Injection de Dépendances en .NET

Contexte

L'injection de dépendances (Dependency Injection, DI) est un principe fondamental en programmation orientée objet, surtout dans le développement d'applications évolutives et maintenables. Elle permet de déléguer la responsabilité de la création des instances des classes à un conteneur d'injection de dépendances, ce qui facilite le test unitaire et réduit le couplage entre les composants.

Dans ce contexte, nous allons démontrer l'utilisation de l'injection de dépendances dans deux environnements :

1. **.NET Framework avec AutoFac**
2. **.NET Core avec le conteneur d'injection de dépendances intégré**

Partie 1 : Injection de Dépendances en .NET Framework avec AutoFac

Installation

1. Créez un projet de console en .NET Framework.
2. Ajoutez le package NuGet AutoFac :

```
Install-Package Autofac
Install-Package Autofac.Extensions.DependencyInjection
```

Exemple de Code

```
using System;
using Autofac;

namespace GarageManagement
{
    public interface IVehiculeRepository
    {
        void AddVehicule(string Vehicule);
    }

    public class SQLVehiculeRepository : IVehiculeRepository
    {
        public void AddVehicule(string Vehicule)
        {
            Console.WriteLine($"Vehicule added: {Vehicule}");
```

```
        }
    }

    public interface IVehiculeService
    {
        void RegisterVehicule(string Vehicule);
    }

    public class VehiculeService : IVehiculeService
    {
        private readonly IVehiculeRepository _VehiculeRepository;

        public VehiculeService(IVehiculeRepository VehiculeRepository)
        {
            _VehiculeRepository = VehiculeRepository;
        }

        public void RegisterVehicule(string Vehicule)
        {
            _VehiculeRepository.AddVehicule(Vehicule);
            Console.WriteLine($"Vehicule registered: {Vehicule}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var builder = new ContainerBuilder();
            builder.RegisterType<SQLVehiculeRepository>().As<IVehiculeRepository>();
            builder.RegisterType<VehiculeService>().As<IVehiculeService>();
            var container = builder.Build();

            var VehiculeService = container.Resolve<IVehiculeService>();
            VehiculeService.RegisterVehicule("Toyota Corolla");

            Console.ReadLine();
        }
    }
}
```

Explications

1. **Interface `IVehiculeRepository` et implémentation `SQLVehiculeRepository`** : Cette interface représente le dépôt de véhicules, et `SQLVehiculeRepository` en est une implémentation concrète.
2. **Interface `IVehiculeService` et implémentation `VehiculeService`** : `VehiculeService` utilise `IVehiculeRepository` pour gérer les véhicules.

3. **Configuration du conteneur AutoFac** : `ContainerBuilder` est utilisé pour enregistrer les types. Ici, `SQLVehiculeRepository` est enregistré comme implémentation de `IVehiculeRepository`, et `VehiculeService` est enregistré comme implémentation de `IVehiculeService`.
4. **Résolution de la dépendance** : `Resolve<IVehiculeService>()` permet de récupérer l'instance de `VehiculeService` via le conteneur AutoFac.
5. **Veuilliez noter que la résolution se fait en cascade.**

Partie 2 : Injection de Dépendances en .NET Core

Installation

1. Créez un projet de console en .NET Core.
2. Utilisez le conteneur d'injection de dépendances natif.

Exemple de Code

```
using System;
using Microsoft.Extensions.DependencyInjection;

namespace GarageManagementWithDotNetCore
{
    public interface IVehiculeRepository
    {
        void AddVehicule(string Vehicule);
    }

    public class SQLVehiculeRepository : IVehiculeRepository
    {
        public void AddVehicule(string Vehicule)
        {
            Console.WriteLine($"Vehicule added: {Vehicule}");
        }
    }

    public interface IVehiculeService
    {
        void RegisterVehicule(string Vehicule);
    }

    public class VehiculeService : IVehiculeService
    {
        private readonly IVehiculeRepository _VehiculeRepository;

        public VehiculeService(IVehiculeRepository VehiculeRepository)
```

```
        {
            _VehiculeRepository = VehiculeRepository;
        }

        public void RegisterVehicule(string Vehicule)
        {
            _VehiculeRepository.AddVehicule(Vehicule);
            Console.WriteLine($"Vehicule registered: {Vehicule}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var serviceProvider = new ServiceCollection()
                .AddSingleton<IVehiculeRepository, SQLVehiculeRepository>()
                .AddSingleton<IVehiculeService, VehiculeService>()
                .BuildServiceProvider();

            var VehiculeService = serviceProvider.GetService<IVehiculeService>();
            VehiculeService.RegisterVehicule("Toyota Corolla");

            Console.ReadLine();
        }
    }
}
```

