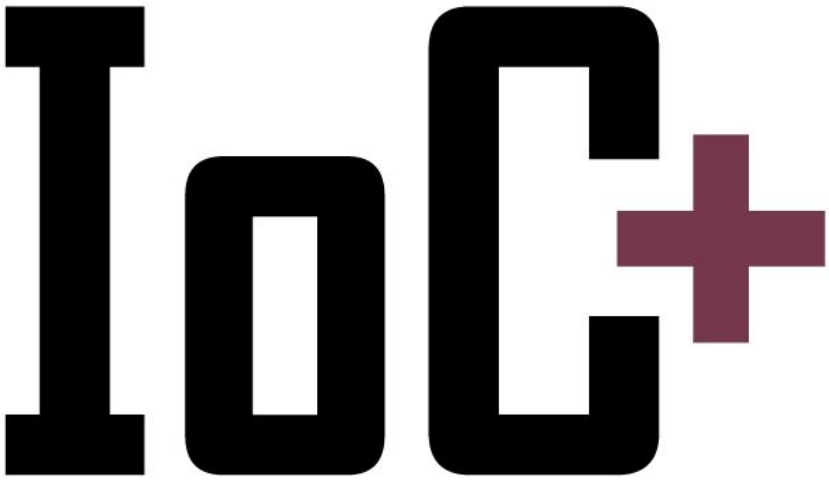


IoC for Unity

Offline Documentation

Table of Contents

Introduction

If you are new to the Inversion of Control (IoC) pattern then you're in the right spot. This section explains the basics of IoC, gives an introduction to IoC+ and explains what IoC+ adds to the standard IoC foundation so it can serve as much more.

IoC is a design principle that can be used as a foundation for an entire game or application. It focuses on keeping classes decoupled so they can easily be managed, extended and unit tested. IoC+ consists out of 5 component types: mediators, views, commands, contexts and signals.

Views are objects we can see and/or interact with. A mediator controls a view and a command performs a task. The mediators and commands use signals to communicate. Contexts link the other components together and inject dependencies into the mediators and commands. Having mediators, commands and their dependencies decoupled means that they can easily be swapped, reused and unit tested.

Added value of working with Inversion of Control:

- Sets the foundation for a decoupled, modular code base
- Keeps classes focused on what they are designed for
- Great for both individuals and teams
- Enforces a clear structure for projects of any scale
- Allows for maximum unit testing

The following explains the components of IoC+ in more detail.

View

A view can display an object, receive user input and/or check for collisions. It is controlled by a mediator. Views are the only Unity MonoBehaviours in IoC+

Mediator

A mediator controls a view. It listens to signals from commands and its view and makes the view act accordingly. In IoC+, the type of mediator that is attached to a view is based on bindings set in a context.

Command

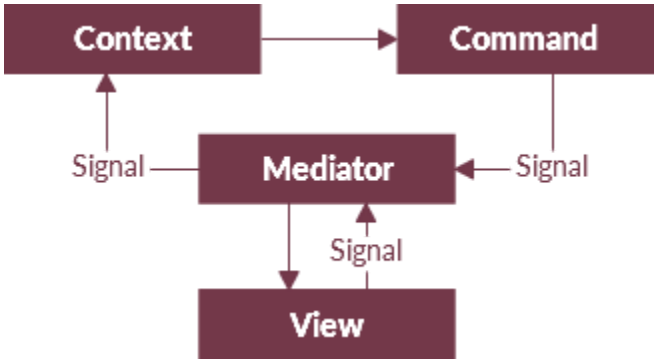
A command is preferably made to do only one thing. Like instantiating a view, loading a save-file or opening a door. In IoC+, commands are executed as a response to a signal, defined in a context.

Context

A context sets injection values, associates mediators to views and assigns commands as responses to signals. In IoC+, contexts are able to have child contexts and can easily be switched during run-time.

Signal

Signals allow views to trigger their mediator, mediators to trigger commands and commands to trigger mediators. In IoC+, signals are scoped based on the context that binds them in its injection bindings.



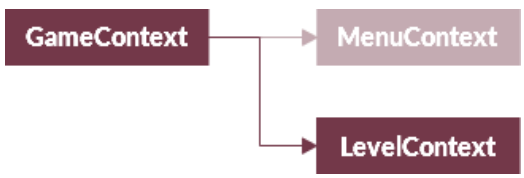
Why we made IoC+

We've been focusing a lot on code quality and find that patterns like MVC and IoC are a great foundation to enforce quality in code. A framework that enforces such a pattern is a great benefit for both beginning and skilled programmers. IoC+ takes inspiration from our previous work, Code Control (MVC for Unity), and from our experience using the StrangeIoC framework, which does a great job implementing IoC in Unity. IoC+ builds on those and aims to be the most complete IoC framework for Unity.

Why use IoC+

In traditional IoC development, a bunch of contexts are set to define a game's (or application's) behavior but the set of contexts do not change during run-time. The contexts live next to each other, their bindings can't be scoped and contexts generally do not overwrite each other's bindings. For applications this is usually enough, but games often require more control.

IoC+ makes it easy to instantiate contexts during run-time. A game can have multiple context instances that each control its own, for example, robot without having to worry about its signals leaking to other contexts as the injections are isolated in each context. It's also possible to have nested contexts in IoC+. The robot contexts could all live in a parent level context. This level context could listen for any robot to be hit and make all robots act accordingly as signal injection bindings of parent contexts will be injected in each child context.



Another feature that makes IoC+ unique is its combination of the finite state machine pattern and IoC. A child context can act as its parent's state. With simple commands this state can be changed, calling the state's enter / leave signal accordingly to

conveniently enter or leave game menu's, robot behaviors or even switch entire levels. This is a big benefit for games.

