Snabb guide till ”Mirror / Unet”

**Documentation:** <https://vis2k.github.io/Mirror/>

**GOOD TO KNOW ABOUT MIRROR!**Mirror builds upon Unet. This means that solutions that works for Unet might work for Mirror too!

**NetworkLobbyManager:**The Network Manager is a component for managing the networking aspects of the game. Will be carried over between scenes.

All prefabs that is going to be spawn and be used in the game needs to be placed in the “Registered Spawnable Prefabs” list. This includes all the playable characters!!! NetworkLobbyManager will be created in the main menu.

**NetworkIdentity:**Gives a game object a unique identity on the network and uses that identity to make the networking system aware of the game object. This script gives you two options for the game object.  
- **Server Only**: Only spawns the game object on the server.  
- **Local Player Authority:** Gives authoritative network control to the client that owns it.  
- **If nothing is checked:** Server will have authority over the object.

Every script that needs to send information to the server needs to be on the same gameObject as the NetworkIdentity script. Also all scripts that wants to communicate with the server needs to inherit from NetworkBehavior.

**NetworkTransform:**The Network Transform component synchronizes the movement and rotation of the game object across the network. Note that NetworkTransform component only synchronize spawned networked game object.

**NetworkAnimator:**The Network Animator component allows you to synchronize animation states for networked objects. It synchronizes state and parameters from an Animator Controller.

You need to define which animator component you want to sync with the network in the inspector.

**Command:**When you call a command function from a client the function will run that function on the server. This is, for example, used to change variables that uses **SyncVar**. Servers will not be able to call a command function.

Because we want out host to be also a player the host need to take/deal damage another way. See example below…

**NOTE!!!** You can only call a Command function if the game object have a network identity attached to it. Otherwise the command attribute will be ignored and the function will be run locally!!!

[VarSync]   
**int** health;

**Void** hit(**int** damage)  
 {  
  
//Check if the host is dealing damage to a game object. If he is, the change the variable directly.  
 **if**(isServer)

{  
 Health -= damage;  
 }

//If a client is dealing damage to a game object. We will need to tell the server to deal the damage to the game object.

**else**  
 {  
 **CmdTakeHealth**(damage)  
 }  
 }

[Command]  
 **void** **CmdTakeHealth**(**int** damage)  
 {  
 *//Apply damage to the GameObject* **TakeDamage**(damage);  
 }

**public** **void** **TakeDamage**(**int** amount)  
 {  
 **if** (!isServer)  
 **return**;  
 health -= amount;  
 }

**SyncVar:**

[SyncVar]  
**public** **int** health;

SyncVars are variables of scripts that inherit from NetworkBehavior which are synchronized from the server to clients.

A client can’t change the value of a SyncVar. If you want to change a variable that is using SyncVar you’ll need to tell the server to change the value (See **Command**)

SyncVar can only be basic types such as int, string, floats but they can also be Unity types such as Vector3 and user defined structs.

**SyncVar Hooks:**

[**SyncVar**(hook = **nameof**(OnChangeHealth))]  
**public** **int** Health

**void** **OnChangeHealth**(**int** health)  
 {  
 healthBar.sizeDelta = **new** **Vector2**(health, healthBar.sizeDelta.y);  
 }

The hook attribute is used to specify a function to be called when the SyncVar variable changes value on the client side.

Note! The value that the variable is going to be change to is sent to the function called as a parameter. The value of the SyncVar is change AFTER the function have been called.

**SyncLists:**

**public** **struct** **Item**{  
 **public** **string** name;  
 **public** **int** amount;  
 **public** Color32 color;  
}  
  
**class** **SyncListItem** : SyncList<Item> {}  
  
**class** **Player** : NetworkBehaviour {  
 SyncListItem inventory;  
 **public** **int** coins = 100;  
  
 [Command]  
 **public** **void** **CmdPurchase**(**string** itemName)  
 {  
 **if** (coins > 10)  
 {  
 coins -= 10;  
 Item item = **new** Item   
 {  
 name = "Sword",  
 amount = 3,  
 color = **new** **Color32**(125,125,125);  
 };  
 *// during next synchronization, all clients will see the item* inventory.**Add**(item)  
 }  
 }  
}

SyncList works like a C# list that also synchronize their contents from the server to the clients.

SyncList can contain: Basic types, Unity types (Vector3), NetworkIdentity, GameObject with a networkIdentity component attached or a structure with any of the above.

The content of the list can’t be changed on the client side. You need to do this on the server side (See **Command**)

**ClientRpc:**

When you call a ClientRpc function on the server side the function will be called on all clients that are connected to the host. A client can’t call this function.