Service Locator by Fidgetland

The Service Locator pattern is a design pattern used in software development to decouple the service consumers (clients) from the concrete implementations of the services they use. It provides a central registry or "locator" that clients can query to obtain instances of the services they need.

Key Concepts

1. **Service Interface**: Defines the contract that a service must adhere to.
2. **Service Implementation**: The concrete class that implements the service interface.
3. **Service Locator**: A central registry that holds the mappings between service interfaces and their corresponding implementations.
4. **Client**: The consumer of the service, which requests the service from the service locator.

How It Works

1. **Registration**: Services are registered with the service locator. This can be done at application startup or dynamically during runtime.
2. **Lookup**: Clients request services from the service locator by specifying the interface or type of service they need.
3. **Return**: The service locator returns an instance of the requested service to the client.

Advantages

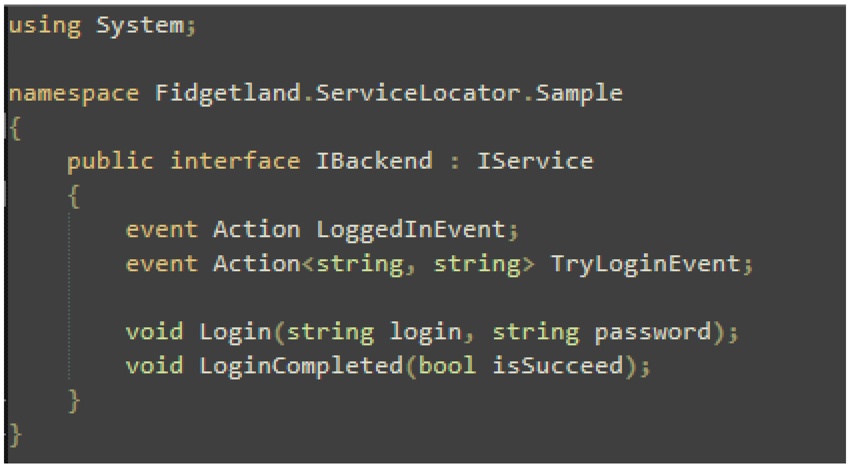
- **Decoupling**: Clients are decoupled from the concrete implementations of services, which makes the system more modular and easier to maintain.
- **Flexibility**: It allows for easy substitution of different service implementations, which can be useful for testing or switching between different configurations.
- **Centralized Control**: Service registration and lookup are centralized, making it easier to manage dependencies and control the lifecycle of services.

Disadvantages

- **Hidden Dependencies**: Dependencies can become less obvious because they are not explicitly defined in the client code, making the system harder to understand and maintain.
- **Global State**: The service locator often relies on a global state, which can lead to issues with concurrency and testability.
- **Difficulty in Testing**: It can be harder to mock or stub services during unit testing because the service locator is a central point of service retrieval.

