapid.Bind<UpdateSphereSignal>(ContextName);
    }

    protected override void UnmapBindings()
    {
        base.UnmapBindings();
        Rapid.Unbind<UpdateSphereSignal>(ContextName);
    }
}
```

Notice the required **base.MapBindings();** call, it's only required for the **MainSceneView** as it binds important signals itself. Also not the **UnmapBindings** method. Because Rapid's contexts are auto-generated and auto-destroyed, it is important to remember to clean up all binded data once the scene is unloaded.

When we pass **UpdateSphereSignal** as a generic argument, Rapid will call an empty constructor and create a new instance of **UpdateSphereSignal** under the hood. Also specifying **ContextName** parameter, makes binding local. If we were to call it parameterless: **Rapid.Bind<UpdateSphereSignal>();**, **UpdateSphereSignal** will be bound to Root context. For more information on Contexts, see documentation here.

11. Now that our signal is binded, lets create an injected property in **SphereView**:

```
using cpGames.core.RapidIoC;
using UnityEngine;

public class SphereView : ComponentModelView<SphereModel>
```

```
{
    [Inject]
    public UpdateSphereSignal UpdateSphereSignal { get; set; }

    protected override void UpdateModel()
    {
        GetComponent<Renderer>().material.SetColor("_Color",
Model.sphereColor);
        transform.localScale = Vector3.one * Model.sphereSize;
    }
}
```

This lets Rapid know, that as soon **UpdateSphereSignal** is binded, it will be injected in all views with **UpdateSphereSignal** property.

Notice that our **[Inject]** attribute is parameterless (it can also take **keyData** as a parameter to identify specific binding by a unique key, you can read more on keys here. In this case empty **keyData**, means binding will be identified by its value type, i.e. **UpdateSphereSignal** type.

12. Now that our signal is injected, we still need to actually update the model. For that we will use Injected Signal Mapping. Add **OnUpdateSphere** method that will update the **SphereView**'s model with that from our signal:

```
using cpGames.core.RapidIoC;
using UnityEngine;

public class SphereView : ComponentModelView<SphereModel>
{
    [Inject]
    public UpdateSphereSignal UpdateSphereSignal { get; set; }

    public void OnUpdateSphere(SphereModel model)
    {
        Model = model;
        UpdateModel();
    }

    protected override void UpdateModel()
    {
        GetComponent<Renderer>().material.SetColor("_Color",
Model.sphereColor);
        transform.localScale = Vector3.one * Model.sphereSize;
    }
}
```

OnUpdateSphere will be called automatically every time **UpdateSphereSignal** is dispatched. As mentioned earlier, how this works can be found here.

13. Finally we need to create a controller that will update our model. In the same **view** folder create **SphereControllerView** script. It should also inject **UpdateSphereSignal**, but this time it will dispatch it. We will tween **sphereSize** and **sphereColor** between two values in a loop:

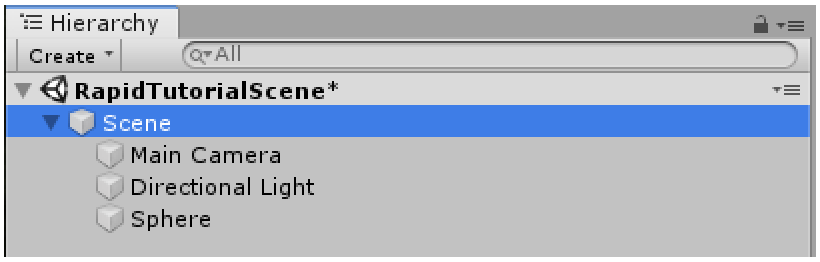
```
using System.Collections;
using cpGames.core.RapidIoC;
using UnityEngine;

public class SphereControllerView : ComponentView
{
    private readonly SphereModel _model = new SphereModel();
    private float _timeIntoAnimation = 0f;

    public float updateInterval = 1.0f;
    public float animationDuration = 10.0f;
    public Color colorStart = Color.white;
    public Color colorEnd = Color.red;
    public float sizeStart = 1.0f;
    public float sizeEnd = 10.0f;