Last but not least, everything that happens in IoC+ can be monitored using the IoC+ context monitor. The monitor gives a live representation of the present contexts, their bindings and their parent/child/state relations. Signals and commands light up when dispatched or executed and signals can even be dispatched interactively from the monitor. The monitor can pause to walk through its history step by step and associated C# code files can conveniently be opened. The IoC+ context monitor really helps keeping track of contexts and is a big help during code reviews and knowledge transfer when working in teams.

To summarize, here are some of the added values of working with IoC+ when using Inversion of Control:

- Easy to add / remove contexts during run-time
- Nested contexts with isolated context injections
- Switchable contexts during run-time to change behavior
- Integrated Finite State Machine in Inversion of Control
- Live monitoring and interaction in Unity Editor

Your first context

Introduction

This tutorial teaches how to define and instantiate a context which will use a predefined signal to execute a command. A context is a container of injection, command and mediation bindings. Contexts are responsible for binding modules of IoC together.

The Context

Open up a fresh new Unity Project and import the IoC+ framework from the Asset Store. Create a new C# script called “GameContext.cs”. This code file will contain the context of our little example. Add the following code.

```
1  using IoCPlus;
2
3  public class GameContext : Context {
4
5      protected override void SetBindings() {
6          base.SetBindings();
7      }
8
9  }
```

Our own contexts are always a subclass of the Context class. In the override SetBindings() method we can set additional bindings, but notice that we call the SetBindings() of the base class. Some bindings are set by default. One of those bindings is the EnterContextSignal.

The EnterContextSignal is a signal that will dispatch when the context is instantiated. In the SetBindings() method, we can bind commands as a response to signals.

The Command

Let’s first define a command. Create a new script called “WelcomeCommand.cs” and add the following code.

```
1  using IoCPlus;
2  using UnityEngine;
3
4  public class WelcomeCommand : Command {
5
6      protected override void Execute() {
7          Debug.Log("Welcome to the Game Context!");
8      }
9
10 }
```

Commands in IoC+ are always a subclass of the Command class. In the override Execute() method we write the code that the command needs to execute. In this case, a little debug log to test whether our command is being executed as a response to our signal.

Binding the Command

In IoC+ we can set bindings in Contexts. There are three types of bindings: injection bindings, mediator bindings and command bindings. We'll cover all bindings in upcoming tutorials. In this tutorial we'll focus on command bindings!

Let's create a command binding now to bind the WelcomeCommand as a response to the EnterContextSignal. Open up the GameContext and add the following code.

```
1  using IoCPlus;
2
3  public class GameContext : Context {
4
5      protected override void SetBindings() {
6          base.SetBindings();
7
8          On<EnterContextSignal>().Do<WelcomeCommand>();
9      }
10
11 }
```

By calling the On<T>() method, we define a signal type that we want to respond to. We can use the result of the On<T>() method to add commands as a response to that signal via the Do<T>() method. In this case, when the context is entered, the WelcomeCommand will be executed.

The Root Context

If we'd hit the Unity play button right now, we'd still get nothing. This is because the GameContext still has to be instantiated. To easily instantiate the first context of a game or application in IoC+, we use a ContextRoot. Create a new script called "GameRoot.cs" and add the following code.

```
1  using IoCPlus;
2
3  public class GameRoot : ContextRoot<GameContext> { }
```

We make our GameRoot a subclass of the ContextRoot<T> class, where we set T to the context type that we want to instantiate: the GameContext. In the Unity scene, create a new empty gameobject. Add the GameRoot script as a component to the gameobject. This gameobject will now instantiate our GameContext on awake.

Go ahead and hit play! If you've followed the steps correctly, you should now see our welcome text pop up in Unity's debugger.

The IoC+ Monitor

Want to see something cool? Open up the IoC+ Monitor via Window -> IoC+ Monitor in the top bar of Unity and hit play!

The IoC+ Monitor gives a live feed of all contexts in game. It shows our instance of the GameContext, flashes the EnterContextSignal and then flashes the WelcomeCommand as it is triggered as a response.

Why does my IoC+ Monitor not show the same things as in the image?
The IoC+ Monitor has a set of options that allows the user to tweak what is displayed. In the top-left of the IoC+ Monitor there are three time options: “Sync”, “Delay” and “Pause”. The tutorial images are all shown in Delay mode. In the top-right of the IoC+ Monitor there is a button for even more options. Make sure to click it and turn on all display options to get the same result as the images in the tutorials.

We can right-click the context, signal and command to open the associated code file. By using the right-click we can even dispatch the EnterContextSignal again!

Why does my IoC+ Monitor not open the correct code file when I right-click?
When opening a code file the IoC+ Monitor tries to open a file with the same name as the type (e.g. “GameContext”). Namespaces make it possible to have multiple types with the same name. In this case it is undefined which code file will be opened. Delete the other tutorials from the IoC+ package to be sure that the correct code files are opened.

Conclusion

In this tutorial we’ve seen how to declare a context and a command. Contexts are there to link IoC modules together and we’ve seen how to bind a command as a response to the EnterContextSignal, a default signal from IoC+. We’ve made a context root to instantiate the root

context of our game. Last but not least, we've used the IoC+ Monitor to visualize and test our creations in real time!

Binding an Injection

Introduction

This tutorial teaches how to bind an injection via a context. Injections are values bound to a type that a mediator or command can depend upon. These values are “configured” in the context. The injection process is called Dependency Injection.

Adding a Dependency

In our WelcomeCommand we are currently logging the text “Welcome to the Game Context!”. Let’s say we want it to log something else, but we want to configure what we log in the GameContext so we can potentially reuse the WelcomeCommand in different contexts. To do this, we are going to add a dependency in the WelcomeCommand that will be injected by the GameContext. Update the WelcomeCommand to the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class WelcomeCommand : Command {
5
6       [Inject] string welcomeText;
7
8       protected override void Execute() {
9           Debug.Log(welcomeText);
10      }
11
12  }
```

We’ve replaced the hard-coded welcome text with a variable that we assigned with an Inject attribute. This attribute does the magic of injecting the value into this variable that is bound to the string type in the context.

Binding a Value

Let’s open up our context and bind a value to the string type.