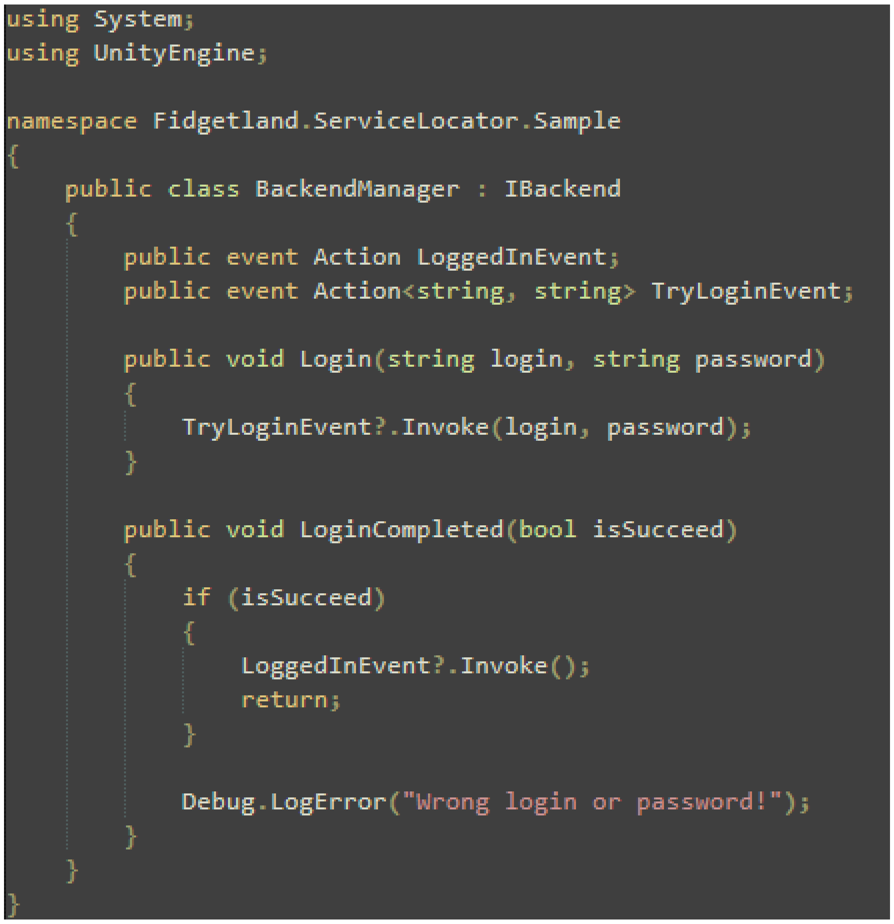
Example

- **IBackend** – interface defines a backend contract.



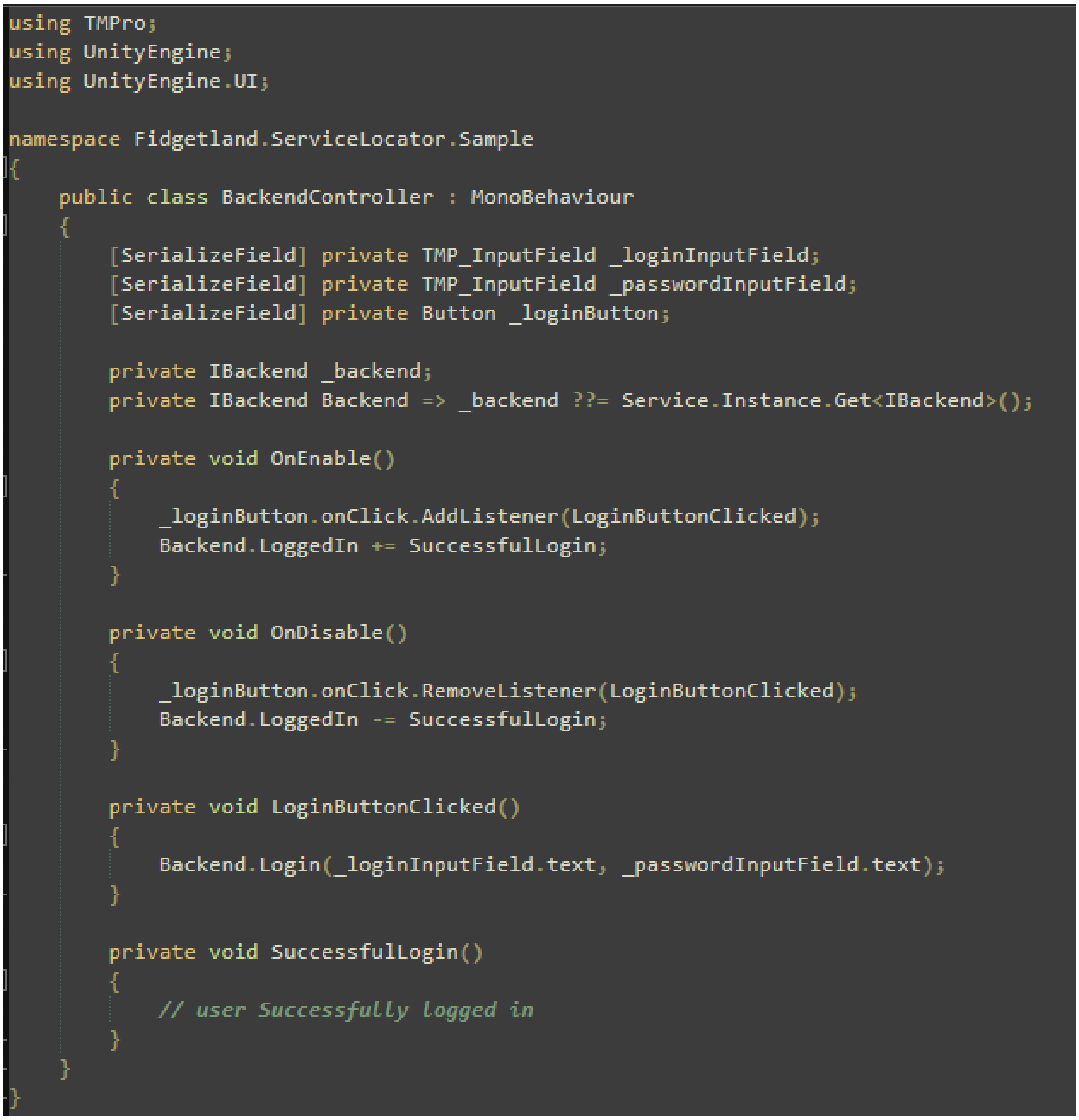
```
using System;

namespace Fidgetland.ServiceLocator.Sample
{
    public interface IBackend : IService
    {
        event Action LoggedInEvent;
        event Action<string, string> TryLoginEvent;

        void Login(string login, string password);
        void LoginCompleted(bool isSucceed);
    }
}
```

