# Microservizi ES

## Introduzione

Oggi costruiamo insieme un piccolo sistema a microservizi in C#.

L'obiettivo non è tanto scrivere tanto codice, ma **capire come si progetta**: come dividere responsabilità, come far comunicare servizi diversi, e come mantenere il codice **mantenibile, testabile e scalabile**.

I microservizi sono essenzialmente **piccoli moduli indipendenti**, ognuno con una responsabilità precisa, che comunicano tra loro tramite messaggi o API.

Nel nostro caso, avremo tre servizi principali e una libreria condivisa:

- `UserService` → gestisce e valida gli utenti
- `OrderService` → crea e registra ordini
- `Gateway` → fa da orchestratore e punto di accesso esterno
- `Shared` → contiene i contratti comuni: DTO e interfacce

Ogni servizio sarà autonomo ma coerente con gli altri grazie a interfacce e DTO condivisi.

---

## Principi di base

Ogni volta che lavoriamo a un microservizio, dobbiamo pensare ai **principi SOLID**, che qui tornano tutti:

- **SRP**: ogni servizio fa una cosa sola.
- **OCP**: possiamo estendere i comportamenti (ad esempio nuove strategie di retry o nuovi repository) senza modificare quelli esistenti.
- **DIP**: dipendiamo sempre da astrazioni, mai da classi concrete.
- **ISP**: ogni interfaccia espone solo ciò che serve.
- **LSP**: le sostituzioni devono mantenere il comportamento atteso.

La guida che avete in mano è divisa in **quattro milestone**: ognuna aggiunge un tassello di complessità e vi fa vedere come si costruisce un sistema robusto, passo dopo passo.

---

# Milestone 1 — UserService (Logging, Repository, Validazione)

Partiamo dal primo microservizio: **UserService**.

È il più semplice, ma fondamentale per capire come si costruisce una base pulita.

## Spiegazione concettuale

UserService è responsabile della gestione degli utenti: validare input, restituire i dati corretti e gestire eventuali errori.

Il concetto chiave è **Single Responsibility Principle** — fa solo questo.

Non conosce gli ordini, non gestisce notifiche, non parla direttamente con il gateway se non tramite interfacce.

Per mantenerlo pulito, separiamo tre responsabilità:

1. **Repository** → dove i dati vengono salvati (anche solo in memoria all'inizio).
2. **Servizio di dominio** → logica di validazione e accesso ai dati.
3. **Logger** → cattura ciò che accade senza mescolarsi alla logica di business.

In termini di codice, questo si traduce in interfacce come:

```
public interface IUserRepository
{
    UserDto? GetById(Guid id);
    void Add(UserDto user);
}

public interface ILogger
{
    void Log(string message);
}
```

E poi l'implementazione:

```
public class InMemoryUserRepository : IUserRepository
{
    private readonly Dictionary<Guid, UserDto> _users = new();
```

```
    public UserDto? GetById(Guid id) => _users.GetValueOrDefault(id);
    public void Add(UserDto user) => _users[user.Id] = user;
}
```

Nel costruttore del servizio, **iniettiamo** il repository e il logger:

```
public class UserReadService
{
    private readonly IUserRepository _repo;
    private readonly ILogger _log;

    public UserReadService(IUserRepository repo, ILogger log)
    {
        _repo = repo;
        _log = log;
    }

    public OperationResult<UserDto> GetUser(Guid id)
    {
        _log.Log($"Richiesta utente {id}");
        var user = _repo.GetById(id);
        return user != null
            ? OperationResult<UserDto>.Ok(user)
            : OperationResult<UserDto>.Fail("Utente non trovato");
    }
}
```

Vedete che qui applichiamo già **Dependency Inversion**: la classe non sa che tipo di repository usa, né come funziona il logger. Sa solo che esistono interfacce con quei metodi.

Questo rende tutto testabile: nei test potremmo passare un `FakeUserRepository` o un `MockLogger` e verificare il comportamento senza toccare la logica interna.

---

## Spiegazione didattica

È importante che capiate una cosa: la validazione e il logging non sono decorazioni, ma strumenti per garantire che il servizio sia coerente, osservabile e robusto.

