LooseLink

Loose Link is a service locator that was built specifically for Unity to connect low level objects to high level services through abstraction. It’s a powerful tool to keep Your codebase clean, testable, flexible, and easy to modify.

The problem, that we want to solve

Projects very often get more and more rigid over time. The price of making changes in the code and the chance of breaking something with it gets higher and higher through the development.

It’s because the code gradually becomes more and more interconnected or tightly coupled. It means, that most of our classes contains too much information about other classes and the overall architecture of the software.

Ideally every scrip should be designed to, know as little about their surroundings as reasonably possible. But it’s easier said than done. Unity’s built-in solutions are not suited well to connect low level objects to high level services.

This is where **Loose Link** can help, which was designed specifically to complement Unity’s shortcomings. Loose Link can locate and create high level objects based on an easy to setup ruleset separated from the codebase.

 No more need for static classes, singletons, FindobjectOfType.

Our Solution: Inversion of Control

Any object oriented software contains so called higher and low-level objects. Low level objects in a game could be the enemies, projectiles, NPCs, interactable objects. Things that populate the virtual word but without any of them the game would still work. In contrast to that, high level objects representing a whole subsystem in the game, like time management, input system or damage management. This makes up the so-called

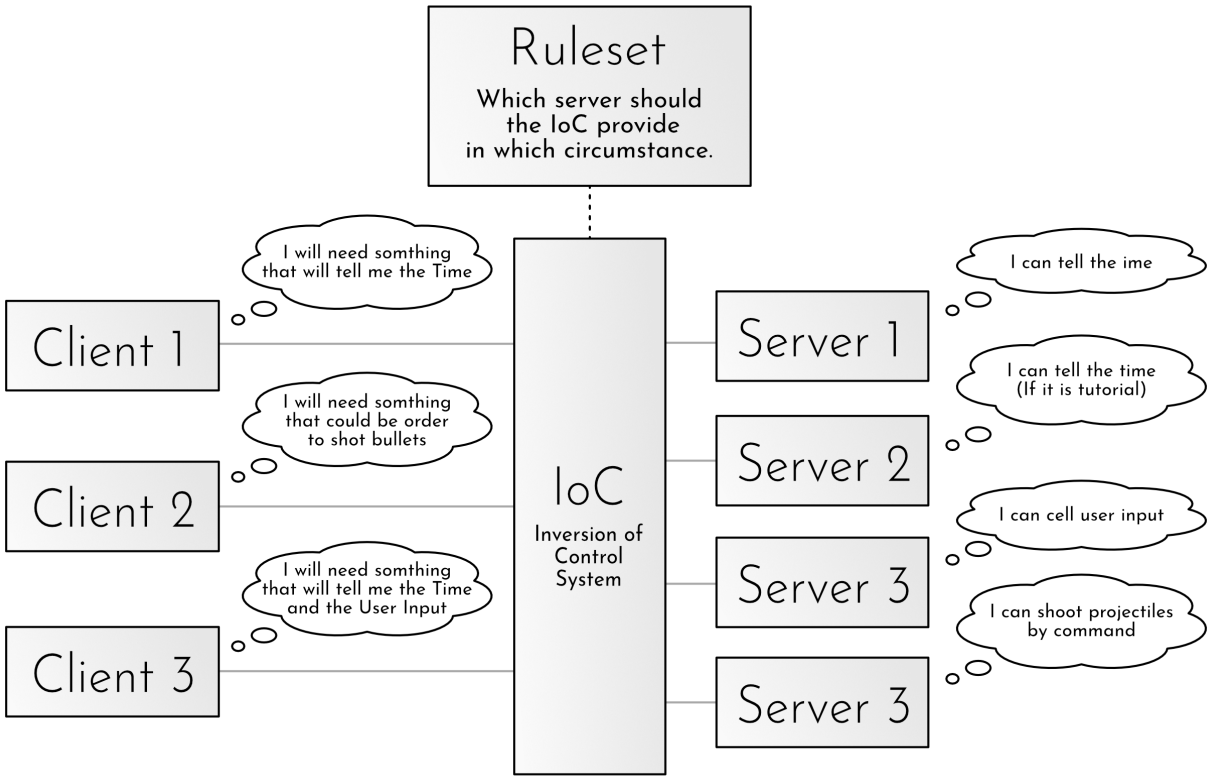
policy layer. Usually, there is only one instance of every of these high-level classes in the game, least one at a time. And usually, a lot of different lower level “smaller” object wants to know and use them.