- BackendManager – interface realization.



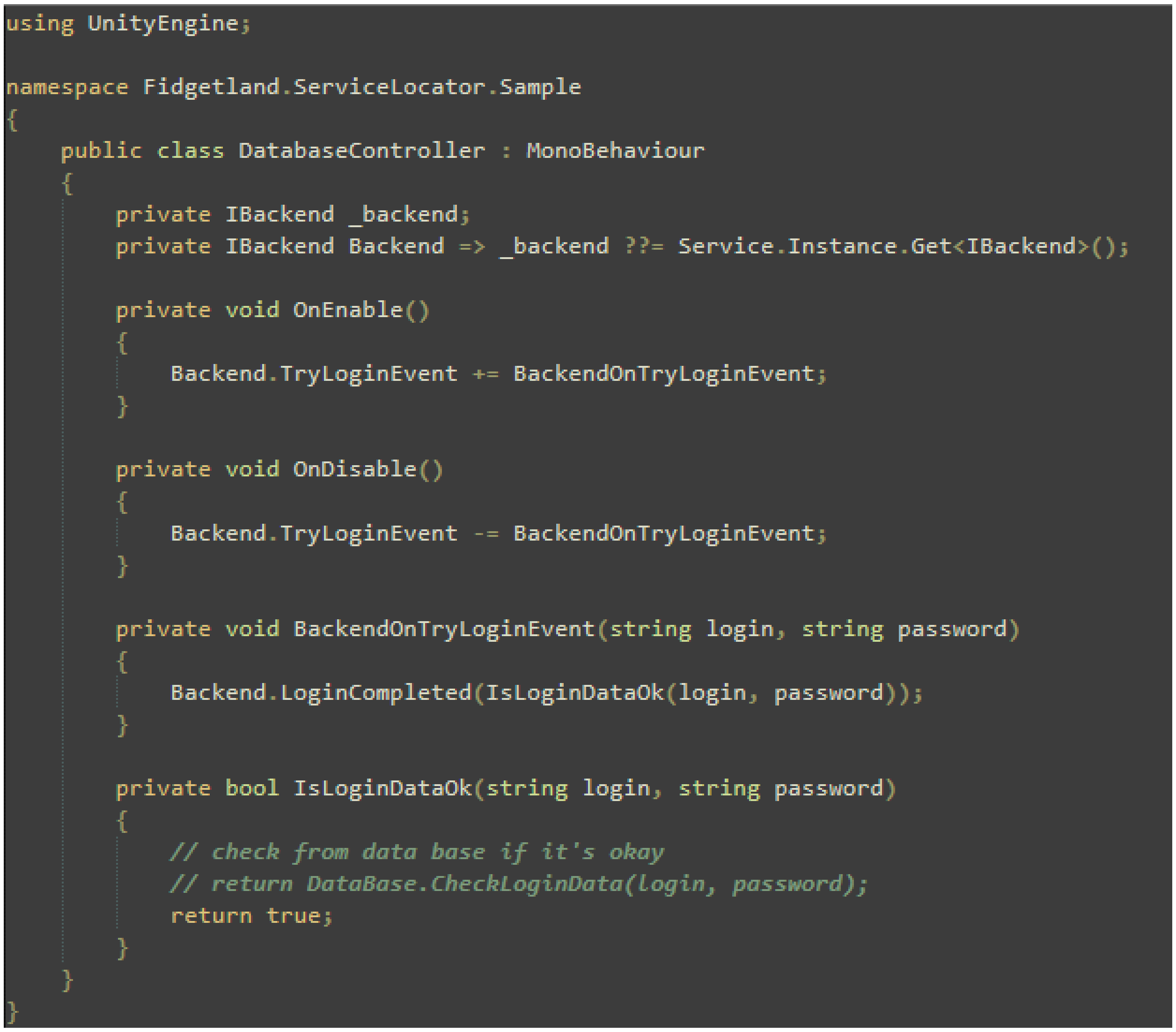
```
using System;
using UnityEngine;

namespace Fidgetland.ServiceLocator.Sample
{
    public class BackendManager : IBackend
    {
        public event Action LoggedInEvent;
        public event Action<string, string> TryLoginEvent;

        public void Login(string login, string password)
        {
            TryLoginEvent?.Invoke(login, password);
        }

        public void LoginCompleted(bool isSucceed)
        {
            if (isSucceed)
            {
                LoggedInEvent?.Invoke();
                return;
            }

            Debug.LogError("Wrong login or password!");
        }
    }
}
```

