Développement d'applications .NET

C# Avancé : Délégués, Événements, Génériques, Async/Await, et Injection de Dépendance

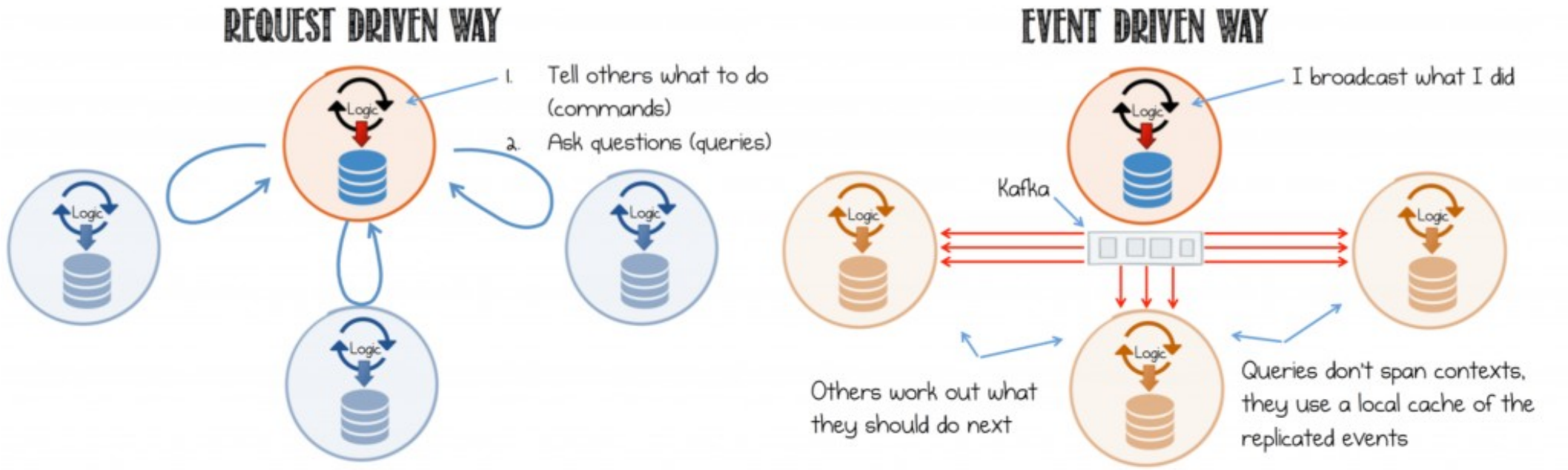
Introduction à la programmation orientée événements (Event-Driven Architecture - EDA)

Pourquoi changer de paradigme ?

Les architectures classiques (REST, procédurales) :

- Dépendent d'un ordre strict d'exécution.
- Couplent fortement les composants.
- Ont du mal à réagir aux changements en temps réel.

L'approche événementielle repose sur un concept simple : "Les composants ne s'appellent pas — ils se notifient."



Qu’est-ce qu’un événement ?

Un événement est une information décrivant un changement d’état.

Exemples:

- Un utilisateur clique sur un bouton
- Une commande est validée
- Un capteur envoie une mesure