```
1   using IoCPlus;
2
3   public class GameContext : Context {
4
5       protected override void SetBindings() {
6           base.SetBindings();
7
8           Bind<string>("Welcome! Look at me, I'm injecting a value!");
9
10          On<EnterContextSignal>().Do<WelcomeCommand>();
11      }
```

```
12
13  }
```

In the override SetBindings() method, we call the Bind<T>() method to bind a value to the string type. All string typed variables in commands and mediators with the Inject attribute will now be set to the “Welcome! Look at me, I’m injecting a value!” value! Go ahead and hit play, the debugger should log our injected value.

Isn’t this awesome? We’ve decoupled the welcome text value from the WelcomeCommand class, so we can potentially use the command in another context with a different welcome text! We just have to configure the value in its context. Dependency injection doesn’t stop at string values; its true power lies in injecting signals, models and services. These injections are covered in later tutorials.

The IoC+ Monitor

Our new injection binding will be visible in the IoC+ Monitor. Go ahead and open it up, you’ll the String type under the list of injections.

If you don’t see the injection bindings, make sure “Show Injection Bindings” is enabled in the Options of the IoC+ Monitor.

Conclusion

In this tutorial we’ve seen how to add a dependency to a command by adding the inject attribute in front of the field. We learned how to extend the context to bind a value to a string type so that all string typed dependencies in commands and mediators are set to that value. Lastly, we’ve seen how the IoC+ Monitor show us what injection bindings are set in the context.

Views and Mediators

Introduction

This tutorial teaches how to make and instantiate a view and how to bind a mediator to it. Views are, next to the ContextRoot, the only MonoBehaviours in IoC+ and therefore the only real connection to the Unity Engine. They visualize an object, process user input or maybe even check for collisions. Mediators link the views to the IoC framework. A mediator listens to its view via signals and make its view act according to signals dispatched by commands.

The View

Views are what we as players of our game can see and interact with. In IoC+ they are the only real connection to the Unity Engine and the only part that we cannot fully unit test. Therefore, we need to make the views as simple and straightforward as possible. Let's make a view that handles the player's input to make a character jump. Create a new script file called "PlayerInputView.cs" and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class PlayerInputView : View {
5
6       public readonly Signal JumpInputSignal = new Signal();
7
8       private void Awake() {
9           Debug.Log("PlayerInputView is instantiated!");
10      }
11
12      private void Update() {
13          if (Input.GetKeyDown(KeyCode.UpArrow)) {
14              JumpInputSignal.Dispatch();
15          }
16      }
17
18  }
```

All views in IoC+ must subclass the View class. This PlayerInputView declares a public signal called JumpInputSignal. In its Update() method, it checks for the up arrow key to be pressed and if so, dispatches the JumpInputSignal. In its Awake() method it logs a short message so we can test whether the view has been instantiated.

The Mediator

Views will need a mediator to control it. Mediators link the views to our IoC framework of contexts and commands. Create a new script file called "PlayerInputMediator.cs" and add the following code.

```
using IoCPlus;

public class PlayerInputMediator : Mediator<PlayerInputView> {

    public override void Initialize() {
        view.JumpInputSignal.AddListener(OnViewJumpInputSignal);
    }

    public override void Dispose() {
        view.JumpInputSignal.RemoveListener(OnViewJumpInputSignal);
    }

    private void OnViewJumpInputSignal() {
        // Dispatch a signal to execute a command
    }

}
```

All mediators in IoC+ must subclass the Mediator<T> class, where we define the type of view that we want to mediate. The instance of the view that a mediator controls is assigned automatically. In the override Initialize() method we add a listener to the view's JumpInputSignal and make it call the OnViewJumpInputSignal() method once it is dispatched. In the Dispose() method we make sure we stop listening to the signal once this mediator is removed, which is considered a good practice even though we're not yet removing any mediators from views.

Binding the View to the Mediator

Now that we have both the view and the mediator we'll have to tell our context we want the PlayerInputMediator to control our PlayerInputView. We do this by adding the following code to the GameContext.

```
using IoCPlus;

public class GameContext : Context {

    protected override void SetBindings() {
        base.SetBindings();

        Bind<string>("Welcome! Look at me, I'm injecting a value!");

        BindMediator<PlayerInputMediator, PlayerInputView>();

        On<EnterContextSignal>().Do<WelcomeCommand>();
    }

}
```

The BindMediator<T,U>() allows us to set what mediator we want to bind to a view type. All PlayerInputView instances that would fall under this context will now be controlled by a PlayerInputMediator instance.

Instantiating a View

That's all fine and dandy, but we're still not seeing any view when we hit play! We still need to instantiate the view. Let's make a command to do that for us. Add a new script file called "InstantiatePlayerInputCommand.cs" and add the following code.

```
using IoCPlus;

public class InstantiatePlayerInputCommand : Command {

    [Inject] IContext context;

    protected override void Execute() {
        context.InstantiateView<PlayerInputView>();
    }

}
```

This command has a dependency on the context from which it is executed. This injection binding is set by default via the base.SetBindings() in the Context. Using the context instance we can instantiate a view in that context using the InstantiateView<T>() method. This will create a new gameobject, add the PlayerInputView component to it and assign a mediator based on the context's mediator bindings. In our case this will be the PlayerInputMediator.

