Simple Dependency Injector for Unity

Introduction

Simple Dependency Injector is a lightweight, efficient, and Unity-focused plugin designed to streamline how you manage shared services throughout your game. Unlike large and complex dependency-injection frameworks, this tool is crafted specifically for Unity’s architecture — respecting object lifecycles, performance constraints, and the unique workflows of game development.

With this plugin, you can register services implemented as ScriptableObjects or MonoBehaviours and retrieve them through a clean and centralized API. The system avoids heavy reflection, unnecessary abstractions, and runtime overhead, delivering a minimal and predictable approach that works seamlessly inside Unity projects of any size.

Key features include:

- Automatic and safe service registration during initialization
- Fast, type-based generic retrieval methods
- Separate support for ScriptableObject-based and MonoBehaviour-based services
- Built-in protection against accidental type duplication
- Optional lazy service access for improved performance
- A minimal and intuitive API designed for clarity and maintainability

The Simple Dependency Injector was created with one goal in mind: to help you keep your project clean, decoupled, testable, and easy to maintain — without adding unnecessary complexity.

If you want a dependency-injection solution that is simple, efficient, and truly built for Unity, you’ve just found it.

How to Install

In your Unity Project, open the Package Manager window, click on + Add git repository and put the follow url to use the current package:

- **https://github.com/icaro56/simple-dependency-injector.git#upm**

Case you want to use a specific version, use the follow url changing the version:

- **https://github.com/icaro56/simple-dependency-injector.git#upm/v1.0.0**

How to work Simple Dependency Injector lib

The Simple Dependency Injector works by registering your services during a central DependencyInjector Setup phase. Any class that implements the IService interface can be added as a service, whether it's a ScriptableObject or MonoBehaviour. During setup, the injector maps each service type to its corresponding implementation.

To use a service inside another class, you simply inject it through an Injected field. This gives you safe, lazy-loaded access to the service without needing to manually initialize it.

In short: **Define a service using IService → Register it in the DependencyInjector → Inject it in your target class and use it immediately.**

DependencyInjector Setup

1. Create a new empty GameObject in your scene. Example name: DependencyInjector.
2. Add the DependencyInjector component to this GameObject.
3. In the Inspector, register your services by assigning implementations to their corresponding interfaces. The injector automatically validates the setup and ensures that each interface has only one implementation.

Once this object is in place, any class using Injected can safely request its service.

Initialization Requirement

Before any class attempts to access an injected service, the injector must be initialized. You have **two options**:

✔ Option 1 — Call Setup() Manually (Recommended)

You can manually trigger initialization from a high-priority script, such as a GameManager, ensuring it runs before any other system tries to resolve services.

```
public class GameManager : MonoBehaviour
{
    [SerializeField] private DependencyInjector _injector;

    private void Awake()
    {
        _injector.Setup(); // Ensures the injector is ready before anything else
    }
}
```

This guarantees full control over startup order and avoids race conditions.

✔ Option 2 — Enable autoInit The DependencyInjector component includes an autoInit toggle. If enabled, the injector will automatically initialize itself during its own Awake() method.

However, **you must ensure its script execution order is set to run before any other script** that relies on services. Unity allows adjusting this via: **Project Settings → Script Execution Order**

⚠ Why Initialization Order Matters If the injector isn't initialized before another script tries to access an injected service:

- Injected.Value may fail
- Services may not be registered yet
- Logic may break during scene startup

Ensuring proper initialization is crucial for predictable, stable behavior.

IService Interfaces

Every service in Simple Dependency Injector must implement the IService interface. You can create a service in two different forms:

ScriptableObject Service

ScriptableObject services are assets stored in your project. To use this approach:

1. Create a new class that inherits from both ScriptableObject and IService.
2. Create an asset instance of this ScriptableObject in your project (via Create → YourService).
3. Assign this asset to the ScriptableObject Services list in the DependencyInjector component.

ScriptableObject services are ideal for global, stateless or configuration-based services.

MonoBehaviour Service

If a service needs to be part of the scene or interact with GameObjects:

1. Create a class that inherits from MonoBehaviour and IService.
2. Add a GameObject to the scene and attach this service script to it.
3. Assign this scene object to the MonoBehaviour Services list in the DependencyInjector component.

MonoBehaviour services are suited for services that require lifecycle events or scene-bound behavior.

Registering Services in the DependencyInjector

After creating your service class (ScriptableObject or MonoBehaviour):

- Open the DependencyInjector GameObject in the scene.
- Add your service to the appropriate list:
  - ScriptableObject Services
  - MonoBehaviour Services

The injector will automatically validate and make your services discoverable by all components using Injected.

Inject on Target Class

To use a service inside any class (MonoBehaviour, ScriptableObject, or plain C# class), you only need a single member:

`private Injected<IUserService> _userService;`

The Injected wrapper automatically:

- Lazily resolves the service when first accessed
- Ensures the service is registered and valid
- Removes the need for manual initialization, reflection, or Awake() boilerplate

Accessing the Service

Whenever you need the service, simply access .Value: `_userService.Value.DoSomething();` You can safely use it anywhere — Start, Update, constructors of plain classes — without worrying about initialization order. If the service was registered in the DependencyInjector, it will always be ready the moment you access .Value.

```
public class UserLabelBehavior : MonoBehaviour
{
    [SerializeField]
    private TextMeshProUGUI _userLabel;

    private Injected<IUserService> _userService;

    private void Start()
    {
        UpdateView();
    }

    private void UpdateView()
    {
        _userLabel.text = _userService.Value.GetName();
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            _userService.Value.SetName("New User Name");
            UpdateView();
        }
    }
}
```

Tests

Testing with Simple Dependency Injector is straightforward. Because injected services use the Injected wrapper, you can easily assign mock or fake implementations during unit tests without requiring a scene or the actual DependencyInjector component.

To keep things simple, this repository already includes a complete example demonstrating how to test a class that uses injected services. Example Test

Examples

In the Simple Atom project, there are examples in the scene folder:

- Example 1 - How to use