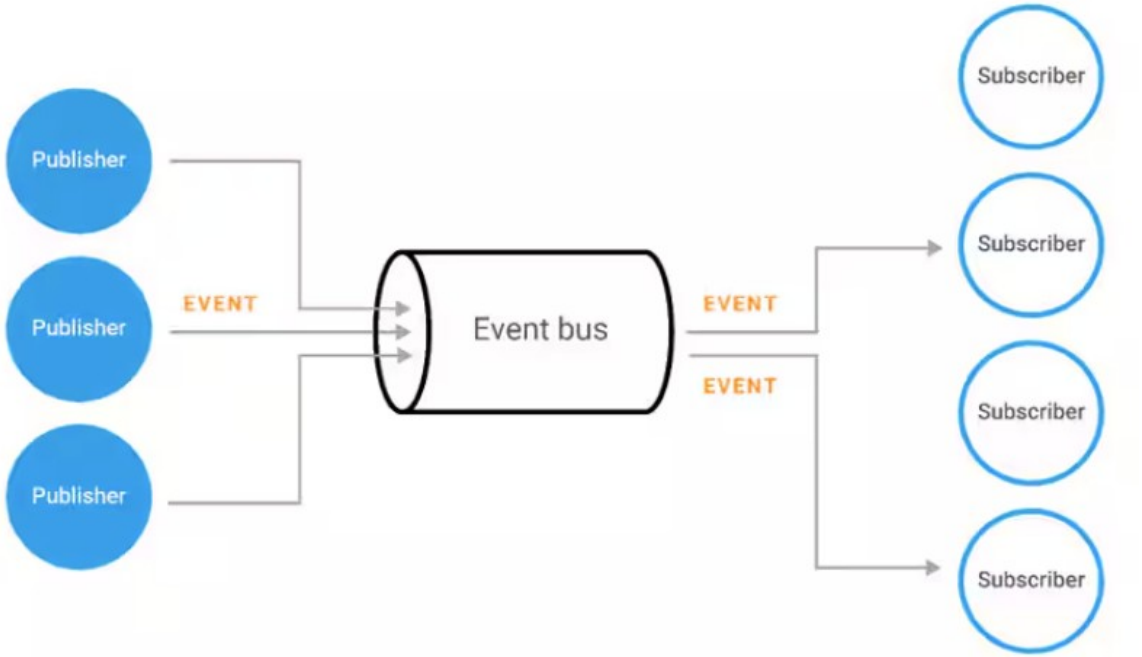
Exemple concret:

```json
{
  "event": "OrderValidated",
  "orderId": "12345",
  "timestamp": "2025-10-16T10:00:00Z"
}
```

Les trois acteurs principaux

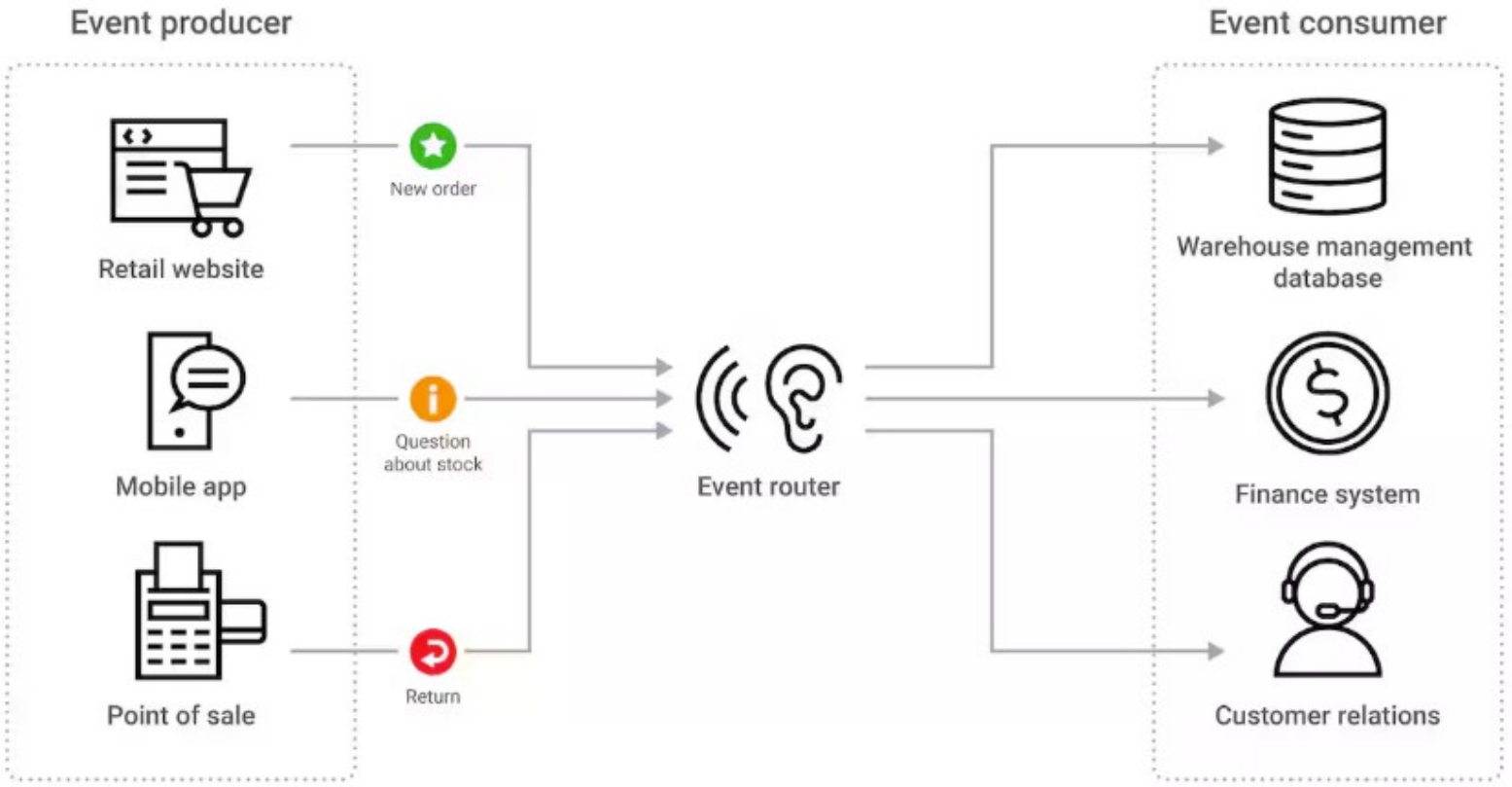
Rôle	Description	Exemple
Émetteur (Publisher)	Génère un événement	Service Paiement
Canal (Event Bus/Broker)	Transporte l'événement	Kafka, RabbitMQ
Consommateur (Subscriber)	Réagit à l'événement	Service Email