Let's execute this command when the context is entered. Add the following code to the GameContext.

```
using IoCPlus;

public class GameContext : Context {

    protected override void SetBindings() {
        base.SetBindings();

        Bind<string>("Welcome! Look at me, I'm injecting a value!");

        BindMediator<PlayerInputMediator, PlayerInputView>();

        On<EnterContextSignal>().Do<WelcomeCommand>()
                                .Do<InstantiatePlayerInputCommand>();
    }

}
```

As you can see, we can bind multiple commands as a response to one signal. These commands will be executed in the same order that we bind them. Go ahead and hit the play button. The view should now be instantiated and visible in Unity's scene hierarchy. We should also get the log message that the InputPlayerView outputs in its Awake() method.

Injecting a Custom Signal

But we're not done yet. In the PlayerInputMediator's OnViewJumpInputSignal() method we would like the context to execute a command, so we can respond to the input of the player. The mediator depends on a signal that needs to be injected. Let's first declare a new signal type. Create a new script called "JumpInputSignal.cs" and add the following code.

```
1  using IoCPlus;
2
3  public class JumpInputSignal : Signal { }
```

Now that we have a custom signal type, we can use this type to bind an injection via the context. Open up the GameContext and add the following code.

```
1  using IoCPlus;
2
3  public class GameContext : Context {
4
5      protected override void SetBindings() {
6          base.SetBindings();
7
8          Bind<string>("Welcome! Look at me, I'm injecting a value!");
9          Bind<JumpInputSignal>();
10
11         BindMediator<PlayerInputMediator, PlayerInputView>();
12
13         On<EnterContextSignal>().Do<WelcomeCommand>()
14                                 .Do<InstantiatePlayerInputCommand>();
15     }
16
17 }
```

Notice that in the Bind<T>() method for the JumpInputSignal we are not setting any value. This is a shorthand method; the IoC+ framework will automatically set the value to a new instance of the given type. Keep in mind that the same instance is injected in all commands and mediators that depend on it.

Dispatching an Injected Signal

The signal will now be injected in all commands and mediators that depend on it, so let's extend our PlayerInputMediator with the following.

```
1  using IoCPlus;
2
3  public class PlayerInputMediator : Mediator<PlayerInputView> {
4
5      [Inject] JumpInputSignal jumpInputSignal;
6
7      public override void Initialize() {
8          view.JumpInputSignal.AddListener(OnViewJumpInputSignal);
```

```
9       }
10
11      public override void Dispose() {
12          view.JumpInputSignal.RemoveListener(OnViewJumpInputSignal);
13      }
14
15      private void OnViewJumpInputSignal() {
16          jumpInputSignal.Dispatch();
17      }
18
19  }
```

The PlayerInputMediator uses the injected JumpInputSignal from the context to dispatch it once its view says it is time to. Keep in mind that there is a big difference between the signal of the view and the signal that is injected by the context. Signals from a view are only listened to by its mediator. The injected signal in the mediator will be listened to by the context to execute a command.

Let's make a simple command to, for now, test our signal. Create a new script file named "JumpCommand.cs" and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class JumpCommand : Command {
5
6       protected override void Execute() {
7           Debug.Log("Jump!");
8       }
9
10  }
```

Now that we have the JumpCommand we can make the GameContext bind it as a response to the JumpInputSignal. Open up the GameContext and add the following code.