Explications

1. **Interface `IVehiculeRepository` et implémentation `SQLVehiculeRepository`** : Identiques à celles du premier exemple.
2. **Interface `IVehiculeService` et implémentation `VehiculeService`** : `VehiculeService` utilise `IVehiculeRepository` pour gérer les véhicules.
3. **Configuration du conteneur d'injection de dépendances natif** : `ServiceCollection` est utilisé pour enregistrer les types. `AddSingleton<IVehiculeRepository, VehiculeRepository>()` et `AddSingleton<IVehiculeService, VehiculeService>()` enregistrent les services respectifs.
4. **Résolution de la dépendance** : `GetService<IVehiculeService>()` permet de récupérer l'instance de `VehiculeService` via le conteneur d'injection de dépendances natif.

Exercices

1. **Exercice 1** : Ajoutez un nouveau service `MaintenanceService` implémentant une interface `IMaintenanceService` pour gérer l'entretien des véhicules. Modifiez la

configuration du conteneur pour enregistrer et utiliser ce service.

2. **Exercice 2** : Créez une interface `INotificationService` avec une méthode `Notify(string notification)` et implémentez-la dans une classe `NotificationService` qui utilise `IVehiculeService` pour envoyer une notification après l'enregistrement d'un véhicule. Enregistrez les services et résolvez `INotificationService` pour envoyer une notification.
3. **Exercice 3** : Implémentez l'injection de dépendances avec un cycle de vie transitoire (`Transient`) pour un service `LoggingService` et démontrez son utilisation en affichant un message de log à chaque appel.
4. **Exercice 4** : Intégrez l'injection des dépendances à votre projet fil rouge.

En complétant ces exercices, vous renforcerez votre compréhension de l'injection de dépendances et de la configuration des conteneurs en .NET Framework et .NET Core.

Stratégies d'Enregistrement des Dépendances

Lors de l'utilisation de l'injection de dépendances (DI) dans .NET, il est essentiel de comprendre les différentes stratégies d'enregistrement des services. Chaque stratégie détermine la durée de vie des instances de service. Voici les principales stratégies utilisées :

1. **Singleton**
2. **Transient**
3. **Scoped**

Source: MSDN

1. Singleton

Description
Un service enregistré en tant que singleton est créé une seule fois et partagé tout au long du cycle de vie de l'application. La même instance est utilisée chaque fois que le service est requis.

Utilisation
La stratégie singleton est idéale pour les services qui doivent conserver l'état tout au long de l'application, comme les services de configuration ou de cache.

2. Transient

Description
Un service enregistré en tant que transient est créé chaque fois qu'il est demandé. Cela signifie qu'une nouvelle instance est fournie à chaque fois que le service est injecté.

Utilisation

La stratégie transient est utile pour les services stateless ou lorsque chaque instance doit être unique et sans état persistant.

3. Scoped

Description
Un service enregistré en tant que scoped est créé une fois par portée (scope). En applications web, une nouvelle instance est créée pour chaque (requête HTTP/Scope) et réutilisée dans le cadre de cette requête.

Utilisation
La stratégie scoped est souvent utilisée dans les applications web pour des services qui doivent être créés une fois par (requête HTTP/Scope), comme les services de gestion des bases de données ou des unités de travail.

Comparaison des Stratégies

Stratégie	Durée de vie	Quand l'utiliser
Singleton	Tout au long de l'application	Pour des services avec état persistant, configuration, cache
Transient	Par demande/ instance	Pour des services stateless, instances courtes
Scoped	Par portée (scope)	Pour des services liés au cycle de vie de la requête (requête HTTP/Scope)

Exemples Pratiques

Exemple 1 : Enregistrement de Services en .NET Core

```
public void ConfigureServices(IServiceCollection services)
 {
     services.AddSingleton<ISingletonService, SingletonService>();
     services.AddTransient<ITransientService, TransientService>();
     services.AddScoped<IScopedService, ScopedService>();
 }
```

Exemple 2 : Enregistrement de Services avec AutoFac

```
var builder = new ContainerBuilder();

 // Singleton
 builder.RegisterType<SingletonService>().As<ISingletonService>().SingleInstance();
```

```
// Transient
builder.RegisterType<TransientService>().As<ITransientService>().InstancePerDepende

// Scoped
builder.RegisterType<ScopedService>().As<IScopedService>().InstancePerLifetimeScope

var container = builder.Build();
```

Conclusion

La compréhension et l'utilisation appropriée des stratégies d'enregistrement des dépendances permettent de créer des applications plus performantes, maintenables et évolutives. En choisissant correctement la durée de vie des services, vous pouvez contrôler la gestion de la mémoire, l'accès aux ressources partagées, et garantir que vos services sont utilisés de manière optimale tout au long du cycle de vie de votre application.