Let’s call this Manager-like objects Servers (for example: ProgressionManager, DamageManager, SoundManager, WindManager, GameplayManager…) Let’s call the functionalities they provide Services and the lower level objects that would like to use those services Clients.

This is broadly speaking a Client-Server architecture, (which is mostly associated with web development, but we have nothing to do with networks here).

Inversion of Control or IoC in short is a design principle to achieve loose coupling between classes. The idea is to add an new abstraction layer to the code which is responsible for connecting the Clients and the Services they need. This is called an Inversion of Control System.

With an Inversion of Control or shortly IoC system the clients are not directly reach the servers. The clients somehow communicate what kind of service will they need, and the servers communicates what kind of service they provide. Between them is an intermediate layer, the IoC system, that decides which server object will be given to the clients. So, for one service there could be multiple Server object each for different situations, like tests or different runtime platforms. Which one will be provided to the client is based on some logic or ruleset outside of the client or server code.



Unity's built in Inversion of Control solutions

Dependency Injectors and Service locators are different ways of implementing an IoC System. Unity-s [SerializeField] system is a form of Dependency Injection and the GetComponent-like functions are a form of implementing the Service Locator pattern. They are really usefult tools, but their specific implementation makes them not well suited to locate high level services throug abstraction.

Loose Link's Functionality

1. **Clients can request a service including interfaces or abstract types.**
2. **If the requested service doesn't exist yet, it will be instantiated and cached automatically.**
3. **Clear Drag and drop UI in the Editor for setting up, real time monitoring and debugging the service environment.**

4. **Works in editor and play time too.**
5. **Requested objects are initialized before it's provided by the service locator.**
6. **Makes testing and other specific situations easy to implement without unnecessary if-else branches in the game codebase.**
7. **Serving the request are fast, without memory allocation.**

Terminology

Server: High level Unity Object that provides one or more service.

Service: A well defined functionality. In this tool any type can be a service. Specifically a class or an interface.

Client: Object that use one or more services.

Service Locator (SL): System that can instantiate and locate services for the clients.

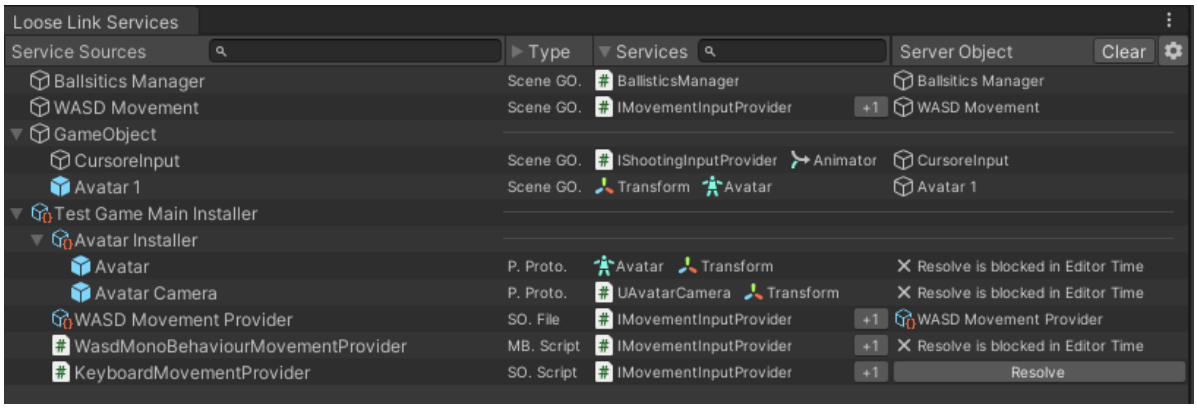
Manual

How to use LooseLink ?

All necessary code is in LooseLink namespace.

▼ **Service Locator window**

Open the Service Locator window: Unity Editor Menu bar / Tools / LooseLink Services



On this window You can see the current Service Environment. It contain a table with 4 columns.

1. **Service Sources column** is a foldable tree view.
   It is the hierarchy of the Service Sources and Service Installers.
   Service source is a rule how to access one or more services.
   To access a service source through the service locator it needed to be installed.
   (The more precise definition of service sources and installers are in the next chapters.)
   The roots of this tree are always installers and the leaf's are Service sources.