Découplage total: l'émetteur ne connaît pas ses consommateurs.



Modèle Publisher / Subscriber

- Le **Publisher** publie un message sans savoir qui l'écoute.
- Le **Subscriber** s'abonne aux types d'événements qui l'intéressent.
- Le **Broker** (intermédiaire) gère la distribution.



Avantages de l’Architecture Orientée Événements

- **Découplage** : Les services communiquent via des événements, sans dépendances directes.
- **Scalabilité** : Les consommateurs peuvent être multipliés facilement.
- **Asynchronisme** : Le système reste réactif même sous charge.
- **Extensibilité** : De nouveaux consommateurs peuvent s’ajouter sans modifier le producteur.
- **Réactivité en temps réel** : Idéal pour les systèmes de monitoring, IoT, finance, etc.

Exemple d’utilisation de l’architecture orienté événements

Un site e-commerce permet à un utilisateur de passer une commande. Quand il valide son panier, plusieurs actions doivent se produire automatiquement :

- Enregistrer la commande dans la base de données.
- Envoyer un e-mail de confirmation.
- Mettre à jour le stock.
- Créer une facture.

Principe Général :

Au lieu d’appeler directement chaque service (méthode classique et synchrone), le système publie un événement unique : “Une commande vient d’être créée.”

Tous les autres services écoutent cet événement et réagissent chacun à leur manière, sans dépendre du service de commande.

Exemple d'utilisation de l'architecture orienté événements : Flux des événements

1. L'utilisateur valide la commande.

Le service "Commande" enregistre la commande en base de données.

2. Le service publie un événement :

"OrderCreated" avec des informations : ID, produits, utilisateur, montant, etc.