```
1   using IoCPlus;
2
3   public class GameContext : Context {
4
5       protected override void SetBindings() {
6           base.SetBindings();
7
8           Bind<string>("Welcome! Look at me, I'm injecting a value!");
9           Bind<JumpInputSignal>();
10
11          BindMediator<PlayerInputMediator, PlayerInputView>();
12
13          On<EnterContextSignal>().Do<WelcomeCommand>()
14                                  .Do<InstantiatePlayerInputCommand>();
15
16          On<JumpInputSignal>().Do<JumpCommand>();
17      }
```

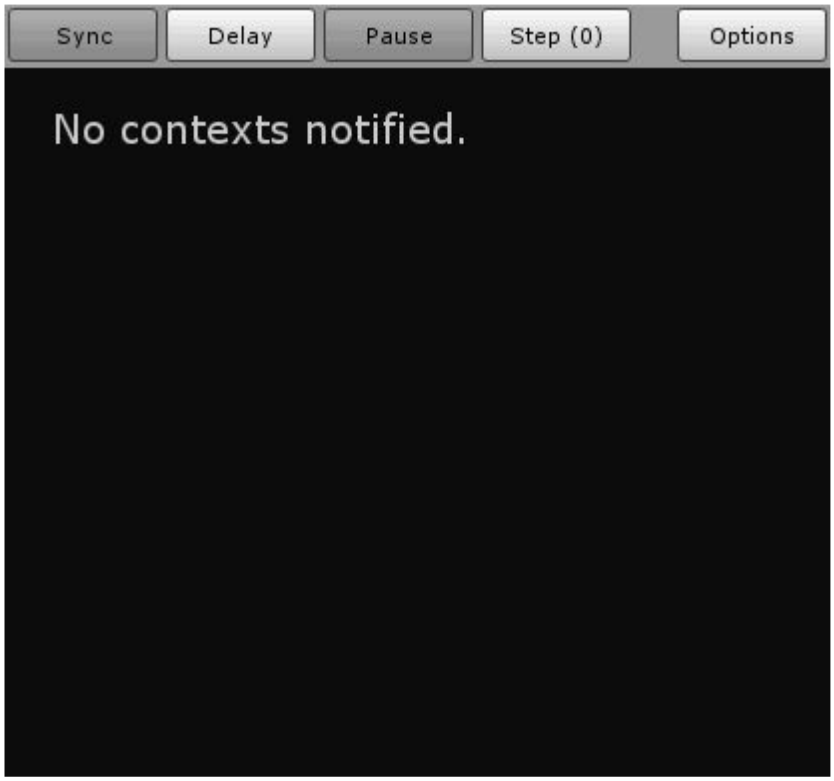
```
18
19  }
```

There! Now, when the PlayerInputView notices the arrow up key being pressed, it dispatches its JumpInputSignal, making the PlayerInputMediator dispatch its injected JumpInputSignal, making the GameContext perform the JumpCommand. Go ahead, hit the play button and press the arrow up key a couple of times. You should get the jump message that is logged by the JumpCommand.

Sure seems like a lot of classes to produce a single log on a key press!
At first, it does seem like a lot of unnecessary steps to output a simple debug log. Without the use of IoC+, you could get the same result with only one script file! This is true, but as your project gets bigger and requires more and more design changes, it will become a real benefit to have your classes decoupled like this. In IoC+, you can even switch a context with another context to dynamically ignore the player's input and trigger the JumpCommand based on something else entirely!

The IoC+ Monitor

Let's review our creations in the IoC+ Monitor!



Woah, a bunch of new stuff is added! We can see that the JumpInputSignal is added to the list of injections. The InstantiatePlayerInputCommand is added as a second response to the EnterContextSignal and the JumpCommand is set as a response to the JumpInputSignal. In the

list of mediators we can see that the PlayerInputView is set to be controlled by the PlayerInputMediator.

The PlayerInputMediator instance that is created to control our instantiated PlayerInputView is displayed under Mediator Instances so we can see in which context the mediator currently resides. Notice how the instance shows up only after the InstantiatePlayerInputCommand is executed.

Last but not least, when we press the arrow up key the JumpInputSignal flashes in both the commands and the injections list. This will come of great use once we start nesting contexts.

Conclusion

In this tutorial we've learned how to make a view and a mediator. We use the BindMediator<T,U>() method in a context's SetBindings() method to set the mediator type that will control a view type. We've learned that views are the only real connection to the Unity Engine and that mediators link the view to the IoC framework. Views use signals to communicate to their mediator and mediators dispatch injected signals, executing commands that are set as a response to those signals in the context.

Command to Mediator

Introduction

This tutorial teaches how to use a signal in a command to make a mediator act accordingly.

A Prefab View

In the previous tutorial we've instantiated the PlayerInput view as a new empty gameobject. Let's create a view that actually displays the player and make it jump based on the player's input. Create a cube in your scene and add a rigidbody to it. Leave all options as default, so the cube will fall down on play.

Create a new script file called "PlayerView.cs" and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class PlayerView : View {
5
6       Rigidbody myRigidbody;
7
8       public void Jump() {
9           myRigidbody.AddForce(Vector3.up * 100.0f);
10      }
11
12      private void Awake() {
13          myRigidbody = GetComponent<Rigidbody>();
14      }
15
16  }
```

A simple view that has only one functionality and that is to jump. Add the PlayerView script to the cube we've just created. Rename the cube to "Player", turn it into a prefab and place the prefab in a folder called "Resources" so we can load it from code.

The Jump Signal

Before we implement its mediator, let's first add a signal that will actually make the player jump. Create a new script file called "JumpSignal.cs" and add the following code.

```
1   using IoCPlus;
2
3   public class JumpSignal : Signal { }
```

Remember the JumpInputSignal? That signal was named after an event, because it was triggered in the player's jump input. This time the signal is named after an action, the "jump" action. Naming signals as event or action accordingly is a great convention to give insight as to

whether we're using the signal to trigger a set of commands or we're using the signal in a command to make a mediator act accordingly.

Let's first bind the JumpSignal and the PlayerView and PlayerMediator in the GameContext. Note that we haven't made our PlayerMediator yet! Open up the GameContext and add the following code.

```
using IoCPlus;

public class GameContext : Context {

    protected override void SetBindings() {
        base.SetBindings();

        Bind<string>("Welcome! Look at me, I'm injecting a value!");
        Bind<JumpInputSignal>();
        Bind<JumpSignal>();

        BindMediator<PlayerInputMediator, PlayerInputView>();
        BindMediator<PlayerMediator, PlayerView>();

        On<EnterContextSignal>().Do<WelcomeCommand>()
                                .Do<InstantiatePlayerInputCommand>();

        On<JumpInputSignal>().Do<JumpCommand>();
    }

}
```

Now that the JumpSignal is bound we can inject it into our commands and mediators.

Listening to Signals from Commands

Until now we've only seen mediators listen to signals from their view. This time we want the PlayerView's mediator to listen to a signal that is potentially dispatched from a command. Create a new script file called "PlayerMediator.cs" and add the following code.

```
using IoCPlus;

public class PlayerMediator : Mediator<PlayerView> {

    [Inject] JumpSignal jumpSignal;

    public override void Initialize() {
        jumpSignal.AddListener(OnJumpSignal);
    }

    public override void Dispose() {
        jumpSignal.RemoveListener(OnJumpSignal);
    }

```

```
15      private void OnJumpSignal() {
16          view.Jump();
17      }
18
19  }
```

We inject the JumpSignal and start listening to it in the Initialize method. As we know, mediators in IoC+ automatically have a reference to their view. When the signal is dispatched, we make the PlayerView jump. Clean and simple.

Instantiating the Prefab View

Let's instantiate our player first. Create a new script file called "InstantiatePlayerCommand.cs" and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class InstantiatePlayerCommand : Command {
5
6       [Inject] IContext context;
7
8       protected override void Execute() {
9           PlayerView prefab = Resources.Load<PlayerView>("Player");
10          context.InstantiateView(prefab);
11      }
12
13  }
```

We first load our PlayerView from the Resources folder. Then we make our context instantiate the view based on the prefab. Prefabs may contain multiple view components using this method. Let's add this command as a response to the EnterContextSignal in the GameContext by adding the following code.