Ogni microservizio deve poter dire *“so fare bene solo una cosa, e so spiegare quando qualcosa va storto”*.

Per questo anche un logger o una struttura dati in memoria sono parte dell’architettura.

---

# Milestone 2 — OrderService (Logica di dominio e Idempotenza)

Ora che abbiamo il nostro UserService stabile e ben isolato, passiamo al secondo microservizio: **OrderService**.

Qui le cose si fanno più interessanti, perché entra in gioco la logica di dominio — cioè la capacità del sistema di *decidere cosa è un ordine valido* e *come viene registrato* — e un concetto chiave dei microservizi: **l’idempotenza**.

---

## Concetto di Idempotenza

L’idempotenza significa che, anche se una stessa operazione viene inviata più volte, il risultato non cambia.

Immaginate che il Gateway ritenti una richiesta d’ordine perché il server ha risposto lentamente.

Noi non vogliamo che vengano creati due ordini identici.

Il servizio deve riconoscere che si tratta della stessa operazione e restituire la stessa risposta.

Questo è fondamentale nei sistemi distribuiti, dove *il fallimento è normale* — un timeout, una disconnessione, un retry automatico — ma il sistema deve comunque garantire coerenza.

---

## Struttura del servizio

Come sempre, partiamo da un’interfaccia chiara per il repository e per la logica del servizio.

```
public interface IOrderRepository
{
    bool Exists(Guid orderId);
```

```
    void Add(Order order);
}
```

L'implementazione base può essere un repository in memoria, ma **con concorrenza protetta**, perché più richieste potrebbero arrivare in parallelo.

```
public class InMemoryOrderRepository : IOrderRepository
{
    private readonly ConcurrentDictionary<Guid, Order> _orders = new();

    public bool Exists(Guid orderId) => _orders.ContainsKey(orderId);

    public void Add(Order order)
    {
        if (!_orders.TryAdd(order.Id, order))
            throw new InvalidOperationException("Ordine duplicato.");
    }
}
```

Qui usiamo `ConcurrentDictionary` per garantire **thread safety**, cioè che due thread non scrivano lo stesso ordine nello stesso momento.

Questo serve a simulare la concorrenza che, in un sistema reale, avremmo tra richieste HTTP simultanee.

---

## La logica di dominio (SRP + DIP)

L'OrderService deve occuparsi solo di una cosa: ricevere una richiesta d'ordine e decidere se e come accettarla.

Tutto il resto — salvataggio, logging, retry — è responsabilità di altri livelli.

Ecco un esempio di implementazione minimale:

```
public class OrderService
{
    private readonly IOrderRepository _repo;
    private readonly ILogger _log;

    public OrderService(IOrderRepository repo, ILogger log)
    {
```

```
        _repo = repo;
        _log = log;
    }

    public OperationResult<OrderConfirmationDto> CreateOrder(OrderDto dt
o)
    {
        if (_repo.Exists(dto.Id))
        {
            _log.Log($"Ordine duplicato: {dto.Id}");
            return OperationResult<OrderConfirmationDto>.Fail("Ordine già esis
tente");
        }

        var order = new Order(dto.Id, dto.UserId, dto.Amount);
        _repo.Add(order);
        _log.Log($"Ordine creato: {dto.Id}");
        return OperationResult<OrderConfirmationDto>.Ok(new OrderConfirm
ationDto(order.Id, "Confermato"));
    }
}
```

## Spiegazione didattica

Notate come la classe `OrderService` non sappia nulla dell'origine della richiesta.

Non sa se viene dal Gateway, da un messaggio in coda o da un'interfaccia web.

Questo è **Dependency Inversion** nella pratica: l'ordine dei livelli è invertito.

Il servizio conosce solo il contratto (l'interfaccia), non il contesto.

Vedete anche un altro principio in azione: **Open/Closed Principle**.

Se volessimo introdurre una logica più complessa, ad esempio sconti, validazioni aggiuntive, o tipi diversi di ordine, possiamo farlo estendendo, non modificando, la classe esistente.

Possiamo decorarla, oppure creare varianti che implementano la stessa interfaccia.