3. L'événement est envoyé sur un bus d'événements (Kafka, RabbitMQ...).

4. Plusieurs services abonnés le reçoivent :

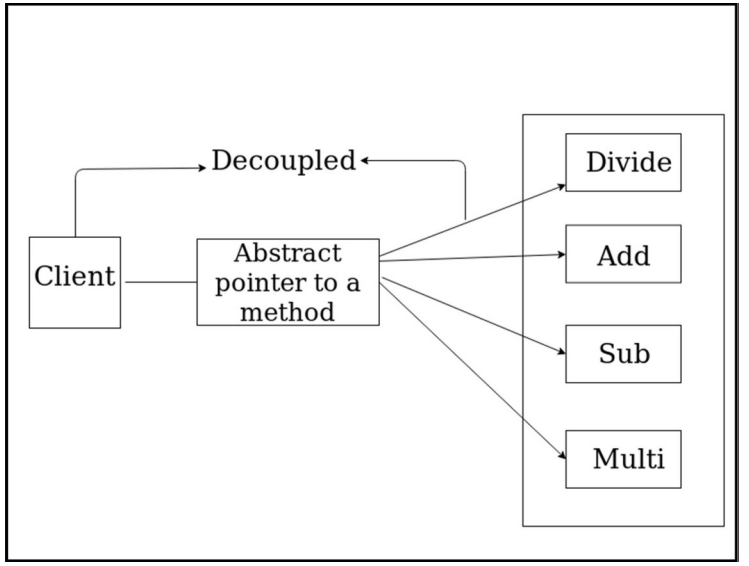
- Le Service Email : envoie un message de confirmation à l'utilisateur.
- Le Service Stock : déduit les articles du stock disponible.
- Le Service Facturation : génère la facture.

5. D'autres services futurs (ex. "Recommandations produit") peuvent écouter le même événement sans modifier les autres composants.

DÉLÉGUÉS

Qu’est-ce qu’un délégué ?

- Un délégué est un type référence de méthode.
- Il permet de stocker une méthode dans une variable et de l’appeler plus tard.
- C’est une brique de base de la programmation orientée événements (Event-Driven Architecture - EDA).



Exemple:

```
public delegate void MonDelegate(string message);

MonDelegate afficher = Console.WriteLine;
afficher("Bonjour !");
```

Analogie: un délégué agit comme une “prise électrique” dans laquelle on peut brancher différentes méthodes.

Délégués prédéfinis

C# fournit déjà des délégués génériques :

- **Action** → méthode sans retour
- **Func<T>** → méthode avec retour
- **Predicate<T>** → méthode retournant un booléen

Exemples:

```
Action<string> log = Console.WriteLine;
Func<int, int, int> add = (x, y) => x + y;
Predicate<int> isEven = n => n % 2 == 0;
```

Délégués multicast

Un délégué peut référencer plusieurs méthodes à la fois

```
void A() => Console.WriteLine("A");
void B() => Console.WriteLine("B");

Action action = A;
action += B;
action(); // Exécute A puis B
```

NB : Les retours de valeurs sont ignorés sauf pour la dernière méthode appelée.

Délégués comme Paramètres

Les délégués permettent de passer des comportements personnalisés à des méthodes.

```
public static void Execute(MessageDelegate action)
{
    action("Exécution personnalisée !");
}

// Appel :
Execute(ShowMessage);
Execute(Alert);
```

Avantages des Délégués

- **Découplage** : le code qui appelle la méthode ne dépend pas de son implémentation.
- **Réutilisation** : permet de passer des fonctions en paramètre.
- **Flexibilité** : change dynamiquement le comportement d'un programme.
- **Base des événements** : utilisé dans toutes les interfaces graphiques, WPF, ASP.NET, etc.