    [Inject]
    public UpdateSphereSignal UpdateSphereSignal { get; set; }

    private void Start()
    {
        StartCoroutine(UpdateSphereModel());
    }

    private IEnumerator UpdateSphereModel()
    {
        _model.sphereColor = Color.Lerp(colorStart, colorEnd,
_timeIntoAnimation / animationDuration);
        _model.sphereSize = Mathf.Lerp(sizeStart, sizeEnd,
_timeIntoAnimation / animationDuration);
        UpdateSphereSignal.Dispatch(_model);
        yield return new WaitForSeconds(updateInterval);
        _timeIntoAnimation = (_timeIntoAnimation + updateInterval) %
animationDuration;
        StartCoroutine(UpdateSphereModel());
    }
}
```

(Note: I have a free tweening library here which can be used for more complex problems if you wish to check it out)

14. Now our coding is complete. Let's setup a scene in Unity. Create a new gameobject, call it **Scene**. Add **RapidTutorialSceneView** component to it. This will be our root object in every scene and **every view that needs to find the local context, needs to be parented to it**. Let's parent existing camera and light to it as well for convenience.

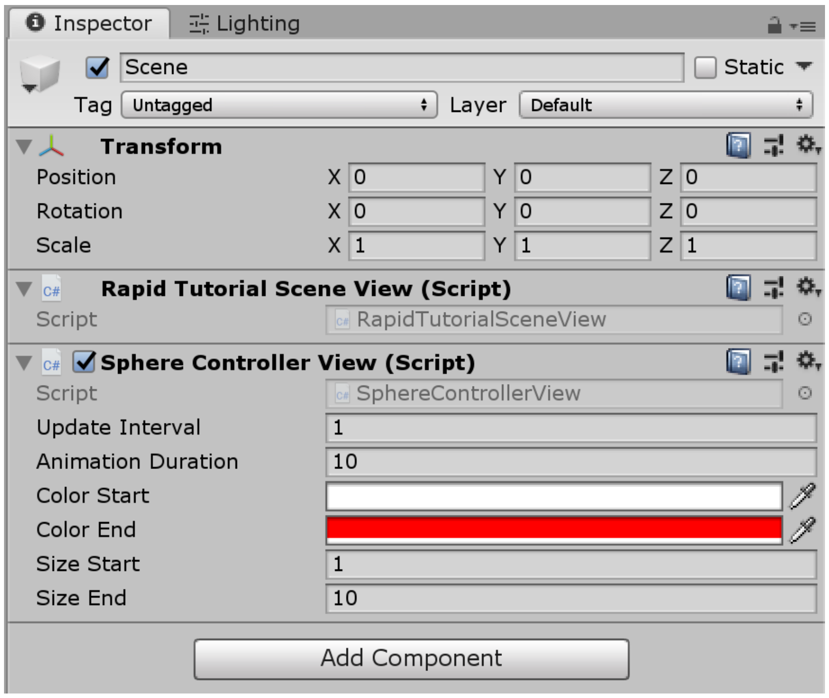
15. Create a sphere child (right-click on **Scene**, 3D Object->Sphere). Add **SphereView** component to your sphere. Finally attach **SphereControllerView** component to your **Scene** object.

Your scene hierarchy should look like this:



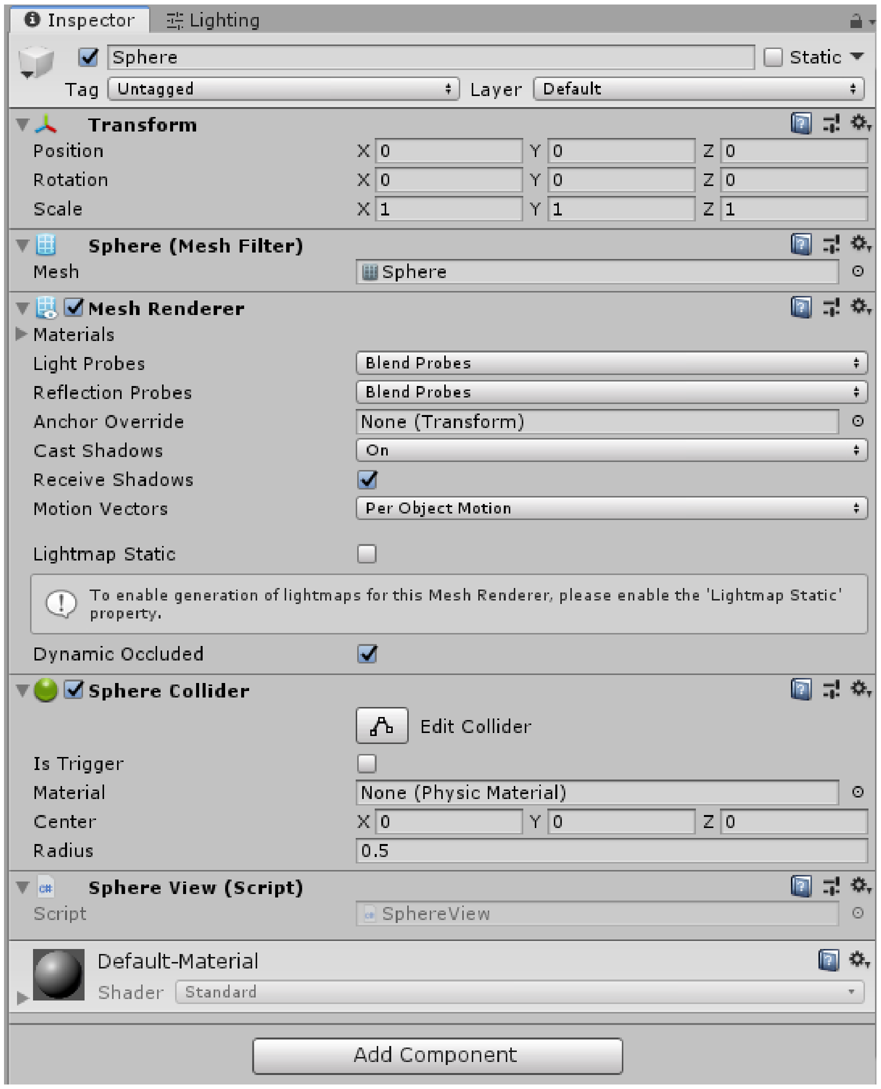
```
RapidTutorialScene*
  Scene
    Main Camera
    Directional Light
    Sphere
```

Your Scene components should look like this:



```
Scene
Tag Untagged    Layer Default

Transform
Position    X 0   Y 0   Z 0
Rotation    X 0   Y 0   Z 0
Scale       X 1   Y 1   Z 1

Rapid Tutorial Scene View (Script)
Script              RapidTutorialSceneView

Sphere Controller View (Script)
Script              SphereControllerView
Update Interval     1
Animation Duration  10
Color Start         ⬜
Color End           🟥
Size Start          1
Size End            10
```

Your Sphere components should look like this:



Sphere [✓] Static [ ]
Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Sphere (Mesh Filter)
Mesh: Sphere

Mesh Renderer [✓]

Materials	
Light Probes	Blend Probes
Reflection Probes	Blend Probes
Anchor Override	None (Transform)
Cast Shadows	On
Receive Shadows	✓
Motion Vectors	Per Object Motion
Lightmap Static	☐

To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.

Dynamic Occluded ✓

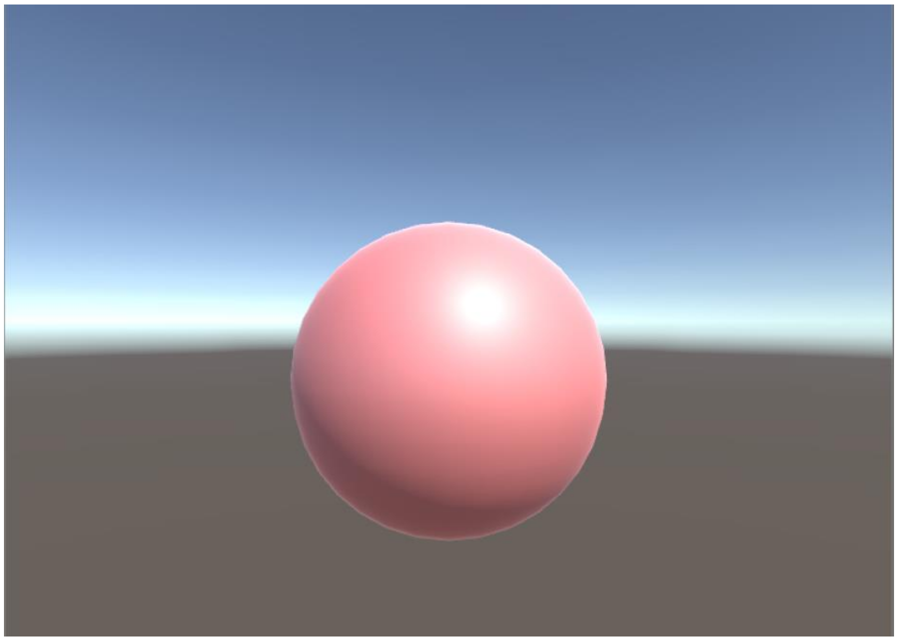
Sphere Collider [✓]

Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.5

Sphere View (Script)
Script: SphereView

Default-Material
Shader: Standard

16. Finally if you have everything setup correctly and press Play, your sphere should increase in 1s intervals and slowly become redder:



This is the end of this tutorial.

Rapid may seem like overly complicated for doing something as simple as we've covered here, however once the project increases in size and complexity, the rigid structure provided can be very helpful for keeping your code manageable.