## Riflessione

Con questa milestone avete completato il cuore del dominio.

Ora avete due microservizi autonomi che non si conoscono ma parlano la stessa lingua, grazie a un livello `Shared` comune che definisce i contratti.

Questa è la base dell'architettura a microservizi: ogni servizio è indipendente, ma interoperabile.

Ognuno è testabile singolarmente e riutilizzabile in contesti diversi.

---

# Milestone 3 — Gateway (Orchestrazione e Validazione delle Richieste)

Ora mettiamo insieme i pezzi.

Abbiamo due microservizi funzionanti — `UserService` e `OrderService` — e una libreria `Shared` che definisce i contratti comuni.

Serve però qualcuno che *coordini* le chiamate, che sappia in che ordine fare le cose e come gestire gli errori.

Questo qualcuno è il **Gateway**.

---

## Ruolo del Gateway

Il Gateway è come un direttore d'orchestra:

riceve una richiesta dall'esterno, la valida, la inoltra ai microservizi corretti, raccoglie le risposte e compone il risultato finale.

Nel nostro caso, il Gateway riceve una richiesta di creazione ordine.

Prima di tutto controlla che l'utente esista (chiamando `UserService`), poi crea l'ordine (chiamando `OrderService`), e infine restituisce al client una risposta unificata.

Il tutto deve avvenire in modo sicuro, con:

- **validazione input**,
- **retry automatico** in caso di errore temporaneo,
- **timeout**,
- e **logging** con un `correlationId`, per tracciare l'intera catena di operazioni.

---

## Il codice dell'orchestrazione

Partiamo da una struttura semplice.

Immaginate che il Gateway esponga un metodo pubblico `PlaceOrder` che coordina tutto:

```
public class OrderGateway
{
    private readonly IUserReadService _userService;
    private readonly IOrderService _orderService;
    private readonly ILogger _log;

    public OrderGateway(IUserReadService userService, IOrderService orde
rService, ILogger log)
    {
        _userService = userService;
        _orderService = orderService;
        _log = log;
    }

    public OperationResult<OrderConfirmationDto> PlaceOrder(Guid userId,
decimal amount)
    {
        var correlationId = Guid.NewGuid();
        _log.Log($"[{correlationId}] Inizio richiesta ordine per utente {userI
d}");

        // 1. Validazione utente
        var userResult = _userService.GetUser(userId);
        if (!userResult.IsSuccess)
        {
            _log.Log($"[{correlationId}] Utente non valido");
            return OperationResult<OrderConfirmationDto>.Fail("Utente non tro
vato");
        }

        // 2. Creazione ordine
        var orderDto = new OrderDto(Guid.NewGuid(), userId, amount);
        var orderResult = _orderService.CreateOrder(orderDto);
```

```
        if (!orderResult.IsSuccess)
        {
            _log.Log($"[{correlationId}] Errore nella creazione ordine");
            return OperationResult<OrderConfirmationDto>.Fail(orderResult.Err
or);
        }

        _log.Log($"[{correlationId}] Ordine completato");
        return orderResult;
    }
}
```

---

## Spiegazione didattica

Vedete che il Gateway non contiene logica di dominio.

Non decide come validare un utente o come creare un ordine.

Il suo compito è coordinare i microservizi.

Questo è un pattern molto importante nel mondo enterprise: si chiama **API Gateway Pattern**.

Serve a *centralizzare* l'accesso ai microservizi senza rompere la loro indipendenza.

Ogni microservizio resta autonomo, ma l'orchestrazione — cioè l'ordine delle chiamate e la gestione dei fallimenti — è gestita in un punto unico, che si può anche scalare separatamente.

---

## Gestione degli errori e resilienza

Ora introduciamo un po' di resilienza.

In un ambiente distribuito reale, le richieste possono fallire per mille motivi: rete lenta, servizio non disponibile, eccezioni temporanee.

Ma ricordate: *un fallimento temporaneo non deve diventare un errore permanente.*

Per questo introduciamo:

- un **retry** con tentativi limitati,
- un **timeout** per evitare blocchi infiniti,

- e un **OperationResult** per rappresentare in modo esplicito successo o errore.