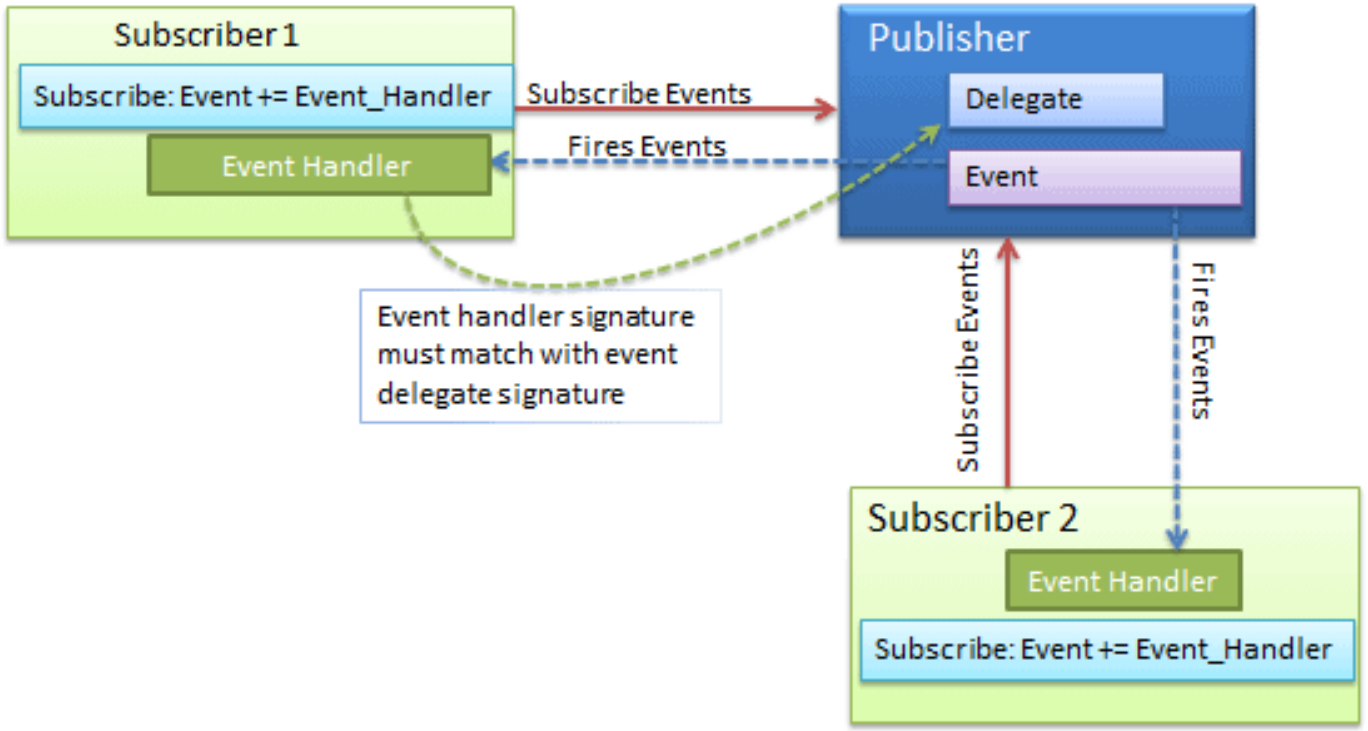
ÉVÉNEMENTS

Qu’est-ce qu’un événement ?

Les événements sont construits sur les délégués. Ils permettent à un objet d’informer d’autres objets qu’une action s’est produite.

```
public class Bouton {
    public event Action OnClick;
    public void Cliquer() => OnClick?.Invoke();
}
```

```
Bouton b = new();
b.OnClick += () => Console.WriteLine("Bouton cliqué !");
b.Cliquer();
```



Mécanisme éditeur / abonné (Publisher/Subscriber)

Publisher : définit l'événement.

Subscriber : s'y abonne avec +=

Invoke : déclenche l'événement.

Exemple:

```
public class Capteur {
    public event Action<int> TemperatureChanged;

    public void SetTemperature(int value) {
        Console.WriteLine($"Température = {value}");
        TemperatureChanged?.Invoke(value);
    }
}

class Program {
    static void Main() {
        var c = new Capteur();
        c.TemperatureChanged += t => Console.WriteLine($"Alerte: {t}°C !");
        c.SetTemperature(30);
    }
}
```

Mécanisme éditeur / abonné (Publisher/Subscriber) : exemple avec plusieurs subscribers

Publisher

```
using System;

public class Capteur
{
    // Event that notifies subscribers when temperature changes
    public event Action<int> TemperatureChanged;

    public void SetTemperature(int value)
    {
        Console.WriteLine($"Température mesurée = {value}°C");
        TemperatureChanged?.Invoke(value); // Notify all subscribers
    }
}
```