```
1   using IoCPlus;
2
3   public class GameContext : Context {
4
5       protected override void SetBindings() {
6           base.SetBindings();
7
8           Bind<string>("Welcome! Look at me, I'm injecting a value!");
9           Bind<JumpInputSignal>();
10          Bind<JumpSignal>();
11
12          BindMediator<PlayerInputMediator, PlayerInputView>();
13          BindMediator<PlayerMediator, PlayerView>();
14
15          On<EnterContextSignal>().Do<WelcomeCommand>()
16                                  .Do<InstantiatePlayerInputCommand>()
17                                  .Do<InstantiatePlayerCommand>();
```

```
18
19          On<JumpInputSignal>().Do<JumpCommand>();
20      }
21
22  }
```

Go ahead and hit play. The player view should now be instantiated on startup, and fall right down because of its rigidbody.

Can't see the player? Make sure to set the position of the prefab to (0, 0, 0), as the instance might be out of camera view.

Dispatching the Signal from a Command

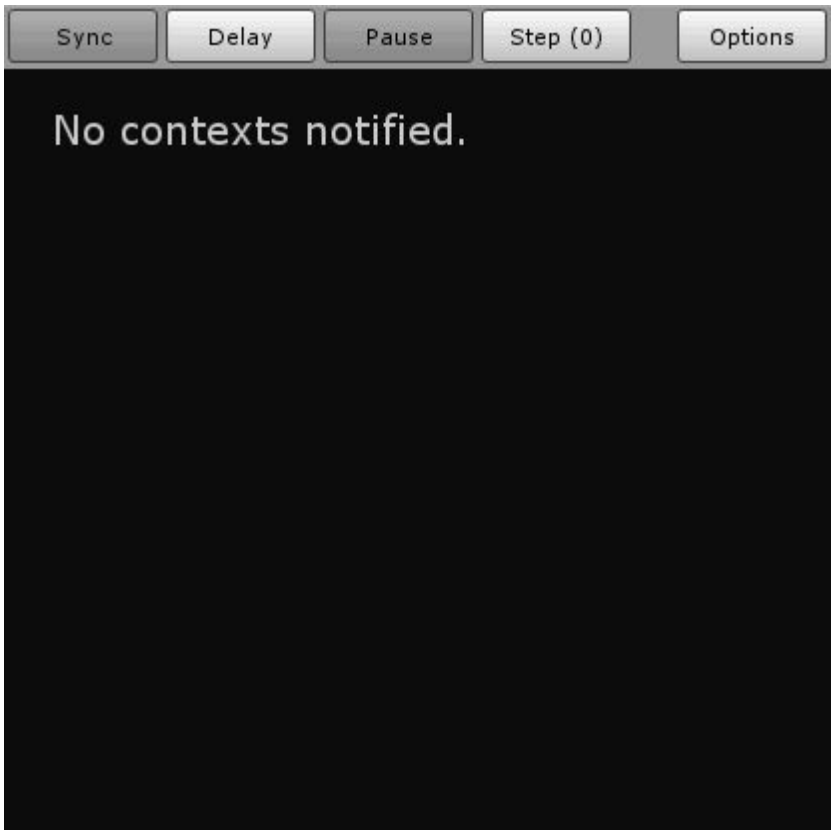
Let's open up our JumpCommand and update it with the following code.

```
1   using IoCPlus;
2
3   public class JumpCommand : Command {
4
5       [Inject] JumpSignal jumpSignal;
6
7       protected override void Execute() {
8           jumpSignal.Dispatch();
9       }
10
11  }
```

There! We inject the JumpSignal, the same signal as the PlayerMediator injects, and dispatch it on execution. The mediator should now respond to the signal from this command. Go ahead and hit play! Once we hit the up arrow key the player will start jump up, so we can try and keep the player in the air.

The IoC+ Monitor

Signals dispatched from commands will also light up in the IoC+ Monitor. Go ahead and open it up, you'll see the JumpSignal light up on the JumpCommand's execution.



That’s a lot of insight for one context! Be sure to use the Options button in the IoC+ Monitor to filter the monitor’s output as your contexts get bigger.

Conclusion

In this tutorial we’ve seen how to dispatch a signal from a command and listen to the same signal in a mediator. We’ve learned about naming a signal as either an event or an action to give insight of its usage. We’ve also seen that the InstantiateView<T>() method can also be used to instantiate a view from a prefab.

Signal Parameters

Introduction

This tutorial teaches how to add parameters to a signal to configure a commands execution.

The Move Signal

Now that our player view can jump and fall down, let's also add the ability to move horizontally both left and right. Instead of making two signals and two commands to accomplish this we're going to make one single move signal with a parameter to define its direction.

Create a new script file called "MoveInputSignal.cs" and add the following code.

```
1  using IoCPlus;
2  using UnityEngine;
3
4  public class MoveInputSignal : Signal<Vector3> { }
```

We use the generic Signal<T> class to define our direction parameter. IoC+ signals can go up to three parameters. To store more data in the parameters, simply make a class or struct and use that as a parameter.

Binding signals with parameters works exactly the same as with signals without parameters. Open up the GameContext and update it with the following code.

```
1  using IoCPlus;
2
3  public class GameContext : Context {
4
5      protected override void SetBindings() {
6          base.SetBindings();
7
8          Bind<string>("Welcome! Look at me, I'm injecting a value!");
9          Bind<JumpInputSignal>();
10         Bind<MoveInputSignal>();
11         Bind<JumpSignal>();
12
13         BindMediator<PlayerInputMediator, PlayerInputView>();
14         BindMediator<PlayerMediator, PlayerView>();
15
16         On<EnterContextSignal>().Do<WelcomeCommand>()
17                                 .Do<InstantiatePlayerInputCommand>()
18                                 .Do<InstantiatePlayerCommand>();
19
20         On<JumpInputSignal>().Do<JumpCommand>();
21     }
22
23 }
```

Now that the signal is bound we can inject it into commands and mediators.

Dispatching the input Signal

Let's open up our PlayerView script to handle the player's input to let the mediator know when to dispatch the signal. Add the following code.