Ecco una versione con retry simulato:

```
private OperationResult<T> Retry<T>(Func<OperationResult<T>> action, i
nt maxRetries = 3)
{
    for (int i = 0; i < maxRetries; i++)
    {
        var result = action();
        if (result.IsSuccess)
            return result;

        _log.Log($"Tentativo {i + 1}/{maxRetries} fallito: {result.Error}");
        Thread.Sleep(200); // simuliamo un piccolo delay tra i retry
    }

    return OperationResult<T>.Fail("Tutti i tentativi falliti");
}
```

Nel codice reale, questo Retry sarebbe usato per chiamare i servizi, così:

```
var userResult = Retry(() ⇒ _userService.GetUser(userId));
```

In questo modo, il sistema diventa più robusto senza complicare la logica dei singoli microservizi.

È una forma di **separazione delle responsabilità**: i servizi restano semplici, e la resilienza viene gestita a livello di orchestrazione.

---

## Logging con correlationId

Infine, introduciamo il concetto di `correlationId`.

In un sistema distribuito è fondamentale poter seguire il flusso di una singola richiesta.

Il correlationId è un identificatore unico che accompagna ogni log, ogni chiamata, ogni errore.

Questo permette di risalire a cosa è successo, anche se i log sono su server diversi.

È una pratica di osservabilità essenziale in architetture a microservizi.

---

### Riflessione

Questa milestone segna un salto di livello:

non stiamo più costruendo "pezzi di codice", ma un sistema coordinato e resiliente.

Ora i vostri studenti iniziano a capire cosa significa **progettare un'architettura**, non solo scrivere classi.

Qui puoi sottolineare come tutto ciò che hanno visto — SRP, DIP, OCP — converge naturalmente verso una struttura mantenibile.

---

## Milestone 4 — Resilienza e Robustezza Distribuita

Bene, ora che abbiamo UserService, OrderService e il nostro Gateway, possiamo affrontare l'ultimo livello: la **resilienza**.

Finora abbiamo costruito un sistema che funziona *quando tutto va bene*.

Ma nei microservizi il punto non è "funziona", è "continua a funzionare anche se qualcosa fallisce".

E questo cambia completamente il modo in cui si scrive codice.

---

### Cos'è la resilienza

Resilienza significa che il sistema:

- *si riprende da errori temporanei* (retry, fallback),
- *evita blocchi lunghi* (timeout),
- *degrada con grazia* invece di crashare (circuit breaker, response default),
- e *fornisce sempre un feedback chiaro* (OperationResult).

Questi sono concetti fondamentali in un mondo distribuito.

Quando un servizio dipende da un altro, la domanda non è "se" fallirà, ma "quando".

---

## Strumenti che introduciamo in questa milestone

1. `HttpClientSimulator` — per simulare chiamate di rete lente o fallite.
2. `RetryPolicy` — per riprovare un'operazione un certo numero di volte.
3. `TimeoutPolicy` — per evitare che un'operazione blocchi tutto il flusso.
4. `OperationResult<T>` — per rappresentare in modo uniforme il successo o il fallimento.

---

## OperationResult: un contratto di comunicazione chiaro

Partiamo da `OperationResult<T>`.

È un oggetto che incapsula sia il valore del risultato, sia lo stato dell'operazione.

È molto meglio di eccezioni non gestite o codici di errore sparsi in giro.

Ecco un esempio:

```
public class OperationResult<T>
{
    public bool IsSuccess { get; }
    public string? Error { get; }
    public T? Value { get; }

    private OperationResult(bool isSuccess, T? value, string? error)
    {
        IsSuccess = isSuccess;
        Value = value;
        Error = error;
    }

    public static OperationResult<T> Ok(T value) ⇒ new(true, value, null);
    public static OperationResult<T> Fail(string error) ⇒ new(false, default, e
rror);
}
```

Ora ogni servizio può restituire un `OperationResult` e lasciare al chiamante decidere cosa fare con l'errore.

Questo semplifica tantissimo i flussi complessi.