Subscriber

```
// Subscriber 1: logs data to a file or console
public class Logger
{
    public void OnTemperatureChanged(int temp)
    {
        Console.WriteLine($"[Logger] Sauvegarde de la température : {temp}°C");
    }
}
```

```
// Subscriber 2: triggers an alarm if too hot
public class Alarm
{
    public void OnTemperatureChanged(int temp)
    {
        if (temp > 25)
            Console.WriteLine($"[ALERTE 🔥] Température critique : {temp}°C !");
        else
            Console.WriteLine($"[Alarm] Température normale ({temp}°C).");
    }
}
```

Mécanisme éditeur / abonné (Publisher/Subscriber) : exemple avec plusieurs subscribers

```
class Program
{
    static void Main()
    {
        var capteur = new Capteur();
        var logger = new Logger();
        var alarm = new Alarm();

        // Subscribe multiple handlers to the same event
        capteur.TemperatureChanged += logger.OnTemperatureChanged;
        capteur.TemperatureChanged += alarm.OnTemperatureChanged;

        // Trigger temperature changes
        capteur.SetTemperature(20);
        capteur.SetTemperature(30);
    }
}
```

GÉNÉRIQUES

Introduction aux génériques

Permettent d'écrire une seule fois du code réutilisable pour plusieurs types. Le type est défini au moment de l'utilisation.

Exemple :

```
class Boite<T> {
    public T Contenu;
    public void Afficher() => Console.WriteLine(Contenu);
}
Boite<int> b = new() { Contenu = 42 };
b.Afficher();
```

Avantages des génériques:

- ✓ Sécurité de type (pas de cast dangereux)
- ✓ Réutilisation du code
- ✓ Meilleures performances (pas de boxing/unboxing)
- ✓ Compatible avec LINQ, Collections, Async, etc.

Méthodes génériques

On peut aussi créer des méthodes génériques dans une classe normale.

```
class Program
{
    // Méthode générique : elle fonctionne avec n'importe quel type T
    public static void Afficher<T>(T valeur)
    {
        Console.WriteLine($"Valeur = {valeur} (type : {typeof(T)})");
    }

    static void Main()
    {
        Afficher<int>(42);           // T = int
        Afficher<string>("Bonjour"); // T = string
        Afficher<double>(3.14);      // T = double
        Afficher<bool>(true);        // T = bool
    }
}
```

Méthodes génériques avec contraintes

On peut aussi utiliser des contraintes génériques pour limiter les types autorisés pour T

```
T Max<T>(T a, T b) where T : IComparable<T>
{
    return a.CompareTo(b) > 0 ? a : b;
}
Console.WriteLine(Max(10, 5));   // 10
Console.WriteLine(Max("A", "B")); // B
```

PROGRAMMATION ASYNCHRONE ET TÂCHES

Pourquoi l'asynchronisme ?

- Les applications modernes (web, desktop, mobile) doivent rester réactives.

- Bloquer le thread principal (UI ou serveur) = mauvaise expérience utilisateur.

Exemple :

```
File.ReadAllText("data.txt"); // Bloque jusqu'à la fin de la lecture
```

Pendant ce temps, l'interface utilisateur se fige ou le serveur ne répond pas.

Solution :

- Utilisation d'**Async** / **Await** avec les **Tasks**.
- L'asynchronisme permet d'exécuter plusieurs opérations en parallèle ou sans bloquer le thread principal.

Le mot clé async:

Déclare une méthode asynchrone: elle peut attendre (await) des opérations non bloquantes.

Exemple :

```
public async Task ChargerDonnéesAsync()
{
    Console.WriteLine("Début du chargement...");
    await Task.Delay(2000); // simulation d'une tâche longue
    Console.WriteLine("Chargement terminé !");
}
```

Ici, **await** libère le thread pendant 2 secondes au lieu de le bloquer.