2. **Source Type**

   Source type tells You from witch kind server object contains the services and what way will the service located or instantiated. (See the chapter about Service Source Types for more information)

3. **Services**

   The services are the classes and interfaces the sorvice source can provide.

4. **Server** (Always Unity Object)

   The exact server object instance that is available through the source.

   If it is not instantiated yet, there is button to instantiate it.

   (In Editor time the Service Locator wont Instantiate new GameObject in Scene)

▼ **Service Source**

Service Source: Basically a rule how to reach a server object. It contains information about how the server object can be accessed, and what services does it provide.

In Loose Link there is multiple **Service Source Types:**

Every service source needs a source Unity Object, that can determine it's type. The following Unity Object can be a source: GameObject in scene, Prefab file, ScriptableObject file, non abstract MonoBehaviour or ScriptableObject script file.

- **In Scene GameObject**

  The SL. (service locator) gives back the service from one of the component of the the GameObject.

- **Prefab**
  - **Prefab File**: The SL. gives back one component on the original prefab file.
  - **Prefab Prototype**: The SL. instantiates the prefab, and gives back one component on the newly created game object in the scene.

  (If there are multiple source type to an object, you can select from a dropdown menu.)
- **ScriptableObject**
  - **ScriptableObject File**: The SL. gives back the original ScriptableObject file.
  - **Prefab Prototype**: The SL. creates a copy of the ScriptableObject file, then gives the copy to the client.
- **MonoBehaviour script file** (Non abstract only)

  The SL creates an instance of the Component, caches it, then gives it to the client.
- **ScriptableObject script file** (Non abstract only)

  The SL creates an instance of the ScriptableObject, caches it, then gives it to the client.

When a service source Instantiate a new object, it will cache it and in the later the cached objects will be provided.

▼ **Installers & Service Service Sets**

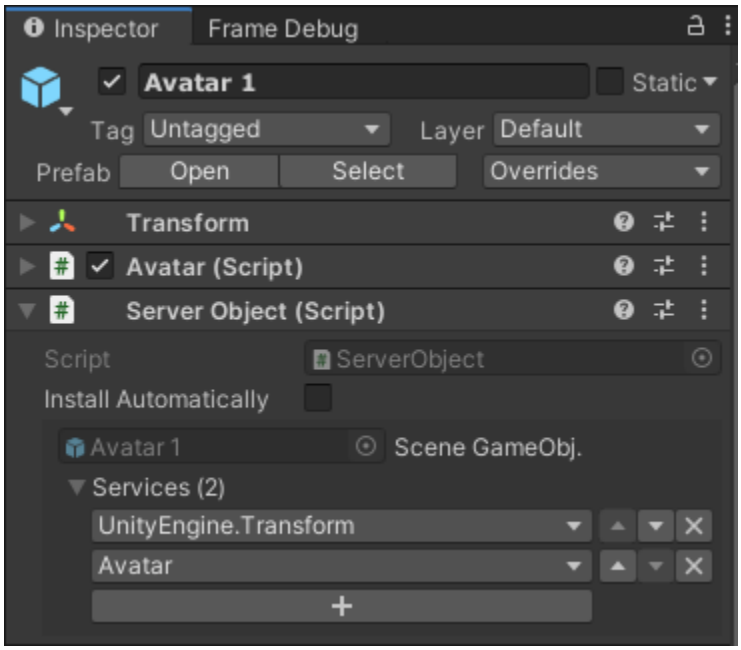
For a Service Source to be available through Service Locator it need to be installed.

To install service sources, we need an **Service Installer** which can containing any number of sources.

You can install Service Sources locally and globally.

Local Service Installer is a component that contains service sources. If this component is in the scene, active and enabled, than their service sources are available through service locator.

You can add **Server Object Component** to an in-scene GameObject or Prefab. On this component You can setup the provided Services.

