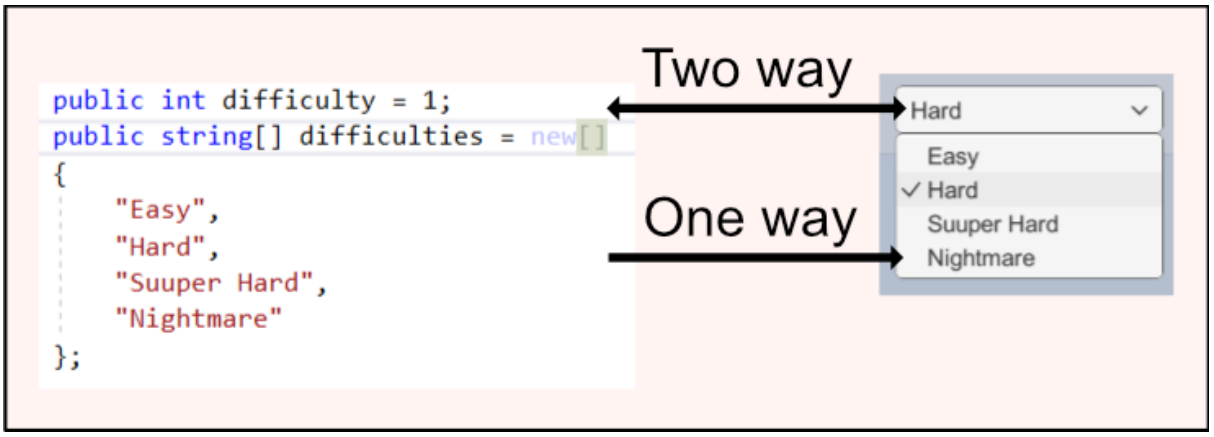
UBindr Overview

An overview on how UBindr can help you create UIs faster. For support contact mattias.fagerlund@carretera.se.

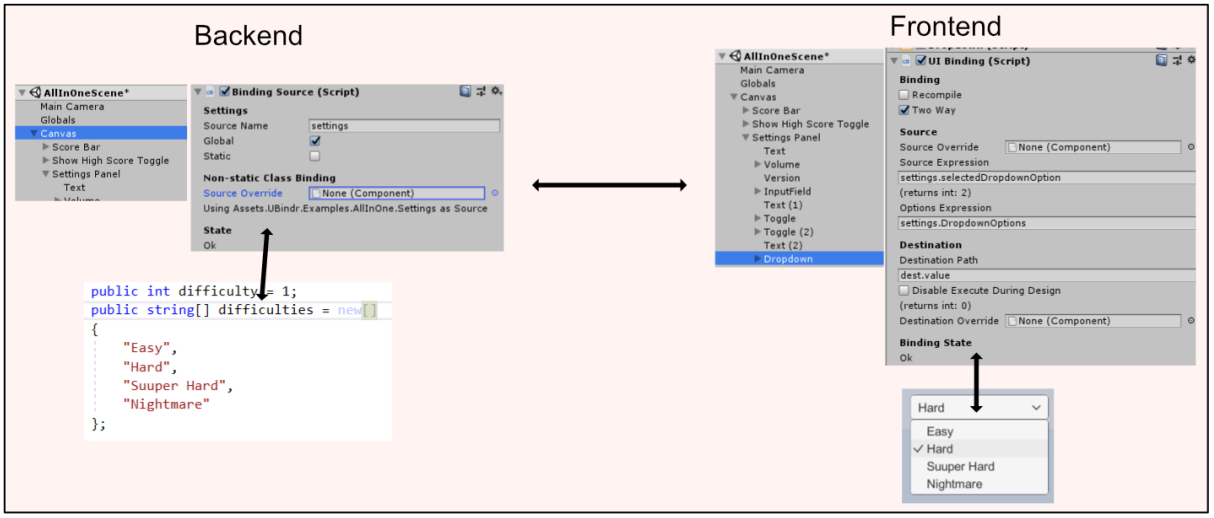


TL;DR

UBindr allows you to publish data from your C# code to the Unity UI with minimal to no code changes. It supports two way binding, so that changes made by the user are passed back to the C# code. It fills some of the same need that Angular, Knockout or React fills for Web applications. It makes building UIs easier and leaves the code cleaner.