Le mot clé await:

- Permet d'attendre la fin d'une tâche sans bloquer le reste du programme.
- Le thread principal continue de tourner pendant l'attente.

```
await Task.Delay(3000);
Console.WriteLine("Terminé !");
```

Async et Await : Exemple 1

```
class Program
{
    static async Task Main()
    {
        Console.WriteLine("Début du programme");
        await AfficherMessageAsync();
        Console.WriteLine("Fin du programme");
    }

    static async Task AfficherMessageAsync()
    {
        Console.WriteLine("Chargement...");
        await Task.Delay(2000);
        Console.WriteLine("Message chargé !");
    }
}
```

Sortie :

```
Début du programme
Chargement...
Message chargé !
Fin du programme
```

Async et Await : Exemple 2

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Début du programme");

        // On lance la méthode asynchrone sans attendre
        Task t = AfficherMessageAsync();

        Console.WriteLine("Le programme continue pendant le chargement...");

        // On attend la fin de la tâche avant de fermer le programme
        t.Wait();

        Console.WriteLine("Fin du programme");
    }

    static async Task AfficherMessageAsync()
    {
        Console.WriteLine("Chargement en cours...");
        await Task.Delay(2000); // pause de 2 secondes
        Console.WriteLine("✅ Message chargé !");
    }
}
```

Sortie ?

Async et Await : Exemple 2

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Début du programme");

        // On lance la méthode asynchrone sans attendre
        Task t = AfficherMessageAsync();

        Console.WriteLine("Le programme continue pendant le chargement...");

        // On attend la fin de la tâche avant de fermer le programme
        t.Wait();

        Console.WriteLine("Fin du programme");
    }

    static async Task AfficherMessageAsync()
    {
        Console.WriteLine("Chargement en cours...");
        await Task.Delay(2000); // pause de 2 secondes
        Console.WriteLine("✅ Message chargé !");
    }
}
```

Sortie:

```
Début du programme
Chargement en cours...
Le programme continue pendant le chargement...
✅ Message chargé !
Fin du programme
```

Async et Await : Exemple 3

```
static void Main(string[] args)
{
    Console.WriteLine("Début du programme");

    // On démarre une méthode asynchrone sans l'attendre
    Task tache = AfficherMessageAsync();

    // Le programme continue pendant que la tâche s'exécute
    Console.WriteLine("Le programme continue son exécution...");

    // Boucle principale (travail parallèle simulé)
    for (int i = 1; i <= 3; i++)
    {
        Console.WriteLine($"Travail principal étape {i}");
        Task.Delay(500).Wait(); // petite pause pour la simulation
    }

    // On attend la fin de la tâche avant de quitter
    tache.Wait();

    Console.WriteLine("Fin du programme");
}
```

```
static async Task AfficherMessageAsync()
{
    Console.WriteLine("Chargement du message...");
    await Task.Delay(2000); // simulation d'un travail long (2 secondes)
    Console.WriteLine("✅ Message chargé !");
}
```

Sortie ?

Async et Await : Exemple 3

```
static async Task Main(string[] args)
{
    Console.WriteLine("Début du programme");

    // On démarre la tâche asynchrone
    Task tache = AfficherMessageAsync();

    // Le programme continue pendant que la tâche s'exécute
    Console.WriteLine("Le programme continue son exécution...");

    // Boucle principale (travail parallèle simulé)
    for (int i = 1; i <= 3; i++)
    {
        Console.WriteLine($"Travail principal étape {i}");
        await Task.Delay(500); // petite pause pour la simulation
    }

    // Ici on attend la fin de la tâche asynchrone proprement
    await tache;

    Console.WriteLine("Fin du programme");
}
```

```
static async Task AfficherMessageAsync()
{
    Console.WriteLine("Chargement du message...");
    await Task.Delay(2000); // simulation d'un travail long (2 secondes)
    Console.WriteLine("✅ Message chargé !");
}
```

Sortie ?

```
Début du programme
Chargement du message...
Le programme continue son exécution...
Travail principal étape 1
Travail principal étape 2
Travail principal étape 3
✅ Message chargé !
Fin du programme
```

Async et Await : Exemple 4

```
static async Task Main(string[] args)
{
    Console.WriteLine("Début du programme");

    // On démarre et attend directement la tâche asynchrone
    await AfficherMessageAsync();

    // Le reste du programme s'exécute après la fin de la tâche
    Console.WriteLine("Le programme continue son exécution...");

    for (int i = 1; i <= 3; i++)
    {
        Console.WriteLine($"Travail principal étape {i}");
        await Task.Delay(500);
    }

    Console.WriteLine("Fin du programme");
}
```

```
static async Task AfficherMessageAsync()
{
    Console.WriteLine("Chargement du message...");
    await Task.Delay(2000);
    Console.WriteLine("✅ Message chargé !");
}
```

Sortie ?