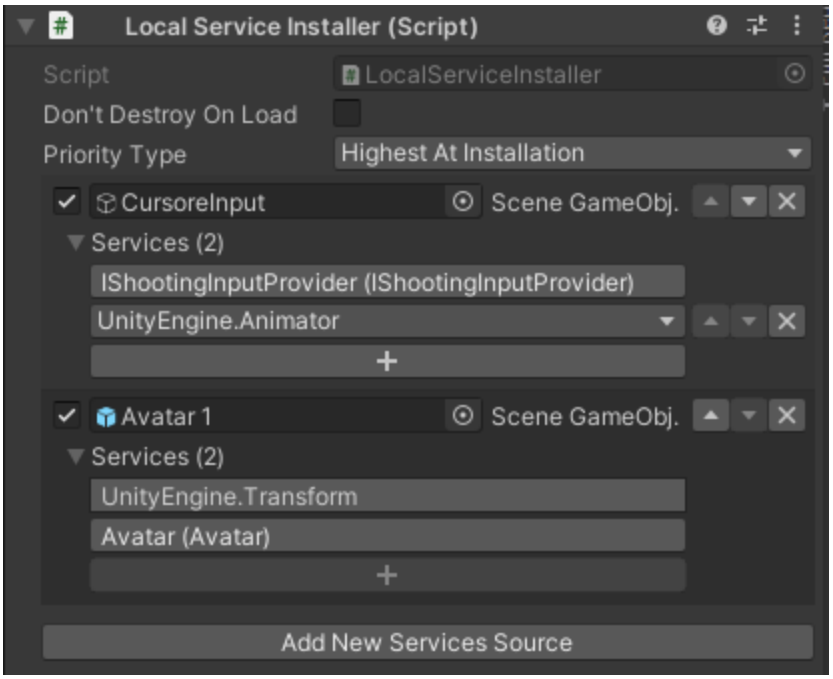


There is a **Install Automatically** checkbox on the Server Object Component. If You enable it, You don't need to use a Local Service Installe**r.**

Service Source Set is a file in the project containing service sources.

Set the **Global Service Installer** checkbox true on a service source set to make all of it's service sources available through service locator anywhere in the project.

Service Environment: All the currently installed service sources in order.



To add new service sources to an Installer or source set, click the "Add New Service Source" button. Set up the source object, set the source type if needed. If there are more source on an installer change their order with the arrow buttons on the UI.

▼ Adding Services to Service Sources

You can add new services with the "+" button. (On Service Source Set-s, Local Service Installer and Server Object Component.) Than you can select any available service on the server object from a dropdown menu.

Server Object: if you add Server Object component to a server GameObject, than you can setup its services on the game object directly but not on the installer. (Like the "Avatar 1" Service source on the illustration.)

There is a checkbox on a Server Object to install itself as a service source automatically. In this case there is no need for a separated local service installer.

You can tag any class or interface with [ServiceType] attribute. If You do so it will be automatically recognised as a Service, no manual setting needed.

▼ How do clients request a services

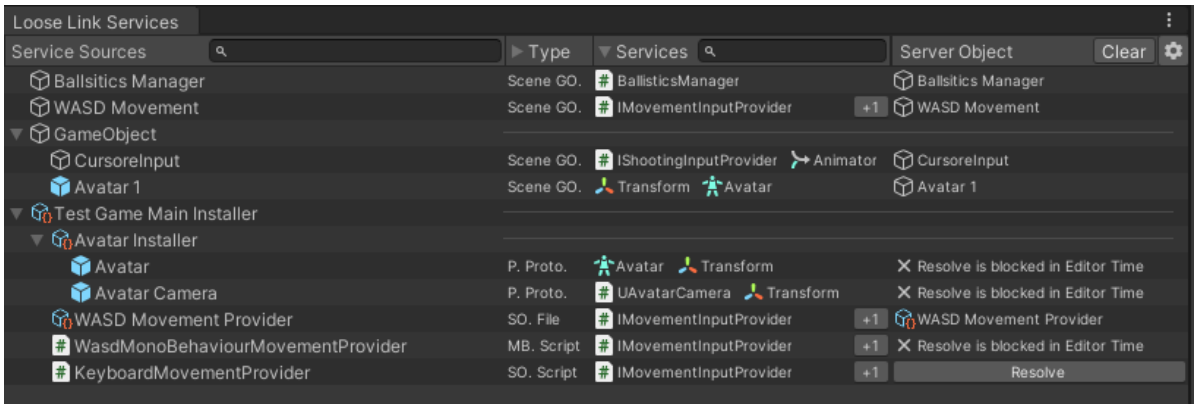
Resolving: The process when the service locator provides the requested service to the client.