```
using IoCPlus;
using UnityEngine;

public class PlayerInputView : View {

    public readonly Signal JumpInputSignal = new Signal();
    public readonly Signal<Vector3> MoveInputSignal = new
Signal<Vector3>();

    private void Awake() {
        Debug.Log("PlayerInputView is instantiated!");
    }

    private void Update() {
        if (Input.GetKeyDown(KeyCode.UpArrow)) {
            JumpInputSignal.Dispatch();
        }
        if (Input.GetKeyDown(KeyCode.LeftArrow)) {
            MoveInputSignal.Dispatch(Vector3.left);
        }
        if (Input.GetKeyDown(KeyCode.RightArrow)) {
            MoveInputSignal.Dispatch(Vector3.right);
        }
    }

}
```

Signals used in views can also contain parameters by using the same generic class. In the Update() method we check if the left or right arrow is pressed. If so, we dispatch the MoveInputSignal. Because the MoveInputSignal is a Signal<Vector3> we can only dispatch it with a Vector3 parameter, so we include the Vector3.left or Vector3.right accordingly.

Let's open up the PlayerInputMediator and add the following code.

```
using IoCPlus;
using UnityEngine;

public class PlayerInputMediator : Mediator<PlayerInputView> {

    [Inject] JumpInputSignal jumpInputSignal;
    [Inject] MoveInputSignal moveInputSignal;

    public override void Initialize() {
```

```
10          view.JumpInputSignal.AddListener(OnViewJumpInputSignal);
11          view.MoveInputSignal.AddListener(OnViewMoveInputSignal);
12      }
13
14      public override void Dispose() {
15          view.JumpInputSignal.RemoveListener(OnViewJumpInputSignal);
16          view.MoveInputSignal.RemoveListener(OnViewMoveInputSignal);
17      }
18
19      private void OnViewJumpInputSignal() {
20          jumpInputSignal.Dispatch();
21      }
22
23      private void OnViewMoveInputSignal(Vector3 direction) {
24          moveInputSignal.Dispatch(direction);
25      }
26
27  }
```

We repeat the same process: inject the MoveInputSignal, listen to our view, and dispatch the signal accordingly. Because the view's MoveInputSignal has a parameter, we use a method with a Vector3 parameter as a listener. There, we dispatch the moveInputSignal that will be injected by the GameContext so we can add a command as a response.

Injecting Signal Parameters

We'll need a command to respond to the input signal. Create a new script file called "MoveCommand.cs" and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class MoveCommand : Command {
5
6       [InjectParameter] Vector3 inputDirection;
7
8       protected override void Execute() {
9           Debug.Log("Move command in direction: " + inputDirection);
10      }
11
12  }
```

Woah, that's a new one. We use the [InjectParameter] to inject a parameter of the signal that will trigger this command. In this case, the inputDirection will be set to the Vector3 parameter of the MoveInputSignal. We can then use this value in our command.

Let's bind the MoveCommand as a response to the InputMoveSignal in the context. Open up the GameContext and add the following code.

```
using IoCPlus;

public class GameContext : Context {

    protected override void SetBindings() {
        base.SetBindings();

        Bind<string>("Welcome! Look at me, I'm injecting a value!");
        Bind<JumpInputSignal>();
        Bind<MoveInputSignal>();
        Bind<JumpSignal>();

        BindMediator<PlayerInputMediator, PlayerInputView>();
        BindMediator<PlayerMediator, PlayerView>();

        On<EnterContextSignal>().Do<WelcomeCommand>()
                                .Do<InstantiatePlayerInputCommand>()
                                .Do<InstantiatePlayerCommand>();

        On<JumpInputSignal>().Do<JumpCommand>();

        On<MoveInputSignal>().Do<MoveCommand>();
    }

}
```

By binding the MoveCommand as a response to the MoveInputSignal the Vector3 parameter of the signal will automatically be injected in the command.

Go ahead and hit play. The debugger should now log the correct message when pressing down the left or right mouse button.

Signal Parameter from Command to Mediator

Now that the parameter is successfully injected into our MoveCommand, we can dispatch a MoveSignal to actually move the player. Create a new script file called “MoveSignal.cs” and add the following code.

```
using IoCPlus;
using UnityEngine;

public class MoveSignal : Signal<Vector3> { }
```

Let’s bind this MoveSignal injection in our context. Open up the GameContext and add the following code.

```
using IoCPlus;

public class GameContext : Context {

    protected override void SetBindings() {
```

```
        base.SetBindings();

        Bind<string>("Welcome! Look at me, I'm injecting a value!");
        Bind<JumpInputSignal>();
        Bind<MoveInputSignal>();
        Bind<JumpSignal>();
        Bind<MoveSignal>();

        BindMediator<PlayerInputMediator, PlayerInputView>();
        BindMediator<PlayerMediator, PlayerView>();

        On<EnterContextSignal>().Do<WelcomeCommand>()
                                .Do<InstantiatePlayerInputCommand>()
                                .Do<InstantiatePlayerCommand>();

        On<JumpInputSignal>().Do<JumpCommand>();

        On<MoveInputSignal>().Do<MoveCommand>();
    }

}
```

Let's dispatch the signal in the MoveCommand. Open up the MoveCommand and add the following code.

```
using IoCPlus;
using UnityEngine;

public class MoveCommand : Command {

    [InjectParameter] Vector3 inputDirection;

    [Inject] MoveSignal moveSignal;

    protected override void Execute() {
        moveSignal.Dispatch(inputDirection);
    }

}
```

The MoveCommand uses the same technique to include the signal parameter as the views and mediators do; by adding it as a parameter in the signal's Dispatch() method.