- **BackendController** – the UI handler.



```
using TMPro;
using UnityEngine;
using UnityEngine.UI;

namespace Fidgetland.ServiceLocator.Sample
{
    public class BackendController : MonoBehaviour
    {
        [SerializeField] private TMP_InputField _loginInputField;
        [SerializeField] private TMP_InputField _passwordInputField;
        [SerializeField] private Button _loginButton;

        private IBackend _backend;
        private IBackend Backend => _backend ??= Service.Instance.Get<IBackend>();

        private void OnEnable()
        {
            _loginButton.onClick.AddListener(LoginButtonClicked);
            Backend.LoggedIn += SuccessfulLogin;
        }

        private void OnDisable()
        {
            _loginButton.onClick.RemoveListener(LoginButtonClicked);
            Backend.LoggedIn -= SuccessfulLogin;
        }

        private void LoginButtonClicked()
        {
            Backend.Login(_loginInputField.text, _passwordInputField.text);
        }

        private void SuccessfulLogin()
        {
            // user Successfully logged in
        }
    }
}
```

- **DatabaseController –** the controller of the database.



```csharp
using UnityEngine;

namespace Fidgetland.ServiceLocator.Sample
{
    public class DatabaseController : MonoBehaviour
    {
        private IBackend _backend;
        private IBackend Backend => _backend ??= Service.Instance.Get<IBackend>();

        private void OnEnable()
        {
            Backend.TryLoginEvent += BackendOnTryLoginEvent;
        }

        private void OnDisable()
        {
            Backend.TryLoginEvent -= BackendOnTryLoginEvent;
        }

        private void BackendOnTryLoginEvent(string login, string password)
        {
            Backend.LoginCompleted(IsLoginDataOk(login, password));
        }

        private bool IsLoginDataOk(string login, string password)
        {
            // check from data base if it's okay
            // return DataBase.CheckLoginData(login, password);
            return true;
        }
    }
}
```