Resolving a service is requested by the client with any of these static calls:

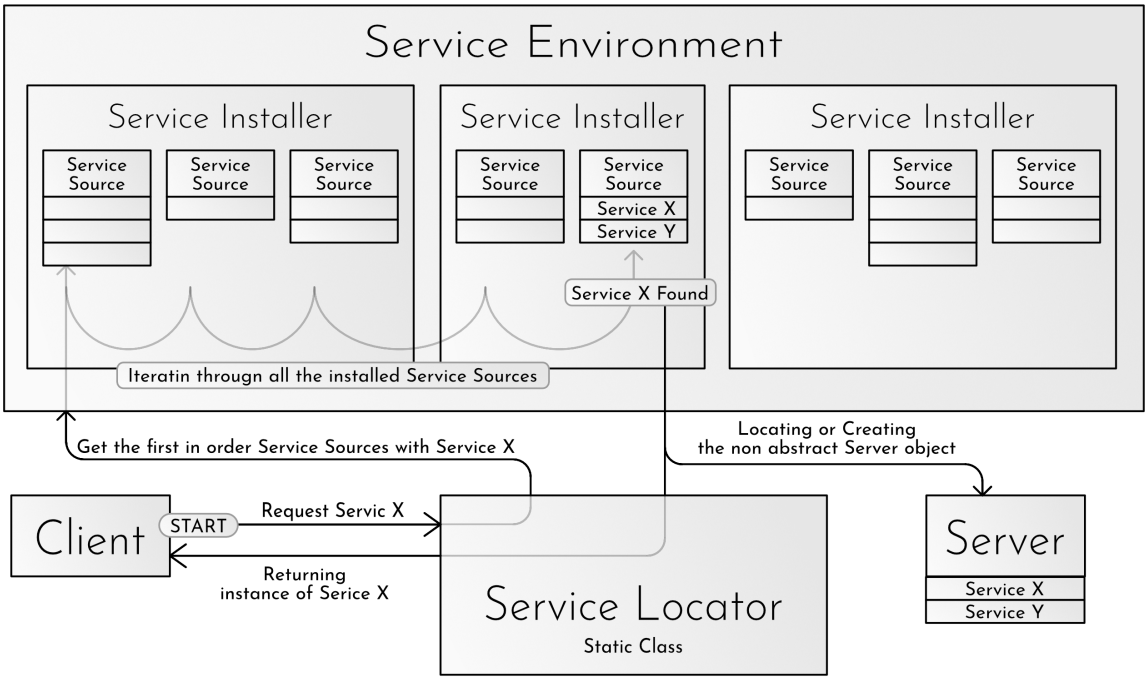
```
Services.Get<TService>();
Services.Get(Type iServiceType);
```

A Service.Get call gives back the first available service.

The list of all the currently available service sources can be checked any time in the Loose Link Services window. The iteration going from top down.



Here is a diagram showing the process of resolving a service:



The order of Service Sources:

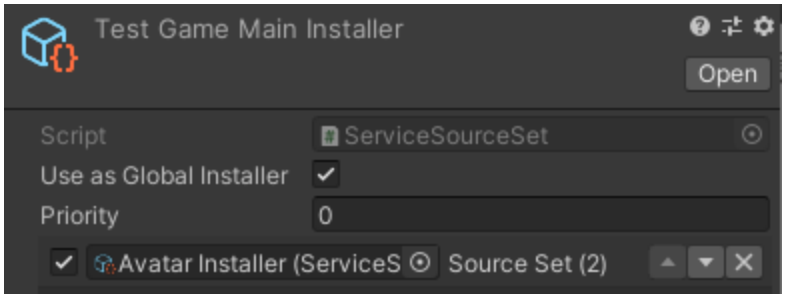
Inside one installer the order of it source can be changed in the inspector.

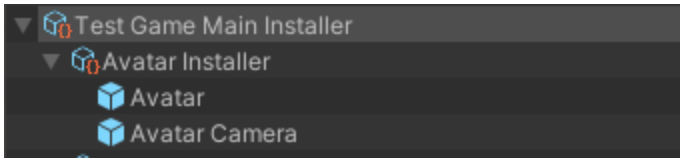
The order of the installers is determined by their priority number. (Higher number gets higher in the list)

If the priority number equals locally installed sources will be higher in the list than globally installed ones.