Before listening to the MoveSignal in the PlayerMediator, let's first add the functionality to move to the PlayerView. Open up the PlayerView and add the following code.

```
using IoCPlus;
using UnityEngine;

public class PlayerView : View {

    Rigidbody myRigidbody;
```

```
7
8       public void Jump() {
9           myRigidbody.AddForce(Vector3.up * 300.0f);
10      }
11
12      public void Move(Vector3 direction) {
13          myRigidbody.AddForce(direction * 100.0f);
14      }
15
16      private void Awake() {
17          myRigidbody = GetComponent<Rigidbody>();
18      }
19
20  }
```

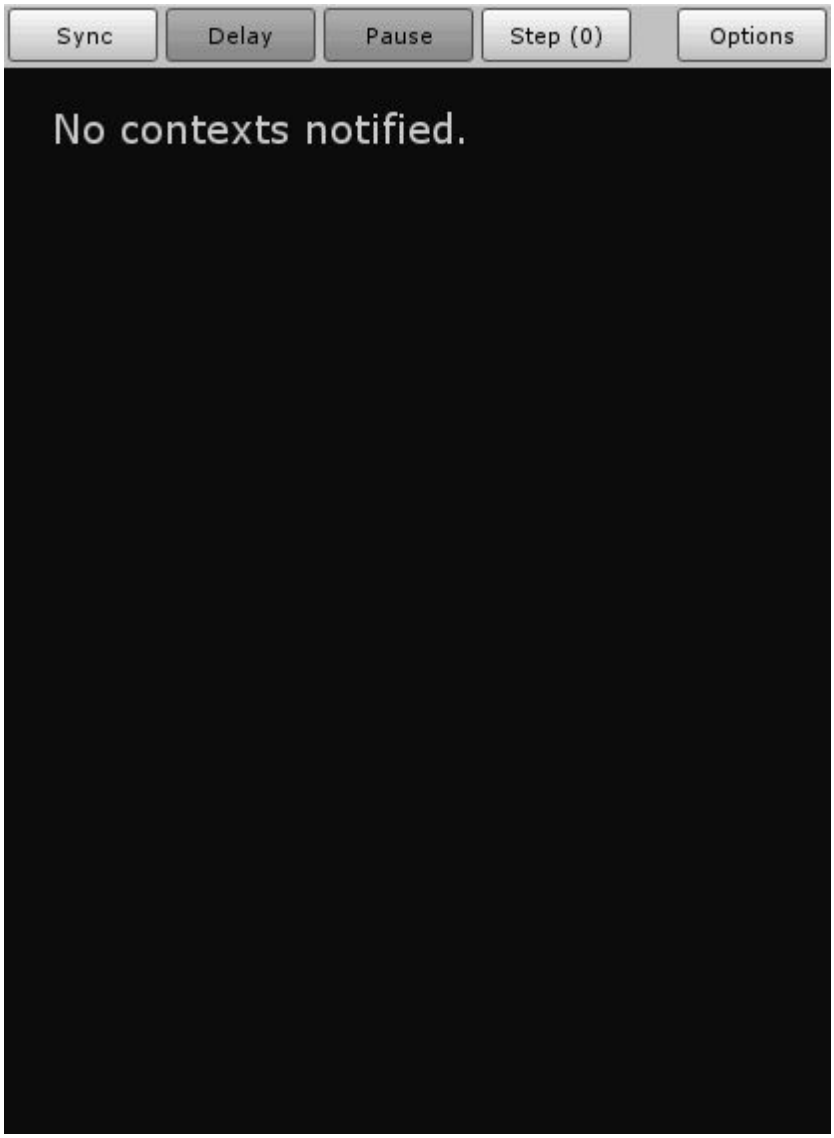
We've added the Move(Vector3) method to add force to the player's rigidbody in the given direction. Now, open up the PlayerMediator and add the following code.

```
1   using IoCPlus;
2   using UnityEngine;
3
4   public class PlayerMediator : Mediator<PlayerView> {
5
6       [Inject] JumpSignal jumpSignal;
7       [Inject] MoveSignal moveSignal;
8
9       public override void Initialize() {
10          jumpSignal.AddListener(OnJumpSignal);
11          moveSignal.AddListener(OnMoveSignal);
12      }
13
14      public override void Dispose() {
15          jumpSignal.RemoveListener(OnJumpSignal);
16          moveSignal.RemoveListener(OnMoveSignal);
17      }
18
19      private void OnJumpSignal() {
20          view.Jump();
21      }
22
23      private void OnMoveSignal(Vector3 direction) {
24          view.Move(direction);
25      }
26
27  }
```

There! The PlayerMediator now injects the MoveSignal, listens to it and moves its view based on the given direction! All dots are now connected. Go ahead and hit play! Once we hit the left or right arrow key the player will start to move in that direction.

The IoC+ Monitor

Go ahead and open up the IoC+ Monitor, you'll see the MoveInputSignal, MoveSignal and MoveCommand light up while it is happening in game.



Note that we can't dispatch signals with parameters from the IoC+ Monitor.

Conclusion

In this tutorial we've seen how to create a signal with parameters. We've seen how to dispatch it from a view and listen to it from the view's mediator. Commands use the [InjectParameter] to inject a signal parameter and can use the value in its execution. We've also seen how to dispatch a signal with a parameter from a command and listen to that signal in a mediator to make its view act accordingly.

Further Reading

This concludes the offline documentation of IoC+. For further reading, please visit our official site http://iocplus.com. The you will find extra tutorials, documentation and more. For support, visit our Unity thread at https://forum.unity3d.com/threads/434417/ where we answer all questions regarding IoC+ for Unity.

Thank you for using IoC+. Please leave a review in the Unity Asset Store and let us hear about what you've accomplished in the Unity thread.