Async et Await : Exemple 4

```
static async Task Main(string[] args)
{
    Console.WriteLine("Début du programme");

    // On démarre et attend directement la tâche asynchrone
    await AfficherMessageAsync();

    // Le reste du programme s'exécute après la fin de la tâche
    Console.WriteLine("Le programme continue son exécution...");

    for (int i = 1; i <= 3; i++)
    {
        Console.WriteLine($"Travail principal étape {i}");
        await Task.Delay(500);
    }

    Console.WriteLine("Fin du programme");
}
```

```
static async Task AfficherMessageAsync()
{
    Console.WriteLine("Chargement du message...");
    await Task.Delay(2000);
    Console.WriteLine("✅ Message chargé !");
}
```

Sortie ?

```
Début du programme
Chargement du message...
✅ Message chargé !
Le programme continue son exécution...
Travail principal étape 1
Travail principal étape 2
Travail principal étape 3
Fin du programme
```

La classe Task

Task représente une opération asynchrone.

Exemple :

```
Task t = Task.Run(() => Console.WriteLine("Tâche exécutée"));
await t;
```

Avec retour de valeur :

```
Task<int> t = Task.Run(() => 42);
int result = await t;
Console.WriteLine(result);
```

NB : **await** ne s'utilise que sur une méthode asynchrone (retournant **Task** ou **Task<T>**), car il doit attendre une opération non bloquante.

INJECTION DE DÉPENDANCE (Dependency Injection)

Qu’est-ce qu’une dépendance ?

Une dépendance est un objet dont une autre classe a besoin pour fonctionner.

Exemple simple :

```
public class ServiceEmail
{
    public void Envoyer(string msg)
    {
        Console.WriteLine("Email envoyé : " + msg);
    }
}

public class Client
{
    private ServiceEmail _email = new ServiceEmail(); // dépendance directe
}
```

Ici, **Client** dépend directement de **ServiceEmail**.

C’est un **couplage fort** : difficile à tester, à changer ou à étendre.

Problème du couplage fort

- **Difficile à tester** : Impossible de remplacer ServiceEmail par un faux (mock)
- **Difficile à faire évoluer** : Si on change ServiceEmail, on doit modifier Client.
- **Non réutilisable** : Client ne peut pas utiliser un autre service sans modification.
- **Code rigide** : Dépendance créée “en dur” dans la classe.

Solution : l’Inversion de Contrôle (IoC) :

L’objet ne crée plus lui-même ses dépendances. On les fournit depuis l’extérieur.

Exemple sans DI (mauvaise pratique)

Client crée lui-même son ServiceEmail → dépendance rigide.

```
public class Client
{
    private ServiceEmail _email = new ServiceEmail();

    public void EnvoyerNotification()
    {
        _email.Envoyer("Message client");
    }
}
```

Exemple avec Injection de Dépendance

```
public interface IEmailService
{
    void Envoyer(string msg);
}

public class ServiceEmail : IEmailService
{
    public void Envoyer(string msg) => Console.WriteLine("Email : " + msg);
}

public class Client
{
    private readonly IEmailService _email;

    public Client(IEmailService email)
    {
        _email = email;
    }

    public void EnvoyerNotification()
    {
        _email.Envoyer("Message client");
    }
}
```

Utilisation :

```
var email = new ServiceEmail();
var client = new Client(email);
client.EnvoyerNotification();
```

Client ne dépend plus d'une implémentation concrète, mais d'une abstraction (interface).

Avantages de l'injection de dépendance

- **Découplage** : Les classes dépendent d'abstractions, pas d'implémentations
- **Testabilité** : Facile d'injecter des mocks pour les tests unitaires
- **Flexibilité** : On peut changer le service sans toucher au code métier
- **Lisibilité** : Les dépendances sont explicites
- **Maintenance** : Code plus propre, plus modulaire