Local installers can be set up to get the highest existing priority + 1 at the time of installation.

You can add any service source set to an installer or other set like a service source. This way you can build a tree, where the root of the tree is an installer, the nodes in the tree are service source sets, and the leafs are service sources.





▼ Initializing the Server Object

Lot of time the Server object is instantiated through a service source at the time of the first resolve request.

In this case the new object can be in a state when the Server is not properly set up. For example the Awake method going to run next frame after the Service.Get call.

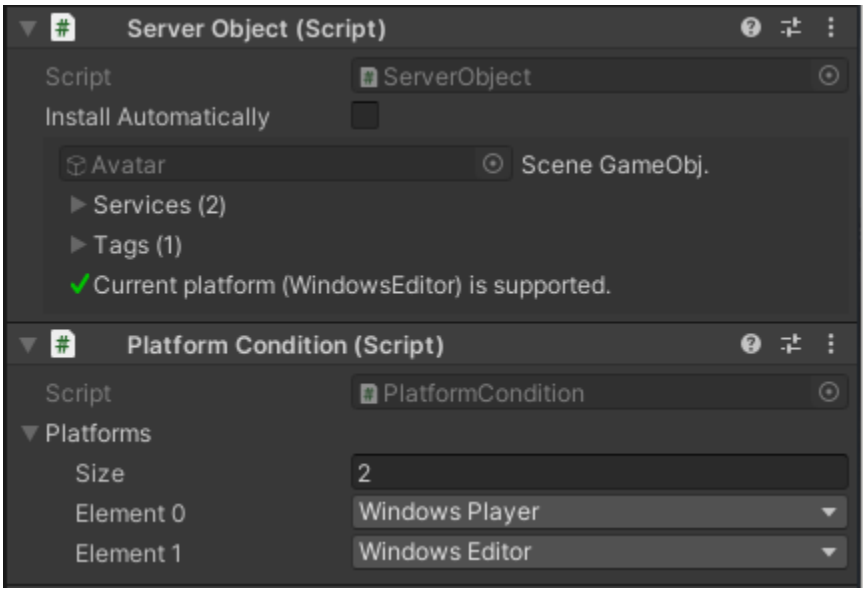
To avoid that use the IInitializable interface on any class on the server object. The IInitializable interface has only one method: Initialize(). This method will be called before the newly instantiated object will be returned to the client.

Server object can be clients of other server objects. In this way there could be circles in the Initialization method. (A → B, B → C, C → A) **This is NOT a problem.** Every initialisation method going to be called only once.

▼ Adding conditions to ServiceSources

Conditions can be added to a service source with IServiceSourceCondition interface.

The interface need to implement a boolean typed method: CanResolve(). If the method returns false, the service source will bi skipped at the process of resolve.



▼ Tagging Service Sources

If there is more than one valid source to one service at the time, you need to differentiate between them somehow. In LooseLink you wave an option to a mark them with one or more tags.

Tag: Any C# object, with which we can mark a service source, making it possible to specify the service searches at the time of resolve with the following getter functions:

```
Services.Get<TService>(params object[] tags);
Services.Get(Type iServiceType, params object[] tags);
```

To use tags you need to enable them in the settings menu (Top right corner of the LooseLink Services Window.)

With the tagging option enabled there will be a new column in the "LooseLink Services" window.

Adding Tags to a Service Source

There is two way to add tags to a service source.

Manually on the installer, or by adding the ITagged interface to an source object.

> ⚠️ **Notice that with tagging, you reintroduce extra information into the client abut the service. It's not necessarily a problem, but keep it in mind, that the reason to use an IoC system in the first place was to avoid that.**

▼ Notifying Clients about the change of the Service Environment

If there is a client that can be present when the environment changes, it might need to update its services. To do that, You can subscribe to the change of the environment by service type.

```
Services.Environment.
    SubscribeToEnvironmentChange<IMovementInputProvider>(OnEnvironmentChanged);

void OnEnvironmentChanged() { /* ... */ }
```

> ⚠️ **This function is only works, when service sources are installed or uninstalled, not when objects change their conditions and tags. For this, it is strongly suggested to not change the conditions and tags of a service source in playtime.**