---

## HttpClientSimulator: simulare fallimenti reali

Perché parliamo di resilienza se il codice è locale?

Perché dobbiamo *abituarci a ragionare come se fossimo in produzione*, dove ogni microservizio può trovarsi su un server diverso.

`HttpClientSimulator` serve proprio a questo: simulare le chiamate HTTP con ritardi, errori o timeouts casuali.

È una classe semplice, ma potentissima didatticamente.

```
public class HttpClientSimulator
{
    private readonly Random _rnd = new();

    public async Task<OperationResult<T>> CallAsync<T>(Func<Task<T>>
action, int maxDelay = 1000)
    {
        try
        {
            await Task.Delay(_rnd.Next(100, maxDelay)); // simula latenza
            if (_rnd.NextDouble() < 0.2) // 20% fallimenti simulati
                throw new TimeoutException("Simulated timeout");

            var result = await action();
            return OperationResult<T>.Ok(result);
        }
        catch (Exception ex)
        {
            return OperationResult<T>.Fail(ex.Message);
        }
    }
}
```

In pratica, ogni volta che chiamiamo un servizio, lo facciamo passare da qui.

Così possiamo vedere come si comporta il sistema se qualcosa va storto.

---

## Timeout e Retry combinati

Ora integriamo due concetti fondamentali: *timeout* e *retry*.

Un timeout serve a dire “non aspettare troppo”.

Un retry serve a dire “riprovaci, forse era un errore momentaneo”.

Combinati, diventano una strategia solida per la resilienza.

Ecco un esempio:

```
private async Task<OperationResult<T>> CallWithRetryAsync<T>(
    Func<Task<OperationResult<T>>> action,
    int maxRetries = 3,
    int timeoutMs = 500)
{
    for (int attempt = 1; attempt <= maxRetries; attempt++)
    {
        using var cts = new CancellationTokenSource(timeoutMs);
        try
        {
            var task = action();
            var completed = await Task.WhenAny(task, Task.Delay(timeoutMs,
cts.Token));
            if (completed == task)
                return await task;

            _log.Log($"Tentativo {attempt}: timeout scaduto dopo {timeoutMs}
ms");
        }
        catch (Exception ex)
        {
            _log.Log($"Tentativo {attempt}: errore {ex.Message}");
        }

        await Task.Delay(200); // piccola pausa tra i retry
    }

    return OperationResult<T>.Fail("Tutti i tentativi falliti");
}
```

Con questa logica, il Gateway diventa *davvero resiliente*:

ogni chiamata è gestita in modo controllato, ogni fallimento produce un messaggio chiaro, e il sistema non blocca mai il flusso principale.

---

## Concetto finale: “Fallo bene una volta, non aggiustarlo cento volte”

Questa è la filosofia di fondo della milestone:

la resilienza non è un’aggiunta, è una *caratteristica di design*.

Un sistema ben progettato non ha bisogno di essere continuamente rattoppato, perché è nato per gestire il fallimento.

In termini di principi SOLID:

- DIP: il Gateway non dipende da implementazioni concrete di retry o timeout.

  Potremmo sostituirle con altre strategie senza cambiare codice.
- SRP: ogni componente fa una cosa sola (retry gestisce retry, timeout gestisce timeout).
- OCP: possiamo estendere la resilienza (es. circuit breaker, exponential backoff) senza toccare il codice base.

---

## Conclusione e riflessione finale

Ora il quadro è completo.

Abbiamo costruito:

- Servizi autonomi (User, Order) → **SRP, DIP**
- Orchestrazione con Gateway → **API Gateway Pattern**
- Resilienza e logging → **Observer e Retry Pattern**

Il risultato è un piccolo ecosistema C# che *sembra semplice ma contiene tutti i principi architetturali moderni*:

testabilità, modularità, estendibilità e robustezza.

E il messaggio finale che voglio lasciarvi è questo:

> Non scrivete mai codice solo per farlo funzionare.
>
> Scrivetelo perché possa continuare a funzionare anche quando il mondo intorno smette di farlo.

Questo è il senso dei microservizi. Non è il numero di classi o progetti, ma *la mentalità architetturale* che ci sta dietro.