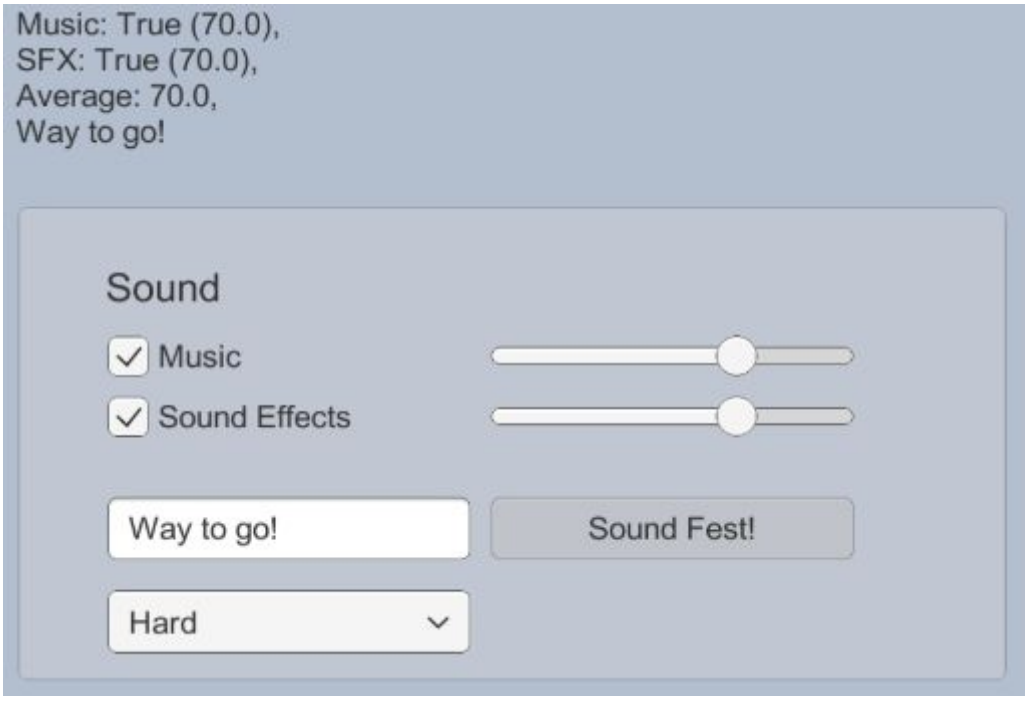
Don't Complicate the Model to Support the UI

UBindr makes the UI design much simpler and it keeps both the view and the model cleaner. The less the model knows about the view, the better.



When using UBindr, you add a Binding Source for any component you want to publish to UI bindings. Bindings can access Binding Sources that are available in it's hierarchy - or that are declared global.

We use a Game Settings scenario as our example. The "Model" in this case is our Settings class. For a typical Unity project, your Settings class will contain knowledge of the UI elements that's used to render it and you need to add code to keep the UI version of a value (Music Volume, for instance) in sync with your Model version. If you use Model-View binding, instead, that's taken care of for you.



The settings model we end up with, at the end of the tutorial series, looks like this;

```
public class TutorialSettings : MonoBehaviour
{
```

```
        public bool soundEffectsEnabled = true;
        [Range(0, 100)]
        public float soundEffectsVolume = 70;

        public bool musicEnabled = true;
        [Range(0, 100)]
        public float musicVolume = 70;

        public string message = "Way to go!";

        public int difficulty = 1;
        public string[] difficulties = new[] { "Easy", "Hard", "Suuper Hard", "Nightmare"
};

        public void SoundFest()
        {
        message = "FIESTA!";
        musicVolume += 1;
        soundEffectsVolume -= 1;
        var diff = difficulties.ToList();
        diff.Add("Festive!");
        difficulties = diff.ToArray();
        }
}
```

You'll notice that it has no knowledge about the UI that presents the information, no component references to Toggles and Inputs. The model only knows about the model. It can change the internal values without a care in the world for who's (if anyone) rendering the information on the screen.

A Highscore List

Another example we explore is a highscore list where the c# code looks like this;

```
public class HighScoreList : MonoBehaviour
{
   public HighScoreList()
   {
      Singleton = this;

      Players = new List<Player>
      {
         new Player{Name="Mr Splendid", Score = 75},
         new Player{Name="Xerxes", Score = 15},
         new Player{Name="Madam GodMode", Score = 100},
      };
```

```
    }

    public static HighScoreList Singleton { get; set; }
    public List<Player> Players { get; set; }
    public Player selectedPlayer;
    public IEnumerable<Player> SortedPlayers { get { return
Players.OrderByDescending(x => x.Score); } }

    public class Player
    {
        public string Name;
        public float Score;
        public Settings Settings;

        public void SelectMe()
        {
            HighScoreList.Singleton.selectedPlayer = this;
        }

        public void DeleteMe()
        {
            HighScoreList.Singleton.Players.Remove(this);
        }

        public void ChangeScore(float delta)
        {
            Score += delta;
            SelectMe();
        }
    }
}
```

Continue

Continue by checking out our tutorials;
Tutorial 1
Tutorial 2
Tutorial 3